Extreme API with Python

For Extreme Networks Products

Part Number 9036931-00 Rev AA

February 2021

Copyright © 2021 Extreme Networks, Inc. All rights reserved.

Table of Contents

1 Preface 5
1.1 References 5
1.2 Acknowledgements 5
2 Introduction 6
2.1 Using Python 6
2.1.1 Install Python 6
2.1.2 Update Your PATH 6
2.1.3 Virtual Environment 7
2.1.4 PIP 7
2.1.5 Editors and IDE 7
2.2 REST APIs 8
2.2.1 URLs 8
2.2.2 HTTP Status Codes 9
2.2.3 HTTP Request Methods 9
2.2.4 HTTP Headers 9
2.2.5 Manipulating Headers with Python 11
2.3 Authentication and Authorization 12
2.3.1 Basic Authentication 12
2.3.2 Bearer Authentication 12
2.3.3 API Key 13
2.3.4 OAuth 2.0 13
2.3.5 Managing Passwords or Tokens with Python 13
2.4 Understanding JSON 15
2.5 Manipulating JSON with Python 16
2.6 Interact with a REST API using Python 18
2.6.1 Urllib 18

2.6.2 Requests 20
2.6.3 Testing a REST API 25
2.7 Webhooks 27
2.8 HTTPS with Python 28
3 EXOS APIs 29
3.1 On-Switch APIs 29
3.1.1 Python Scripting 29
3.1.2 Python Application 35
3.2 External APIs 40
3.2.1 RESTCONF API 41
3.2.2 JSON-RPC API 49
4 VOSS API 55
4.1 VOSS RESTCONF Documentation 55
4.2 Enable RESTCONF 55
4.3 Use RESTCONF with Python 55
4.4 EXOS & VOSS Restconf Python Classes 58
5 XMC API 59
5.1 Python Scripting Engine 59
5.1.1 Default Location for Scripts 59
5.1.2 Add a User-Created Script 59
5.1.3 Python Modules Shipped with XMC 59
5.1.4 System Path and Precedence 60
5.1.5 Install a Library 60
5.1.6 XMC Python Module 60
5.2 Workflow Engine 69
5.2.1 emc_vars 69
5.2.2 Create Workflows 71
5.2.3 Create Variables 73
5.2.4 emc_results 77

5.2.5 Add User Inputs ... 81
5.2.6 Automate Workflow Execution ... 83
5.2.7 Workflow Example ... 88
5.3 NorthBound Interface API ... 93
5.3.1 emc_nbi ... 94
5.3.2 GraphQL Query ... 94
5.3.3 GraphQL Mutation ... 100
5.3.4 RBAC for API Usage ... 102
5.3.5 External Access to the NBI API ... 103
5.3.6 Use NBI to Execute a Workflow ... 106
5.4 Axis API ... 109
5.4.1 Analytics Methods ... 109
5.4.2 Analytics API with Python ... 113

6 ExtremeCloud IQ API ... 116

6.1 Connect to the xAPI ... 116
6.1.1 Create Tokens ... 116
6.1.2 Headers for the xAPI ... 119
6.1.3 xAPI Endpoints ... 119
6.1.4 Parameters ... 121
6.2 Use Python with XIQ ... 123
6.2.1 Use GET ... 123
6.2.2 Use POST ... 125
6.2.3 Use Webhooks ... 128

7 Extreme Campus Controller API ... 134

7.1 Set Up Authorization ... 134
7.2 Use GET method ... 136
7.3 Use POST ... 139
7.4 Use PUT ... 141
7.5 Use DELETE ... 143

1 Preface

This document is provided for information only, to share technical experience and knowledge. Do not use this document to validate designs, features, or scalability.

1.1 References

The following references are used extensively in the preparation of this document:

EXOS 30.6 User Guide

EXOS 30.6 RestConf Developer Guide

XMC 8.4.3 GraphQL API

Configuring User Interfaces and Operating Systems for VOSS 8.1.5

ExtremeCloud IQ Developer Portal

Engineering API/Application Documentation

Extreme Developer Center

Extreme Networks product documentation (software)

RFC 8040 RESTCONF Protocol

RFC 8343 A Yang Data Model for Interface Management

RFC 8259 The JavaScript Object Notation (JSON) Data Interchange Format

HTTPS://www.python.org/

HTTPS://www.openconfig.net/

1.2 Acknowledgements

Document author: Stéphane Grosjean, Principal Systems Engineer

Content in this document is based on training modules provided by Markus Nispel and lab guides developed by the Extreme Systems Engineering team.

2 Introduction

This document provides an easy approach to the various APIs within Extreme Networks solutions.

The programming language used here is Python 3, but other languages can also be used. The Python 3 was selected based on how easy it is for beginners to learn, and its wide use in the market.

2.1 Using Python

At the time this document was written, Python had two major versions: Python 2.7 and Python 3. Although Python 2 has seen wide use and is often the default version of the programming language installed on OS (Linux, MacOS), it has reached an end-of-support milestone (January 1st, 2020) and no further development is planned. The current version of Python 2 is 2.7.18, released in April 2020. As a result, this document references only the most recent version of Python 3, which is 3.8.3.

2.1.1 Install Python

This document does not address Python installation details. There are many resources available in blogs, YouTube videos, and books.

The recommendation is to download the latest version of Python 3 directly from the Python website (HTTPS://www.python.org/downloads/) if you haven't already installed it on your system. Linux and MacOS had Python 2.7 pre-installed, however it is not included by default with Windows.

Use the commands shown below to verify which release you have from the command shell. These are examples for Windows 10 and Ubuntu 20.04:

```
C:\Users\stgrosjean>python --version
Python 3.7.7
```

```
stef@ubuntu-lab:~$ python --version
Python 2.7.18rc1
stef@ubuntu-lab:~$ python3 --version
Python 3.8.2
```

Note: On the Ubuntu output, we have both Python 2 and Python 3 installed. To differentiate between the two, you need to type either python (for Python 2) or python3 (for Python 3).

2.1.2 Update Your PATH

Update your PATH system variable to execute Python from anywhere on your PC. The most recent Python installer can do this for you automatically.

2.1.3 Virtual Environment

The easiest way to manage multiple packages, modules and libraries is to work with virtual environments. If you do not use a virtual environment, every time you install a new package, module, or library, it is added into the global Python installation. This becomes problematic if you install packages with dependencies that may require older or more recent versions of modules you are already using. In these cases, some applications can fail as versions of existing modules are changed.

A virtual environment creates a fresh copy of the Python global environment. You can create multiple virtual environments and install the packages you want only into the environment you specify, without breaking existing installations and applications.

HTTPS://docs.python.org/3/library/venv.html

***Note**: How you create and manage virtual environments has changed with Python 3.6. Make sure to use the recommended methods.*

2.1.4 PIP

The best tool for installing new modules and libraries is PIP, which is highly recommended, and installed by default with Python since release 3.4.

HTTPS://docs.python.org/3/installing/index.html

2.1.5 Editors and IDE

You will need a text editor to work with a programming language. Although nearly any text editor will do the job, specialized editors and IDEs (integrated development environments) can help simplify your workflow.

Many popular text editors and IDEs have Python support built-in, for example:

- Vim
- Sublime Text
- Notepad++
- Visual Studio Code
  PyCharm
  Spyder
  Jupyter

These editors and environments may provide a color scheme to quickly identify reserved words, functions, and variables. Many also have an integrated help system, and intelligent auto-completion. Depending on the tool, auto-completion may propose functions and methods associated with variables, depending on various factors.

Typically, if you use a simple text editor, you test your code from the command line in a separate window. Using Python, simply type **python** (or **python3**) to enter the Python interactive shell and type and test your code.

***Note**: If you need more information about a function, method or library, Python has a built-in help system. In the Python interactive shell, type* `help(<method>)` *or* `dir(<method>)`*.*

If you use an IDE, you can run code directly from the editor and, with more advanced IDEs, you can manage virtual environments and execute selected portions of code. Spyder and Jupyter are part of the Anaconda distribution, which is a typical environment for data science.

2.2 REST APIs

This document describes with forms and variants of REST APIs, sometimes using Openconfig or GraphQL for more standard ones, or a specific API for some others. They all share the same generic logic, requiring you to access a specific URL via HTTP to retrieve data, most likely, formatted in JSON. To do this, you may need to install some Python libraries.

First it is necessary to understand URLs.

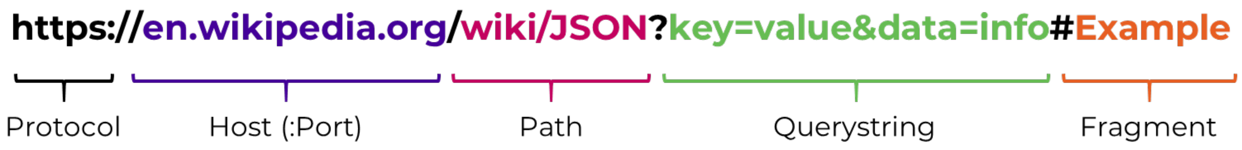
2.2.1 URLs

URL stands for Uniform Resource Locator, which is a string you enter in a browser, typically to access a site. This string contains a great deal of information.

HTTPS://en.wikipedia.org/wiki/JSON?key=value&data=info#Example

For example, this URL can be broken down into the following elements:

The protocol, which can be HTTPS, HTTP, ftp, etc.
The host, often an IP address, which is the location of the server you want to reach.
The host is sometimes followed by ":" and a value, which is the port number. If this is not present, then the browser defaults to the default protocol value (HTTP = 80, HTTPS = 443, for example).
The path at the destination server to reach the content. In the context of a REST API, this is often called an endpoint.
A query string follows an optional ?, and is used to pass arguments to the server, typically in name=value pairs. To pass several arguments, use the `&` character to separate them.
A fragment appears after an optional `#`, and leads directly to a given part of the content.


https://en.wikipedia.org/wiki/JSON?key=value&data=info#Example
Protocol
Host (:Port)
Path
Querystring
Fragment

2.2.2 HTTP Status Codes

When working with HTTP, it is important to understand the status code that is returned in a CALL. There are five status code categories:

1xx: informational response
2xx: successful
3xx: redirection
4xx: client error
5xx: server error

You should see a 200 when everything is operating normally (OK). Error codes such as 403 (forbidden access) or 404 (not found) help identify issues.

A complete list of the status codes is available in several locations, such as Wikipedia:

HTTPS://en.wikipedia.org/wiki/List_of_HTTP_status_codes

2.2.3 HTTP Request Methods

In REST, you send commands centered around a resource, which is anything that can be pointed to via HTTP protocol. CALL the API using standard HTTP request methods, such as: GET, POST, PUT, DELETE, and PATCH. There are more methods, but for REST APIs they are usually not required.

HTTPS://www.w3schools.com/tags/ref_httpmethods.asp

Note*: This is sometimes referred to as CRUD (Create, Read, Update, Delete), which is basically a cycle for database records.*

It is important to understand the role of the request methods:

Use **GET** to request data from a specified resource. You can think of it as a read.
Use **POST** to send data to a server to create or update a resource.
Use **PUT** to send data to a server to create or update a resource. This might seem redundant with POST, but there is a difference between them: PUT is idempotent. That is, if you are calling multiple times using the same PUT request, it will always produce the same result. POST will create multiples of the same resource.
Use **PATCH** to send data to a server to update an existing resource.
Use **DELETE** to delete a specified resource.

The API dictates the method you use. For example, if you use PUT instead of POST, you will receive an error from the server.

2.2.4 HTTP Headers

With HTTP, the data you transfer can be separated in headers and body. The body contains the usable data (the html to represent a web page, the data in JSON, etc.). The headers are very important because they provide crucial information about the body content.

Wen working with REST API, (and any API using HTTP), you will most likely need to manipulate the headers using the `content-type`, `accept`, `authorization` and `x-auth-token` commands.

***Note**: The HTTP Archive site is an excellent resource for learning more about HTTP. This site monitors the top 1.3M web sites and extracts the HTTP information for analyses. This information, along with reports such as State of the Web , are accessible to the public.*

2.2.4.1 Content-Type

The content-type indicates, as the name implies, what is the format of the data in the body. This is a very important piece of knowledge, as you would not treat that data the same way if this is pure text html, some binary form or some JSON data for an application.

In your context of a REST API, you'll most likely use the "application/json" value for this parameter, as long as you are, indeed, transmitting data in JSON format.

```
Content-Type: application/json
```

HTTPS://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type

2.2.4.2 Accept

The client uses the Accept header to advertise which content types it can understand. The server informs the client of the choice using the Content-Type header. In REST API, the accept header is often set to "application/json".

```
Accept: application/json
```

HTTPS://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept

2.2.4.3 Authorization

The authorization request header contains the credentials required to authenticate a user agent (a client) with a server. The authorization header value format is `<type> <credential>` with a space between them.

The typical Basic authentication type concatenates the username and the password in a single string, separated by a colon (:). This means that a username cannot also contain a colon. The result is encoded in base64. This is not an encryption because it is reversible.

To access an online tool that can encode and decode in base64, visit: HTTPS://www.base64encode.org/

As an example, the string `stef:extreme` is encoded in base64 as `c3RlZjpleHRyZW1l`.

```
authorization: Basic c3RlZjpleHRyZW1l
```

HTTPS://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization

2.2.4.4 X-Auth-Token

The X-Auth-Token is an unregistered header and is not subject to formal specification. Its presence and content are always tied to a respective application. It typically stores a token for authorization and can be considered as a shortcut of the bearer token defined in OAuth 2.0.

For Extreme APIs, the X-Auth-Token will be used with EXOS and VOSS Restconf implementation.

2.2.5 Manipulating Headers with Python

The requests module, and by extension the Urllib module, provide easy access to HTTP Headers. From requests, headers are simply a Python dictionary, and you can manipulate both the request and response headers.

```
import requests

r = requests.get("HTTPS://api.nasa.gov/planetary/apod")

print("Headers sent: ", r.request.headers)
print("\nHeaders received: ", r.headers)
```

The result is shown below:

```
C:\Extreme API with Python> headers-example.py
Headers sent:  {'User-Agent': 'python-requests/2.22.0', 'Accept-Encoding':
'gzip, deflate', 'Accept': '*/*', 'Connection': 'keep-alive'}
Headers received:  {'Server': 'openresty', 'Date': 'Sun, 14 Jun 2020 14:55:49
GMT', 'Content-Type': 'application/json', 'Transfer-Encoding': 'chunked',
'Connection': 'keep-alive', 'Vary': 'Accept-Encoding', 'Access-Control-Allow-
Origin': '*', 'X-Cache': 'MISS', 'Strict-Transport-Security': 'max-
age=31536000; preload', 'Content-Encoding': 'gzip'}
```

You can easily customize headers using the requests module:

```
import requests

headers = {
    'content-type': 'application/json',
    'x-auth-token': 'c3RlZjpleHRyZW1l'
}

r = requests.get("HTTPS://httpbin.org/get", headers=headers)

print("Headers sent: ", r.request.headers)
print("\nHeaders received: ", r.headers)
```

The result is shown below:

```
C:\Extreme API with Python> headers-example.py
Headers sent:  {'User-Agent': 'python-requests/2.22.0', 'Accept-Encoding':
'gzip, deflate', 'Accept': '*/*', 'Connection': 'keep-alive', 'content-type':
'application/json', 'x-auth-token': 'c3RlZjpleHRyZW1l'}

Headers received:  {'Date': 'Sun, 14 Jun 2020 15:03:53 GMT', 'Content-Type':
'application/json', 'Content-Length': '390', 'Connection': 'keep-alive',
'Server': 'gunicorn/19.9.0', 'Access-Control-Allow-Origin': '*', 'Access-
Control-Allow-Credentials': 'true'}
```

2.3 Authentication and Authorization

Most APIs will ask for some form of authentication or authorization before granting access to data. Some form of authentication will be necessary even when you are accessing public data.

Because HTTP is stateless, this is a key aspect of HTTP transport . Being stateless means a server, API, has no idea who is requesting/sending data for every transaction. So, if that data is not public, authentication/authorization information must be sent in every request. As you cannot expect to type your login/password for every CALL, there's a need to have an alternate way to manage that authentication process.

As a quick reminder, although authentication and authorization are related, they are different concepts. Authentication validates who you are, to give you access to what belongs to your profile, while authorization is more about what data you can access.

The most common authentication methods include Basic, Bearer, API key, and OAuth.

Several other methods exist, and some are standardized.

http://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml

These methods are described in the following sections.

2.3.1 Basic Authentication

Basic authentication is a classic and is well-defined with HTTP. These are the usual login and password credentials that you provide when identifying yourself to the API. These credentials are stored in the authorization portion of the HTTP headers. The credentials are not sent in plain text but are encoded in base64. Because this encoding mechanism is not an encryption, the best practice is to use it only with HTTPS.

2.3.2 Bearer Authentication

This method is also called token authentication and involves security tokens called bearer tokens. The name can be understood as "give access to the bearer of this token". The token is a cryptic string generated by the server in response to a login request, and is sent in the authorization headers. The bearer scheme was created as part of OAuth 2.0 (rfc 6750), but is sometimes used alone. As the token must remain secret, the best practice is to use it only with HTTPS.

2.3.3 API Key

API keys are common. This is what you would typically use when working with YouTube APIs, for example.

The benefit of this method is that it uses a different set of identification credentials than those used for the account itself, (for example, what basic authentication doesn't provide). The drawback with this method is that it is not standardized, and so the API determines how the key is passed. It could be hidden in the body, in the authorization header, in a cookie, or as a query string. Because the key must remain a secret, the best practice is to use it only with HTTPS.

2.3.4 OAuth 2.0

The Open Authorization protocol gives an API client limited access to user data. GitHub, Google, and Facebook APIs notably use it. This standard is defined in rfc 8252.

With OAuth 2.0, the authentication scenarios are called flows. Flows allow the resource owner to share the protected content from the resource server without sharing their credentials. For this purpose, access tokens (see bearer tokens) are issued by the server to client applications, giving them access the protected data.

Several flows are defined in the standard:

- Authorization code
- Implicit
- Resource owner password credentials
- Client credentials

Learn more about how to use flows on the getting started official site:

HTTPS://oauth.net/getting-started/

2.3.5 Managing Passwords or Tokens with Python

When you are writing applications that need to access APIs, the best practice is to not store credentials (hard-code them) in the code. Although this approach is convenient for testing purposes, it presents an obvious security risk.

One way to handle this situation is to ask for credentials when executing the application, using at a minimum a library (such as getpass) to hide the password. This approach is simple, but it requires someone in front of the application to enter the information.

```
import getpass

username = input("Username: ")
```

```
password = getpass.getpass()

print("\nYou entered:\nUsername: {} Password: {}".format(username, password))
```

The result is shown below:

```
C:\Extreme API with Python> auth-examples.py
Username: Stef
Password:

You entered:
Username: Stef Password: extreme
```

If the application is running on a secure environment, another way to is to store passwords, keys, and tokens as environment variables that the application can access. How you create environment variables will depend on your operating system:

- In Windows, create environment variables from **Control Panel > System > Advanced System Settings > Environment Variables.**
- With MacOS and Linux, add variables in the .batch_profile file, located in your home directory. The syntax is export <Var Name>=”<Value>”. There are no spaces between the variable name, the = sign and the value.

You can then access these environment variables with the OS module.

In this example, you have created (on Windows 10) 2 environment variables:

- MY_USER
- MY_PASSWORD

You can retrieve them from Python, without exposing them in the code:

```
import os

username = os.environ.get('MY_USER')
password = os.environ.get('MY_PASSWORD')

print("Username is: {}\nPassword is: {}".format(username, password))
```

The result:

```
C:\Extreme API with Python> auth-examples.py
Username is: Stef
Password is: extreme
```

2.4 Understanding JSON

JSON (JavaScript Object Notation) is an open standard file format, defined in RFC 8259 (HTTPS://tools.ietf.org/html/rfc8259), widely used to format the data transmitted and stored with modern APIs and tools. Despite its name, JSON is language-independent and is used with whatever programming language needed.

It is the preferred data format when working with REST APIs, but also is widely used as config files for many applications (from your child's Minecraft server settings to XMC settings, for example). It's worth to note that the first version of the JSON RFC (rfc 4627) registered the media type "application/json".

JSON is a text format, human-readable tool that has been widely adopted to replace other standards such as XML. The JSON format accepts basic data types such as number, string (delimited with double-quotation marks), Boolean (true or false), array (ordered list of any type delimited with square brackets, and each element separated with a comma), objects (collection of key-value pairs where the key is a string) and empty value with the word null. A JSON object is always delimited between curly brackets or square brackets. Each entry is separated with a comma, except for the last one for each object.

The following example was taken from Wikipedia (HTTPS://en.wikipedia.org/wiki/JSON#Example):

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Comments are not allowed in a JSON document.

Although JSON is not the only format for APIs, it is currently the most widely used. It comes with some limitations that open doors for new formats. However, when you need a text file format with information that can be read by humans, JSON is currently the most efficient tool. Nevertheless, when you need to exchange a vast amount of information, performance and efficiency become more important and you can consider new standards. These alternatives are not yet used in external Extreme APIs, although binary formats, such as Protobuf, are becoming popular as well. JSON and Protobuf are expected to co-exist, serving different needs.

2.5 Manipulating JSON with Python

Manipulating JSON-formatted data with Python is simple because the data types are well matched. The standard library, included by default with Python, has a JSON module that converts JSON to Python, and vice-versa, as defined in the Python documentation, and shown in this table:

JSON	Python
Object	dict
Array	list
String	str
number (int)	int
number (real)	float
true	True
false	False
null	None

HTTPS://docs.python.org/3/library/json.html#encoders-and-decoders

View the functions and methods available with JSON using the `print(dir(json))` command:

```
import json

print(dir(json))
```

The result:

```
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__', '__author__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__', '__version__',
'_default_decoder', '_default_encoder', 'codecs', 'decoder',
'detect_encoding', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']
```

The methods most often used are highlighted in this example. The following examples illustrate how to use some of these methods.

```
import json

json_sample = '''
{
  "whisky": [
      {
        "name": "Hibiki",
        "type": "Blended",
        "age": 17
      },
      {
        "name": "Old Pulteney",
        "type": "Single Malt",
        "age": 21
      }
  ],
  "stock": null,
  "alcohol": true
}
'''

data = json.loads(json_sample)
print(type(data))
print(data)

new_data = json.dumps(data)
print(type(new_data))

print(new_data)
```

This example imports the JSON module and manipulates a JSON entry in Python. You must first transform it to an editable dictionary, then reconvert it to JSON format. The null and Boolean values change accordingly.

```
<class 'dict'>
{'whisky': [{'name': 'Hibiki', 'type': 'Blended', 'age': 17}, {'name': 'Old
Pulteney', 'type': 'Single Malt', 'age': 21}], 'stock': None, 'alcohol':
True}
<class 'str'>
```

```
{"whisky": [{"name": "Hibiki", "type": "Blended", "age": 17}, {"name": "Old
Pulteney", "type": "Single Malt", "age": 21}], "stock": null, "alcohol":
true}
```

In this example, a string is the source, but you could also have uploaded information from a file, and saved it back to a file using the `json.load()` and `json.dump()` commands.

2.6 Interact with a REST API using Python

Now that you have a basic understanding of what a REST API is, the following examples show how you to interact with one using Python.

2.6.1 Urllib

When you are working with Python, you can access HTTP or HTTPS URLs using the standard (included) Urllib package. For details about how to use Urllib, see the official documentation, or use any of the many tutorials available online.

HTTPS://docs.python.org/3/library/urllib.html

Urllib has several modules, the request module being the most useful.

HTTPS://docs.python.org/3/library/urllib.request.html#module-urllib.request

2.6.1.1 Urllib examples

Enter `dir()` of the `urllib.request` to see the list of available methods and functions.

```
from urllib import request

print(dir(request))
```

The output is shown below, with the most useful function highlighted.

```
['AbstractBasicAuthHandler', 'AbstractDigestAuthHandler',
'AbstractHTTPHandler', 'BaseHandler', 'CacheFTPHandler',
'ContentTooShortError', 'DataHandler', 'FTPHandler', 'FancyURLopener',
'FileHandler', 'HTTPBasicAuthHandler', 'HTTPCookieProcessor',
'HTTPDefaultErrorHandler', 'HTTPDigestAuthHandler', 'HTTPError',
'HTTPErrorProcessor', 'HTTPHandler', 'HTTPPasswordMgr',
'HTTPPasswordMgrWithDefaultRealm', 'HTTPPasswordMgrWithPriorAuth',
'HTTPRedirectHandler', 'HTTPSHandler', 'MAXFTPCACHE', 'OpenerDirector',
'ProxyBasicAuthHandler', 'ProxyDigestAuthHandler', 'ProxyHandler', 'Request',
'URLError', 'URLopener', 'UnknownHandler', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', '__version__', '_cut_port_re', '_ftperrors', '_have_ssl',
'_localhost', '_noheaders', '_opener', '_parse_proxy',
'_proxy_bypass_macosx_sysconf', '_randombytes', '_safe_gethostbyname',
'_thishost', '_url_tempfiles', 'addclosehook', 'addinfourl', 'base64',
'bisect', 'build_opener', 'contextlib', 'email', 'ftpcache', 'ftperrors',
'ftpwrapper', 'getproxies', 'getproxies_environment', 'getproxies_registry',
'hashlib', 'http', 'install_opener', 'io', 'localhost', 'noheaders', 'os',
'parse_http_list', 'parse_keqv_list', 'pathname2url', 'posixpath',
'proxy_bypass', 'proxy_bypass_environment', 'proxy_bypass_registry', 'quote',
're', 'request_host', 'socket', 'splitattr', 'splithost', 'splitpasswd',
'splitport', 'splitquery', 'splittag', 'splittype', 'splituser',
'splitvalue', 'ssl', 'string', 'sys', 'tempfile', 'thishost',
'time', 'to_bytes', 'unquote', 'unquote_to_bytes', 'unwrap', 'url2pathname',
'urlcleanup', 'urljoin', 'urlopen', 'urlparse', 'urlretrieve', 'urlsplit',
'urlunparse', 'warnings']
```

The following example shows how to use urlopen, and how to find other functions:

```
from urllib import request

resp = request.urlopen('HTTPS://www.youtube.com')
print(dir(resp))
```

Perform a `dir()` of the object returned from `request.urlopen` to see more functions.

```
['__abstractmethods__', '__class__', '__del__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_abc_impl',
'_checkClosed', '_checkReadable', '_checkSeekable', '_checkWritable',
'_check_close', '_close_conn', '_get_chunk_left', '_method', '_peek_chunked',
'_read1_chunked', '_read_and_discard_trailer', '_read_next_chunk_size',
'_read_status', '_readall_chunked', '_readinto_chunked', '_safe_read',
'_safe_readinto', 'begin', 'chunk_left', 'chunked', 'close', 'closed',
'code', 'debuglevel', 'detach', 'fileno', 'flush', 'fp', 'getcode',
'getheader', 'getheaders', 'geturl', 'headers', 'info', 'isatty', 'isclosed',
'length', 'msg', 'peek', 'read', 'read1', 'readable', 'readinto',
'readinto1', 'readline', 'readlines', 'reason', 'seek', 'seekable', 'status',
'tell', 'truncate', 'url', 'version', 'will_close', 'writable', 'write',
'writelines']
```

The following example shows how to use these functions:

```
from urllib import request

resp = request.urlopen('HTTPS://www.python.org')

print(resp.code)
print(resp.length)

data = resp.read()
```

```
print(type(data))
print(len(data))
```

The result is shown below:

```
200
48959
<class 'bytes'>
48959
```

This example shows the success HTTP status code (200), and the amount of data returned in bytes.

2.6.2 Requests

Although Urllib provides all the required tools to manipulate URLs and HTTP CALLs, a better package called Requests is commonly used.

HTTPS://requests.readthedocs.io/en/master/

The best practice is to install Requests with PIP.

***Note**: The Requests module is part of XMC Python scripting engine and EXOS Python scripting capability.*

The following example creates a virtual environment in Windows 10 to demonstrate how to create and activate a venv.

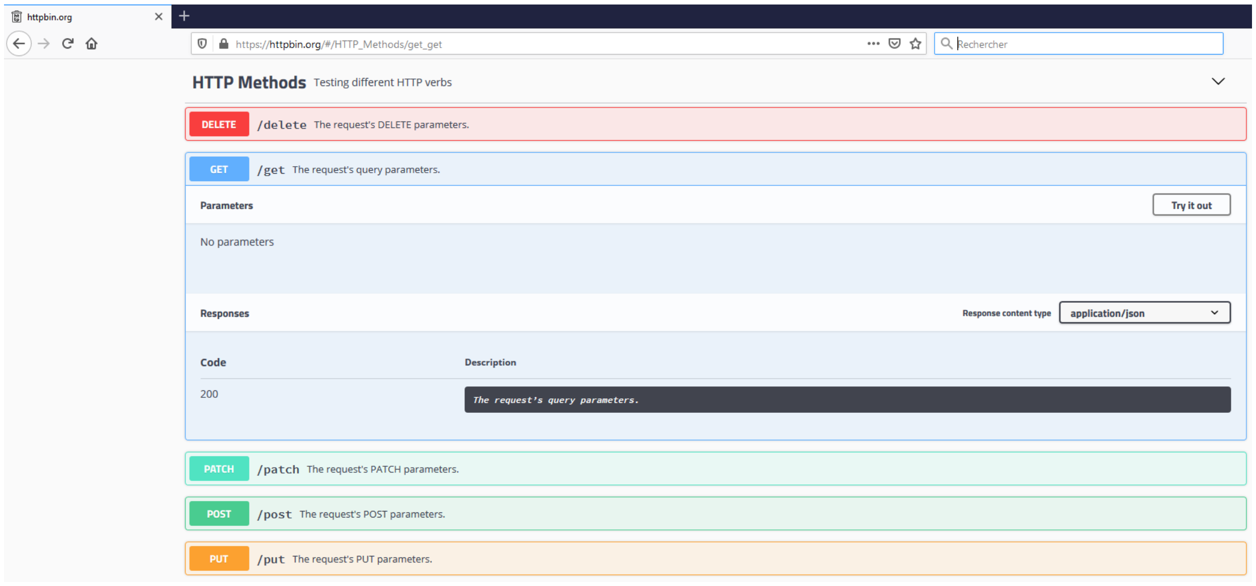
```
C:\> python -m venv "c:\Extreme API with Python"

C:\> cd "\Extreme API with Python"

C:\Extreme API with Python> dir

Directory of C:\Extreme API with Python

05-Jun-20  09:19    <DIR>          .
05-Jun-20  09:19    <DIR>          ..
03-Jun-20  16:00    <DIR>          Extreme API with Python
05-Jun-20  01:47           492,137 Extreme API.docx
03-Jun-20  11:55    <DIR>          Include
04-Jun-20  00:59               453 json-example.py
03-Jun-20  11:55    <DIR>          Lib
02-Jun-20  19:43         5,537,071 Presentation1.pptx
03-Jun-20  11:55               125 pyvenv.cfg
04-Jun-20  13:47               167 requests-example.py
03-Jun-20  12:03    <DIR>          Scripts
03-Jun-20  01:11               177 urllib-example.py
               6 File(s)      6,030,130 bytes
               6 Dir(s)  691,064,119,296 bytes free

C:\Extreme API with Python>
C:\Extreme API with Python> Scripts\activate.bat

(Extreme API with Python) C:\Extreme API with Python> pip install requests
Collecting requests
```

```
  Downloading
HTTPS://files.pythonhosted.org/packages/1a/70/1935c770cb3be6e3a8b78ced23d7e0f3b187f5cb
fab4749523ed65d7c9b1/requests-2.23.0-py2.py3-none-any.whl (58kB)
     |████████████████████████████████| 61kB 1.9MB/s
Collecting certifi>=2017.4.17 (from requests)
  Downloading
HTTPS://files.pythonhosted.org/packages/57/2b/26e37a4b034800c960a00c4e1b3d9ca5d7014e98
3e6e729e33ea2f36426c/certifi-2020.4.5.1-py2.py3-none-any.whl (157kB)
     |████████████████████████████████| 163kB 6.4MB/s
Collecting idna<3,>=2.5 (from requests)
  Downloading
HTTPS://files.pythonhosted.org/packages/89/e3/afebe61c546d18fb1709a61bee788254b40e736c
ff7271c7de5de2dc4128/idna-2.9-py2.py3-none-any.whl (58kB)
     |████████████████████████████████| 61kB 4.1MB/s
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 (from requests)
  Downloading
HTTPS://files.pythonhosted.org/packages/e1/e5/df302e8017440f111c11cc41a6b432838672f5a7
0aa29227bf58149dc72f/urllib3-1.25.9-py2.py3-none-any.whl (126kB)
     |████████████████████████████████| 133kB 3.3MB/s
Collecting chardet<4,>=3.0.2 (from requests)
  Using cached
HTTPS://files.pythonhosted.org/packages/bc/a9/01ffebfb562e4274b6487b4bb1ddec7ca55ec751
0b22e4c51f14098443b8/chardet-3.0.4-py2.py3-none-any.whl
Installing collected packages: certifi, idna, urllib3, chardet, requests
Successfully installed certifi-2020.4.5.1 chardet-3.0.4 idna-2.9 requests-2.23.0
urllib3-1.25.9

 (Extreme API with Python) C:\Extreme API with Python>
```

To better illustrate its use, you can create some GET and POST examples. A good resource for making HTTP CALLs is HTTPS://httpbin.org which is a simple HTTP request and response service. These can be considered the first REST API CALLs.

***Note**: The httpbin.org service has been written by the same author than the Requests module.*

Examine the GET method. The response content type is set to *application/json* by default. Keep this setting so that you can manipulate JSON data.



HTTP Methods Testing different HTTP verbs

DELETE `/delete` The request's DELETE parameters.

GET `/get` The request's query parameters.

Parameters

No parameters

Responses

Response content type: application/json

Code	Description
200	The request's query parameters.

PATCH `/patch` The request's PATCH parameters.

POST `/post` The request's POST parameters.

PUT `/put` The request's PUT parameters.

Make a REST CALL, using GET to retrieve data. This service lets you add parameters (arguments) to the URL in a query string, so that the server also returns this information .

Requests has an integrated JSON function that you can use also, as shown below :

```python
import requests

qstring = {"h2g2": 42, "elite": 1337}
r = requests.get('HTTPS://httpbin.org/get', params=qstring)

print(r.url)
print(r.status_code)
print(r.headers['content-type'])
print(r.encoding)
print(r.text)

data = r.json()
print(type(data))
print(data)
```

In this example, the query string has been separated from the URL. You could have added the parameters directly to the URL, but it is a good practice to work this way, which allows you to reuse the same URL with different parameters. When you test using this method, you use the GET function with requests and the GET path on httpbin.org.

The result of a test is the shown below:

```
https://httpbin.org/get?h2g2=42&elite=1337
200
application/json
None
{
  "args": {
    "elite": "1337",
    "h2g2": "42"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.22.0",
    "X-Amzn-Trace-Id": "Root=1-5ed8c610-484d6854daf7112485a3b020"
  },
  "origin": "109.13.132.180",
  "url": "https://httpbin.org/get?h2g2=42&elite=1337"
}

<class 'dict'>
{'args': {'elite': '1337', 'h2g2': '42'}, 'headers': {'Accept': '*/*',
'Accept-Encoding': 'gzip, deflate', 'Host': 'httpbin.org', 'User-Agent':
'python-requests/2.22.0', 'X-Amzn-Trace-Id': 'Root=1-5ed8c610-
484d6854daf7112485a3b020'}, 'origin': '109.13.132.180', 'url':
'https://httpbin.org/get?h2g2=42&elite=1337'}
```

Next, send data to the service by using the POST method from requests and changing the path to the service to POST. Because you are now sending data to the service, you must remove the params keyword and replace it with the data keyword, as shown below:

```python
import requests

payload = {"h2g2": 42, "elite": 1337}
r = requests.post('HTTPS://httpbin.org/post', data=payload)

print(r.url)
print(r.status_code)
print(r.headers['content-type'])
print(r.text)

data = r.json()
```

```
print(type(data))
print(data)
```

In the results, you can see that the URL no longer contains a query string, and in the JSON returned, there is a form entry with the data you sent.

```
HTTPS://httpbin.org/post
200
application/json
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "elite": "1337",
    "h2g2": "42"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "17",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.22.0",
    "X-Amzn-Trace-Id": "Root=1-5ed8cb37-4c412bacebb72bbcaf3e5bfc"
  },
  "json": null,
  "origin": "109.13.132.180",
  "url": "HTTPS://httpbin.org/post"
}

<class 'dict'>
{'args': {}, 'data': '', 'files': {}, 'form': {'elite': '1337', 'h2g2':
'42'}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate',
'Content-Length': '17', 'Content-Type': 'application/x-www-form-urlencoded',
'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.22.0', 'X-Amzn-Trace-
Id': 'Root=1-5ed8cb37-4c412bacebb72bbcaf3e5bfc'}, 'json': None, 'origin':
'109.13.132.180', 'url': 'HTTPS://httpbin.org/post'}
```

Another option when using the requests.get function is the timeout parameter. Without this parameter, the requests module waits indefinitely for an answer, which can be a problem if you have made a mistake. This can also result in very slow server speeds and you don't want the application to spend too much time waiting. You can set a limit before raising an error. Httpbin can help you to simulate this, with the delay service in the dynamic data menu. To call it, add /delay/<value in seconds> to the URL.

For example:

```
import requests
```

```
r = requests.get('HTTPS://httpbin.org/delay/4', timeout=3)
```

Add a delay of 4 seconds for the answer with a timeout of 3 seconds. Running the script gives you a traceback:

```
Traceback (most recent call last):
[…]
    raise ReadTimeout(e, request=request)
requests.exceptions.ReadTimeout: HTTPSConnectionPool(host='httpbin.org',
port=443): Read timed out. (read timeout=3)
```

You can also use a Python try/except, which provides a cleaner result without breaking code.

```
import requests

try:
    r = requests.get('HTTPS://httpbin.org/delay/4', timeout=3)
except requests.exceptions.Timeout:
    print("Server is too long to answer")
```

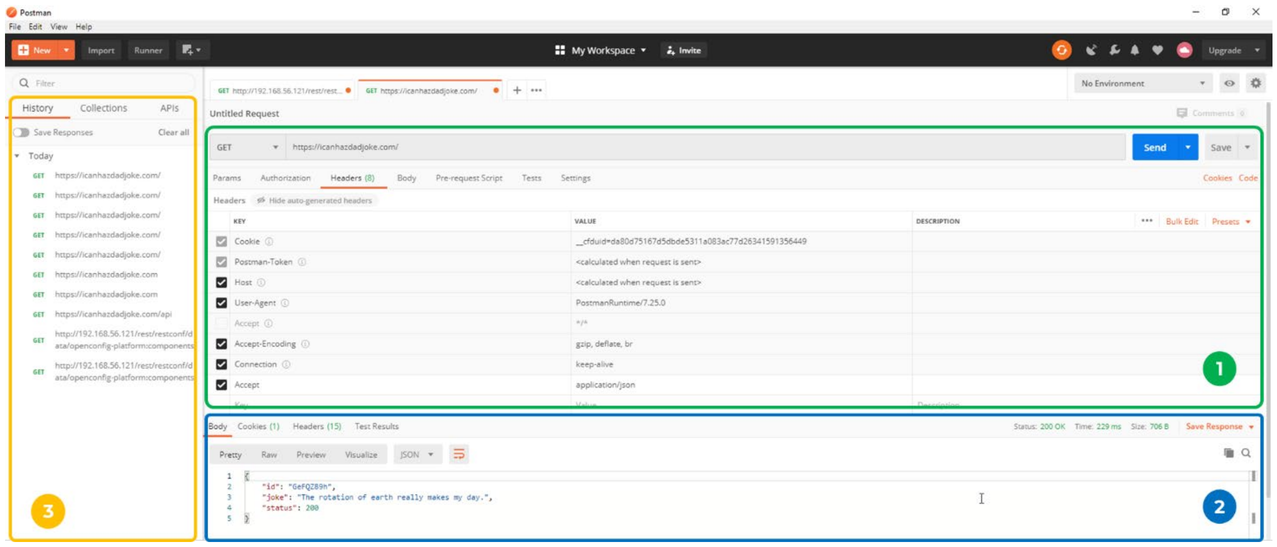
2.6.3 Testing a REST API

Now that you are familiar with REST and Python, another useful tool when working with a REST API is Postman.

***Note**: You can add Postman to some browsers via plug-in or it can be run as an external application on most systems.*

When you work with an API, you must know exactly what URL to use, and which data format to send or receive. Having the ability to quickly test a CALL and interpret the results without having to write the code for it is extremely useful. This is where Postman can help.

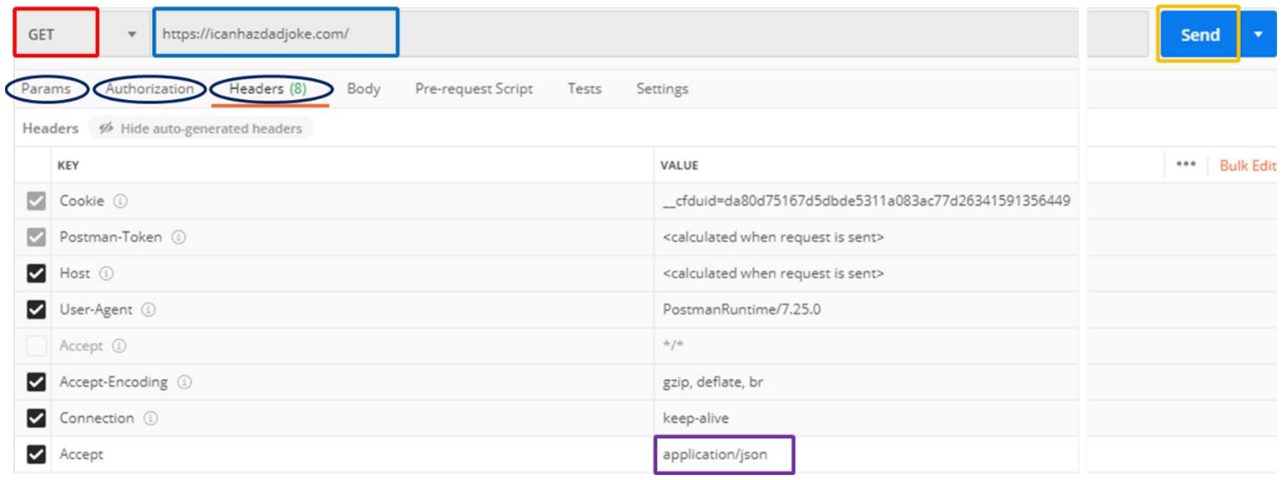
The Postman GUI can be broken down into three main sections, as shown below:



GET https://icanhazdadjoke.com/

```
{
    "id": "GeFQZ89h",
    "joke": "The rotation of earth really makes my day.",
    "status": 200
}
```

These sections are the Request builder, the Response window, and the Explorer window. In the Request builder, you create the HTTP CALL, specify the URL, add parameters, and set the authentication and headers. The Request and Response windows display requests and responses.

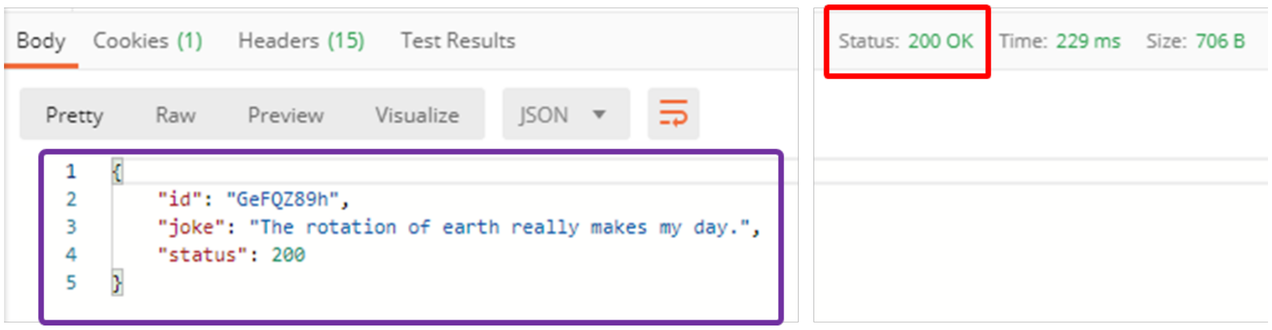


GET https://icanhazdadjoke.com/

	KEY	VALUE
✓	Cookie	__cfduid=da80d75167d5dbde5311a083ac77d26341591356449
✓	Postman-Token	<calculated when request is sent>
✓	Host	<calculated when request is sent>
✓	User-Agent	PostmanRuntime/7.25.0
	Accept	*/*
✓	Accept-Encoding	gzip, deflate, br
✓	Connection	keep-alive
✓	Accept	application/json

Zoom into the Request builder, to see (in this example) an HTTP GET method, the URL for the API next to it, and several tabs where you can personalize the CALL. In this example, no authorization is necessary (this is very rare) so you just must set JSON as the application.

Select **Send** to see the response from the API.



```
{
    "id": "GeFQZ89h",
    "joke": "The rotation of earth really makes my day.",
    "status": 200
}
```

In the response window, you can see the HTTP Status code, in this case an encouraging 200, and the data sent back in JSON by the API. You can now use this information in the application, as you can see the keys and value types returned.

You can now write a Python application to interface with this API.

```python
import requests

headers = {"Accept": "application/json"}
try:
    r = requests.get('HTTPS://icanhazdadjoke.com/', headers=headers, timeout=3)
except requests.exceptions.Timeout:
    print("Service is currently unavailable, please try again later")
    exit(0)

if r.ok:
    joke = r.json()
    print(joke["joke"])
else:
    print("No joke today!")
```

You can now access the joke for the day:

```
I am so good at sleeping I can do it with my eyes closed!
```

2.7 Webhooks

When dealing with APIs, it is sometimes more practical to rely on webhook services than making REST CALLs. Webhooks are sometimes referred to as reverse API, as they push data automatically from a service to an application, instead of having the application request data. This approach can be more elegant when you want to update data as it changes, and this can also be a better way to interact with an official API, as it can potentially limit your number of CALLs per day.

There are several sites to help you test webhooks, such as the Webhook.site.

2.8 HTTPS with Python

Usually, when you work with an API that uses HTTP as the transfer protocol, such as REST API, you will be required to use HTTPS for obvious security reasons. If you are using self-signed certificates, you will see warnings and errors. To avoid this, in the code, add the disable_warnings method from Urllib3 and add the `verify=False` argument with requests.

```
import requests
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

r = requests.get(url, verify=False, params=payload, headers=getHeaders)
```

3 EXOS APIs

EXOS offers several APIs for developers that support on-switch automation and external automation. The following sections describe these options.

3.1 On-Switch APIs

EXOS has included automation and scripting since its original inception. The first scripting interface offered to the users, and still present today, is called CLI Scripting and uses TCL functions for advanced scripting. You'll not cover this scripting interface in this guide, but rather focus on the more modern Python capabilities.

3.1.1 Python Scripting

Starting with the release of EXOS 15.6, Python has been added to the EXOS scripting toolbox. The Python version used is 2.7, and more precisely it was 2.7.3 before being upgraded with EXOS 30.1 to version 2.7.15.

The standard library is available, along with extra modules such as Argparse and Requests.

To see the full list of available modules, create a simple one-line script and execute it on a switch:

```
help('modules')
```

3.1.1.1 Create a Python Script

There are several options for creating a script on an EXOS switch. Either you create it directly on it, from the CLI, or you do it on your computer and send it to switch when completed.

From the CLI, you can access a light version of vi using either the command "vi" or "edit". When you create a new file with this command, it must have a supported file extension by EXOS. The supported file extensions are .pol, .pkt, .xsf, .py and .xml. Obviously, when writing a Python script, you must use the official .py file extension.

Note*: when you create a file, it is located by default into `/config`, which is an alias for `/usr/local/cfg`.*

Some text editors and IDE can edit a remote file using SSH, SCP or similar transfer protocols. This capability can be built-in or added using a plug-in. It may be easier to manipulate files this way.

3.1.1.2 Copying Python Scripts to a Switch

If you are not working directly from the switch CLI, you can copy the files to the switch using TFTP or SCP. For example, assuming you have a TFTP server running on your PC (such as tftpd64) and pointing to the correct directory, you can copy the files from the CLI this way:

```
sw1.2 # tftp get 192.168.56.1 vr VR-Mgmt MyScript.py
```

3.1.1.3 Execute a Python Script

There are two CLI commands you can use to execute an existing Python script from the CLI:

- `run script <script_name>.py`
- `load script <script_name>.py`

The first command was introduced specifically for executing Python script. The second one is the legacy command, used for CLI Scripting.

3.1.1.4 EXOS CLI Module

You can use an EXOS module called exsh to execute a Python script on EXOS. This module allows you to execute any CLI command and returns the output either as a string, XML, both, or none.

```
exsh.clicmd(cmd, capture=False, xml=False, args=None)
```

Parameters:

- **cmd**: a string containing any valid EXOS CLI command.
- **capture**: a Boolean, defaulting to False if not specified, returning as a text (string) the CLI output of the command.
- **xml**: a Boolean, defaulting to False if not specified, returning the XML that EXOS used to create the CLI output
- **args**: a string to provide additional input to some EXOS commands that prompt for more information

Returns:

- **None**: if both capture and xml are False
- **Captured text**: if capture is True
- **XML**: if xml is True
- **Captured text and xml**: if both capture and xml are True

Raises:

- **RuntimeError**: EXOS command is invalid or encountered an error

When you work with JSON data, you can be tempted to use the embedded `cli2json.py` script. Calling a script from another script is not supported, as each script has its own session.

This is an example of a simple script:

```
import exsh

for vid in range(10, 15):
    exsh.clicmd("create vlan {}".format(vid))
```

The result:

```
sw1.2 # run script createVlans.py
sw1.3 # show vlan
Untagged ports auto-move: Inform
-----------------------------------------------------------------------------------------------
Name            VID  Protocol Addr       Flags                         Proto  Ports  Virtual
                                                                              Active router
                                                                              /Total
-----------------------------------------------------------------------------------------------
Default         1    --------------------------------T-------------    ANY    5 /5   VR-Default
Mgmt            4095 192.168.56.121 /24  --------------------------    ANY    1 /1   VR-Mgmt
VLAN_0010       10   ----------------------------------------------    ANY    0 /0   VR-Default
VLAN_0011       11   ----------------------------------------------    ANY    0 /0   VR-Default
VLAN_0012       12   ----------------------------------------------    ANY    0 /0   VR-Default
VLAN_0013       13   ----------------------------------------------    ANY    0 /0   VR-Default
VLAN_0014       14   ----------------------------------------------    ANY    0 /0   VR-Default
-----------------------------------------------------------------------------------------------
Flags : (B) BFD Enabled, (c) 802.1ad customer VLAN, (C) EAPS Control VLAN,
        (d) Dynamically created VLAN, (D) VLAN Admin Disabled,
        (E) ESRP Enabled, (f) IP Forwarding Enabled,
        (F) Learning Disabled, (i) ISIS Enabled,
        (I) Inter-Switch Connection VLAN for MLAG, (k) PTP Configured,
        (l) MPLS Enabled, (L) Loopback Enabled, (m) IPmc Forwarding Enabled,
        (M) Translation Member VLAN or Subscriber VLAN, (n) IP Multinetting Enabled,
        (N) Network Login VLAN, (o) OSPF Enabled, (O) Virtual Network Overlay,
        (p) PIM Enabled, (P) EAPS protected VLAN, (r) RIP Enabled,
        (R) Sub-VLAN IP Range Configured, (s) Sub-VLAN, (S) Super-VLAN,
        (t) Translation VLAN or Network VLAN, (T) Member of STP Domain,
        (v) VRRP Enabled, (V) VPLS Enabled, (W) VPWS Enabled,
        (Y) Policy Enabled

Total number of VLAN(s) : 7
```

3.1.1.5 Automate the Python Script Execution

EXOS offers the ability to dynamically execute scripts when a particular event is met using a feature called UPM.

3.1.1.5.1 UPM

UPM can trigger a script based on time of the day (for example every second, or twice a day at a fixed time or on a given date), LLDP events, or based on events in the log. This capability combined with Python scripting allows for very powerful on-switch automation. UPM can pass event-related parameters to the script; for example, a port number associated to a monitored event, or a MAC address, etc.

As a basic example, when a port goes up or down in the logs, you can ask UPM to trigger a basic Python script to create a VLAN and add this port to it, or delete this VLAN and add the port back to the default VLAN. Obviously, this example is too basic for a real use-case, but it shows the concepts involved. The Python script would look like this:

```
import exsh
import sys

if len(sys.argv) < 3:
    print "Missing arguments\nExpected arguments are Port and Action\nValid Actio
ns are down and up"
    exit(0)

if sys.argv[2] == "down":
    exsh.clicmd("delete vlan 42")
    exsh.clicmd("config vlan Default add port {}".format(sys.argv[1]))
else:
    exsh.clicmd("create vlan 42")
    exsh.clicmd("config vlan 42 add port {}".format(sys.argv[1]))
```

UPM config requires that you create a profile and a log filter to monitor the event you want associated with this profile.

```
create upm profile Up_Down_Profile

enable cli scripting

IF (!$MATCH($EVENT.LOG_COMPONENT_SUBCOMPONENT,vlan.msgs) &&
!$MATCH($EVENT.LOG_EVENT,portLinkStateDown)) THEN
    run script upm_port.py $EVENT.LOG_PARAM_0 down
ENDIF

IF (!$MATCH($EVENT.LOG_COMPONENT_SUBCOMPONENT,vlan.msgs) &&
!$MATCH($EVENT.LOG_EVENT,portLinkStateUp)) THEN
    run script upm_port.py $EVENT.LOG_PARAM_0 up
ENDIF
.

create log filter Port_Up_Down

config log filter Port_Up_Down add event vlan.msgs.portLinkStateUp
config log filter Port_Up_Down add event vlan.msgs.portLinkStateDown

create log target upm Up_Down_Profile
enable log target upm Up_Down_Profile
config log target upm Up_Down_Profile filter Port_Up_Down severity Info
```

You can validate the correct execution on the switch as the event happened:

```
sw1.29 # sh log
06/07/2020 11:03:51.39 <Noti:UPM.Msg.upmMsgExshLaunch> Launched profile
Up_Down_Profile for the event log-message
06/07/2020 11:03:51.38 <Info:vlan.msgs.portLinkStateUp> Port 1 link UP at speed 100
Mbps and full-duplex
06/07/2020 11:03:44.42 <Noti:UPM.Msg.upmMsgExshLaunch> Launched profile
Up_Down_Profile for the event log-message
06/07/2020 11:03:44.42 <Info:vlan.msgs.portLinkStateDown> Port 1 link down
sw1.30 #
```

```
sw1.30 # sh upm history
--------------------------------------------------------------------------------
Exec  Event/                Profile         Port   Status  Time Launched
Id    Timer/ Log filter
--------------------------------------------------------------------------------
2     Log-Message(Port_Up_ Up_Down_Profile  --- Pass    2020-06-07 11:03:51
1     Log-Message(Port_Up_ Up_Down_Profile  --- Pass    2020-06-07 11:03:44
--------------------------------------------------------------------------------

Number of UPM Events in Queue for execution: 0
sw1.31 #
sw1.31 # sh upm history exec-id 2

UPM Profile: Up_Down_Profile
Event:  Log-Message(Port_Up_Down)
Profile Execution start time: 2020-06-07 11:03:51
Profile Execution Finish time: 2020-06-07 11:03:51
Execution Identifier: 2 Execution Status: Pass

Execution Information:
3 # enable cli scripting
4 # configure cli mode non-persistent
5 # set var EVENT.NAME LOG_MESSAGE
6 # set var EVENT.LOG_FILTER_NAME "Port_Up_Down"
7 # set var EVENT.LOG_DATE "06/07/2020"
8 # set var EVENT.LOG_TIME "11:03:51.38"
9 # set var EVENT.LOG_COMPONENT_SUBCOMPONENT "vlan.msgs"
10 # set var EVENT.LOG_EVENT "portLinkStateUp"
11 # set var EVENT.LOG_SEVERITY "Info"
12 # set var EVENT.LOG_MESSAGE "Port %0% link UP at speed %1% and %2%"
13 # set var EVENT.LOG_PARAM_0 "1"
14 # set var EVENT.LOG_PARAM_1 "100 Mbps"
15 # set var EVENT.LOG_PARAM_2 "full-duplex"
16 # set var EVENT.LOG_PARAM_3 "1"
17 # set var EVENT.PROFILE Up_Down_Profile
19 # enable cli scripting
21 # IF (!$MATCH($EVENT.LOG_COMPONENT_SUBCOMPONENT,vlan.msgs) &&
!$MATCH($EVENT.LOG_EVENT,portLinkStateDown)) THEN
22 #     run script upm_port.py $EVENT.LOG_PARAM_0 down
23 # ENDIF
25 # IF (!$MATCH($EVENT.LOG_COMPONENT_SUBCOMPONENT,vlan.msgs) &&
!$MATCH($EVENT.LOG_EVENT,portLinkStateUp)) THEN
26 #     run script upm_port.py $EVENT.LOG_PARAM_0 up
27 # ENDIF

Number of UPM Events in Queue for execution: 0
```

The VLAN is created:

```
sw1.29 # sh vlan
Untagged ports auto-move: Inform
------------------------------------------------------------------------------------
Name            VID  Protocol Addr       Flags                        Proto  Ports  Virtual
                                                                             Active router
                                                                             /Total
-------------------------------------------------------------------------------------
```

```
Default          1    --------------------------------T-------------    ANY    0 /0   VR-Default
Mgmt             4095 192.168.56.121 /24  --------------------------    ANY    1 /1   VR-Mgmt
VLAN_0042        42   ----------------------------------------------    ANY    1 /1   VR-Default
------------------------------------------------------------------------------------
Flags : (B) BFD Enabled, (c) 802.1ad customer VLAN, (C) EAPS Control VLAN,
[…]
Total number of VLAN(s) :
```

3.1.1.5.2 Startup Files

Potential companions for Python Scripting and UPM are the EXOS startup files. Historically, two startup files can be used with EXOS:

- default.xsf
- autoexec.xsf

The autoexec.xsf file starts at every boot of the switch, while the default.xsf is only executed when the switch boots with no configuration (in factory default config or after an unconfigure switch all command). The default.xsf has a higher precedence. An autoexec.xsf cannot be used if default.xsf has been started.

Both startup files execute valid CLI commands, which must be executed within 500 seconds. The startup file aborts after 500 seconds without executing the remaining commands.

***Note**: The results of the startup file execution can be seen using the command* show script output {default | autoexec}.

With the introduction of Python support in EXOS, these two files have been added to Python, and with EXOS 21.1 the .py versions are also supported.

default.py
autoexec.py

EXOS 22.3 introduced a new startup file in EXOS named exshrc.xsf. This file is executed after a successful login in EXOS, and lets you execute specific CLI commands, or scripts, at each login. You can see who is connected and start a required script per user. This can be helpful when creating a menu for specific operators, for example.

In the following example, using the CLI Scripting built-in variables, and, specifically, $CLI.USER, returns the user of the current session. There are many built-in variables available. Refer the EXOS User Guide, in the CLI Scripting chapter, for more information.

```
sw1.1 # vi exshrc.xsf
enable cli scripting
IF (!$MATCH($CLI.USER,admin)) THEN
  create log message "User Admin just connected!"
ENDIF
disable cli scripting
sw1.2 #
```

If you disconnect from the switch then reconnect as admin, you see the following:

```
sw1.1 # sh log
07/01/2020 00:51:44.26 <Info:System.userComment> User Admin just connected!
07/01/2020 00:51:44.26 <Info:AAA.authPass> Login passed for user admin
through telnet (192.168.56.1)
07/01/2020 00:51:37.20 <Info:AAA.logout> Administrative account (admin)
logout from telnet (192.168.56.1)
07/01/2020 00:51:30.32 <Noti:log.ClrLogMsg> User admin: Cleared the log
messages in memory-buffer.
A total of 4 log messages were displayed.
```

Use this feature to execute specific scripts or apps based on the user connected to the switch.

3.1.2 Python Application

Starting with EXOS 15.7, Python application development capability has been added to EXOS.

Instead of writing a script to run to completion every time it is executed (manually or dynamically using UPM), you can create an application that lives as a new process in the system. This process can be started, terminated, or deleted, and runs into a dedicated Linux CGroup named “Other”, while official EXOS processes run in the “EXOS” CGroup.

***Note**: The Linux CGroup was introduced with EXOS 22. Prior to this release there was no differentiation between system and user-created processes. CGroups ensure that user-created applications cannot significantly impact processes in another CGroup. “Other” CGroup is limited, by default, to 10% CPU usage and 5% RAM usage, however these parameters are configurable.*

Processes run in a different system environment than user-created scripts. This environment is called expy and requires a different development approach. It is a more powerful environment that offers access to the dataplane.

The detailed API is documented here:

HTTPS://api.extremenetworks.com/EXOS/ProgramInterfaces/PYTHONAPI/

This API is based on the C SDK for EXOS and offers a wide variety of methods and functions to retrieve large amounts of data. For example, it can check if the process is running on a stackable switch, what role it has, it can manipulate packets, interact with the CLI to pass commands but also create its own CLI command, handle Authentication and so on.

To illustrate the use of this API, create a process that monitors the VLAN events on the switch. You need to subscribe to the event, as provided by the API.

```python
import exos.api.throwapi as throwapi

def event_cb(event, subs):
    print event

ev = throwapi.Subscription("vlan")
```

```
ev.sub(event_cb)
```

3.1.2.1 Create a Process

Enter the *create process command* and provide the necessary parameters:

The process name
The process creation (must be `python-module`)
The name of the Python application, without the .py suffix
The startup behavior, which can be either `on-demand` or `auto`
The VR from which you want it to run. The default is VR-Mgmt

The startup behavior, *on-demand*, runs once like a script would. *Auto* keeps the process running and adds the config line into the config file so it can be automatically restarted when a switch is rebooted.

Assuming your previous code example was in a file named “test.py”, you would create the process:

```
sw1.26 # create process test python-module test start auto
```

Verify that the process is running:

```
sw1.27 # show process test
Process Name      Version  Restart    State             Start Time         Group
--------------------------------------------------------------------------------
test              User         0     LoadCfg      Sun Jun  7 13:31:37 2020  Other
```

First verify that this process is present and running, then validate that it is running in the *Other* CGroup.

3.1.2.2 Create an Application

Manually create two VLANs, 42 and 43, and then connect something on the switch that will trigger the UPM script you configured in the previous chapter (this adds the port that goes up to VLAN 42). You will see the result on the switch, but no messages are displayed if you are connected via Telnet or SSH. When working with process, a print is only redirected to the console. To access the information, you must use the logging capability.

You must terminate and delete the process before you modify your program, after which you can recreate the process.

```
sw1.48 # terminate process test graceful
Do you want to save configuration changes to currently selected configuration
file (primary.cfg)? (y or n) No
You will lose test's configuration if you save the configuration after
terminating this process. Do you want to continue? (y/N) Yes
Successful graceful termination for test
sw1.49 #
sw1.49 # delete process test
```

Your program with the logging capability should look like this:

```
from exos import api
import exos.api.throwapi as throwapi
import logging
logger = logging.getLogger('test')
logger.setLevel(logging.DEBUG)
logHandler = api.TraceBufferHandler("testbuf", 20480)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(name)s:%(funcName)s.%(
lineno)s:: %(message)s"))
logger.addHandler(logHandler)
def event_cb(event, subs):
    logger.info(event)
ev = throwapi.Subscription("vlan")
ev.sub(event_cb)
```

You now can access the information when reading the trace buffer of your application:

```
sw1.50 # create process test python-module test start auto
creating test...
sw1.51 #
sw1.51 # create vlan 10-12
sw1.52 #
sw1.52 # debug ems show trace test testbuf
06/07/2020 14:06:37.002965 [200] <test:testbuf> Begin trace buffer
06/07/2020 14:06:54.479653 [221] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'create', 'timestamp': 1591538814.48, 'object': 'vlan',
'id': 'exos.vlan.create'}, 'data': {'vr_name': 'VR-Default', 'vlan_name':
'VLAN_0010'}}
06/07/2020 14:06:54.485521 [224] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'create', 'timestamp': 1591538814.49, 'object': 'vlan',
'id': 'exos.vlan.create'}, 'data': {'vr_name': 'VR-Default', 'vlan_name':
'VLAN_0011'}}
06/07/2020 14:06:54.491825 [227] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'create', 'timestamp': 1591538814.49, 'object': 'vlan',
'id': 'exos.vlan.create'}, 'data': {'vr_name': 'VR-Default', 'vlan_name':
'VLAN_0012'}}
```

Move a port from one VLAN to another to trigger additional information.

```
sw1.53 # config vlan 12 add port 1
VLAN 12 VLAN_0012:  Port 1 untagged has been auto-moved from VLAN "VLAN_0042"
to "VLAN_0012".

sw1.54 #
sw1.54 # debug ems show trace test testbuf
```

```
06/07/2020 14:06:37.002965 [200] <test:testbuf> Begin trace buffer
06/07/2020 14:06:54.479653 [221] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'create', 'timestamp': 1591538814.48, 'object': 'vlan',
'id': 'exos.vlan.create'}, 'data': {'vr_name': 'VR-Default', 'vlan_name':
'VLAN_0010'}}
06/07/2020 14:06:54.485521 [224] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'create', 'timestamp': 1591538814.49, 'object': 'vlan',
'id': 'exos.vlan.create'}, 'data': {'vr_name': 'VR-Default', 'vlan_name':
'VLAN_0011'}}
06/07/2020 14:06:54.491825 [227] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'create', 'timestamp': 1591538814.49, 'object': 'vlan',
'id': 'exos.vlan.create'}, 'data': {'vr_name': 'VR-Default', 'vlan_name':
'VLAN_0012'}}
06/07/2020 14:15:16.676949 [230] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'update', 'timestamp': 1591539316.68, 'object': 'vlan',
'id': 'exos.vlan.update'}, 'data': {'added': False, 'type': 'port', 'port':
(0, 0, 0), 'vlan_name': 'VLAN_0042'}}
06/07/2020 14:15:16.677836 [233] <test:testbuf> INFO:test:event_cb.15::
{'meta': {'action': 'update', 'timestamp': 1591539316.68, 'object': 'vlan',
'id': 'exos.vlan.update'}, 'data': {'added': True, 'type': 'port', 'port':
(1, 1, 0), 'vlan_name': 'VLAN_0012', 'vlan_id': 12}}
```

3.1.2.3 Add Proper Environment Validation

To demonstrate environment validation in more detail, first write another application with the basic checks any program should contain. Processes run in the expy environment, so when you write an application, make sure you are running in that environment.

```
def main():
    # Verify you are running under EXPY. You can't live without it.
    if not hasattr(sys, 'expy') or not sys.expy:
        print "Must be run within EXPY"
        return
```

Write a new App. This example uses the CLI method for illustration, as this is a common CALL for an App.

```
exos.api.exec_cli(cmds, timeout=0, ignore_errors=False)
```

Parameters:

- **cmds**: list of strings containing valid EXOS CLI command
- **timeout**: defaults to 0. This is a synchronous CALL, and the timeout tells the system how long it should wait for a return
- **ignore_errors**: Boolean. If set to False, which is the default, execution will stop after the first failed command

Returns: The output of the command: string

Raises:

- **CLICommandError(*error_msg*, *cmd*)**: A CLI command returned an error message. The *error_msg* attribute is the message received from the CLI and *cmd* is the command that was being run at the time
- **CLITimeoutError**: A CLI request timed out

Note: *You are presenting the synchronous CALL in this example, but asynchronous CALLs exist as well.*

Enhance your previous example by creating or deleting a VLAN is on the switch:

```
from exos import api
import exos.api.throwapi as throwapi
import sys
import logging

logger = logging.getLogger('test')
logger.setLevel(logging.DEBUG)

logHandler = api.TraceBufferHandler("testbuf", 20480)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(name)s:%(funcName)s.%(
lineno)s:: %(message)s"))

logger.addHandler(logHandler)
def event_cb(event, subs):
    meta = event.get('meta')
    data = event.get('data')

    # Here are some CLI commands based on VLAN events
    if meta.get('action') == 'create':
        api.exec_cli(['config vlan {} description "This is a description for VLAN
 {}"'.format(data.get('vlan_name'), data.get('vlan_name'))])
    elif meta.get('action') == 'delete':
        api.exec_cli(['create log message "Ohoh! VLAN {} has been deleted"'.forma
t(data.get('vlan_name'))])

def main():
    # Verify you are running under EXPY. You can't live without it.
    if not hasattr(sys, 'expy') or not sys.expy:
        print "Must be run within EXPY"
        return
    # Subscribe to vlan events
    ev = throwapi.Subscription("vlan")
    ev.sub(event_cb)
main()
```

***Note**: When you need double quotes for a CLI command in a Python string, you can use a single quote to delimit the string. Another solution is to use double quote and escape the inner ones with a backslash "\". Failing to this will result in an error.*

Your program reacts as expected on a switch.

```
sw1.12 # create process test python-module test start auto
creating test...
* sw1.13 #
* sw1.13 # create vlan 42
* sw1.14 #
* sw1.14 # sh vlan description
-----------------------------------------------------------------------------
Name            VID  Description
-----------------------------------------------------------------------------
Default         1
interco         4094
Mgmt            4095 Management VLAN
VLAN_0042       42   This is a description for VLAN VLAN_0042
-----------------------------------------------------------------------------

> Indicates description string truncated past 57 characters

Total number of VLAN(s) : 4
* sw1.15 #
* sw1.15 # delete vlan 42
* sw1.16 #
* sw1.16 # sh log
06/08/2020 12:03:34.58 <Info:System.userComment> Ohoh! VLAN VLAN_0042 has
been deleted
06/08/2020 12:03:14.89 <Noti:log.ClrLogMsg> User admin: Cleared the log
messages in memory-buffer.

A total of 2 log messages are displayed.
```

3.2 External APIs

More advanced automation solutions manage switches from external resources, running from an application on a server or VM. EXOS offers several APIs.

3.2.1 RESTCONF API

The latest API introduced with EXOS is the RESTConf, which follows the Openconfig model and works in conjunction with the Python module restconf.pyz. This module is available on Extreme Networks github:

HTTPS://github.com/extremenetworks/EXOS_Apps/tree/master/REST

***Note**: The RESTConf module is bundled in EXOS since release 22.4, but is backward compatible with EXOS 22.1, by adding the restconf.pyz module to the system.*

3.2.1.1 RESTCONF Documentation

The documentation is accessible either from the Extreme Networks documentation site, or directly from a switch running the minimal version required (the EXOS web server must be enabled).

The link to the documentation on Extreme Networks site is:

http://api.extremenetworks.com/EXOS/ProgramInterfaces/RESTCONF/RESTCONF.html

To access the documentation directly from a switch (or VM):

http(s)://<switch IP>/apps/restconfdoc

3.2.1.2 Working with EXOS RESTCONF

This section describes some examples using Python 3 to work with EXOS switches. To facilitate the use of RESTCONF CALLs, Extreme Networks offers a Python class the teaches you how to create the CALLs. The Python class is available on github:

HTTPS://github.com/extremenetworks/EXOS_Apps/blob/master/REST/examples/restconf.py

Note: The restconf python class is included by default with XMC Python Engine since XMC 8.2. The latest version of the class – v1.2.0.0 at the time of writing - should be part of XMC 8.5.

3.2.1.3 How to Access Restconf

EXOS Restconf supports GET, POST, PUT, PATCH and DELETE HTTP methods.

You must authenticate to access the API. By default, basic authentication using a login and password is available. When the session is successfully authenticated, a token is generated. This token allows you to make multiple API CALLs without the need to reauthenticate, as long as the token is included as a cookie in the request header.

***Note**: The duration of the token is set to 86400 seconds, which is 1 day.*

EXOS Restconf implementation supports both HTTP and HTTPS protocols. By default, out-of-the-box, EXOS switches only have HTTP enabled. The Python class restconf.py tries both protocols, starting with HTTPS. However, the best practice is to use HTTPS for data integrity and confidentiality.

To access the Restconf server on a switch, RFC 8040 requires a common URL as the root. The root resource for EXOS is `/rest/restconf/`. The datastore is represented by a node named `data`.

***Note**: All methods are supported on data.*

Enable HTTPS on EXOS

To enable HTTPS on an EXOS switch, first enable SSL. The following example starts with a factory default switch (or VM):

```
sw1.2 # show ssl
HTTPS Port Number: 443 (Disabled)
Signature Algorithm configured: sha512 With RSA Encryption
Certificate and Private key not configured
Manufacturing certificate: Not present
sw1.3 #
sw1.3 # config ssl certificate privkeylen 4096 country fr organization extreme common-
name extreme
................++
...................................++
Storing the private key. This may take some time.
.Done
sw1.4 #
sw1.4 # show ssl
HTTPS Port Number: 443 (Enabled)
Signature Algorithm configured: sha512 With RSA Encryption
Private Key matches the Certificate's public key.
RSA Private Key: 4096
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
    Signature Algorithm: sha512WithRSAEncryption
        Issuer: C=fr, O=extreme, CN=extreme
        Validity
            Not Before: Jun  9 10:38:42 2020 GMT
            Not After : Jun  9 10:38:42 2021 GMT
        Subject: C=fr, O=extreme, CN=extreme

Manufacturing certificate: Not present
sw1.4 #
sw1.4 # enable web HTTPS
sw1.5 # disable web http
```

This example uses self-signed certificates. This is adequate for testing but will generate warning messages and could potentially result in errors for some applications.

***Note**: The requests module, and especially urllib3, produces exceptions if you use HTTPS with insecure certificates. To remove these exceptions, add the following line to the Python class, after you import urllib3.*

```
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
```

You will also need to add the `verify=False` *parameter to the request CALLs.*

EXOS allows you to install customer certificates that have been signed by trusted authorities.

3.2.1.4 Using Restconf with Python

Use the restconf.py class available on Extreme Networks Github. At the time of writing of this document, the version of the class is 1.2.0.0.

***Note**: The restconf.py class is compatible with Python 2.7 and 3.x. It tries HTTPS first, then fallback to HTTP if unsuccessful.*

This example creates a session to your switch and list all available VLANs. This example uses Argparse to manage the parameters from the command line.

```python
from restconf import Restconf
import json
import getpass
import argparse
# manage your arguments
def get_params():
    parser = argparse.ArgumentParser(prog = 'RestDemo')
    parser.add_argument('-i', '--ip',
            help='IP Address of the switch',
            required=True)
    parser.add_argument('-u', '--username',
            help='Login username for the remote system')
    parser.add_argument('-p', '--password',
            help='Login password for the remote system',
            default='')
    args = parser.parse_args()
    return args

def main():
    args = get_params()

    if args.username is None:
        # prompt for username
        args.username = input('Enter remote system username: ')
        # also get password
        args.password = getpass.getpass('Remote system password: ')
    # open a restconf session
    rest = Restconf(args.ip, args.username, args.password)
    # you make a GET API call for all the vlans
    info = rest.get('data/openconfig-vlan:vlans')
    data = info.json()

    vlans = data.get('openconfig-vlan:vlans').get('vlan')
    for vlan in vlans:
        print("Found VLAN {} with VID {}".format(vlan.get('state').get('name'), v
lan.get('vlan-id')))
main()
```

As a result, you can see all existing VLANs on the switch:

```
C:\Extreme API with Python> rest_example.py -i 192.168.56.121 -u admin
Found VLAN Default with VID 1
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
```

Locating existing VLANs is easy, as it was just a CALL to the root of the VLANs datastore. For demonstration purposes, you can enhance this example to create a new VLAN and delete an existing one. To modify the configuration, you must understand the YANG model, used in Openconfig.

Refer to the Restconf documentation, or do a GET (using postman for example) you will see the following information about VLANs:

```
{
  "openconfig-vlan:vlans": {
    "vlan": [
      {
        "vlan-id": "1",
        "state": {
          "status": "ACTIVE",
          "vlan-id": 1,
          "name": "Default",
          "tpid": "oc-vlan-types:TPID_0x8100"
        },
        "config": {
          "status": "ACTIVE",
          "vlan-id": 1,
          "name": "Default",
          "tpid": "oc-vlan-types:TPID_0x8100"
        }
      },
[…]
}
```

Below your endpoint is a VLAN entry which is a list of the VLANs. For each entry in the list, you will see the element id (in this case, the VLAN id), a state container and a config container.

The Openconfig data model is very consistent, which means that once you understand it, you can easily access any data: it always follows the same pattern.

To create a VLAN, manipulate the config container following the same structure. To delete a VLAN, simply point to the endpoint.

```
from restconf import Restconf
import json
import getpass
import argparse

# manage your arguments
def get_params():
    parser = argparse.ArgumentParser(prog = 'RestDemo')
    parser.add_argument('-i', '--ip',
            help='IP Address of the switch',
            required=True)
    parser.add_argument('-u', '--username',
            help='Login username for the remote system')
    parser.add_argument('-p', '--password',
            help='Login password for the remote system',
            default='')
    args = parser.parse_args()
    return args

def list_vlans(rest):
    # you make a GET API call for all the vlans
    info = rest.get('data/openconfig-vlan:vlans')
    data = info.json()

    vlans = data.get('openconfig-vlan:vlans').get('vlan')
    for vlan in vlans:
        print("Found VLAN {} with VID {}".format(vlan.get('state').get('name'), v
lan.get('vlan-id')))

def main():
    args = get_params()
    if args.username is None:
        # prompt for username
        args.username = input('Enter remote system username: ')
        # also get password
        args.password = getpass.getpass('Remote system password: ')

    # open a restconf session
    rest = Restconf(args.ip, args.username, args.password)

    # you list the existing vlans prior adding one
    list_vlans(rest)

    # you prepare the data to send
    url = "data/openconfig-vlan:vlans/"
    data = {}
    vlan = {}

    vlan["config"] = {"name": "H2G2", "status": "ACTIVE", "tpid": "oc-vlan-
types:TPID_0x8100", "vlan-id": 42}
    data["openconfig-vlan:vlans"] = [vlan]

    # you make a POST API call
    r = rest.post(url, data)

    # you list the existing vlans after adding one to check
    print("-"*42)
    list_vlans(rest)

    # you delete it now
    del_url = url + "vlan=42"
    rest.delete(del_url)

    # you list the existing vlans after to check again
    print("-"*42)
    list_vlans(rest)

main()
```

The result:

```
C:\Extreme API with Python> rest_example.py -i 192.168.56.121 -u admin
Found VLAN Default with VID 1
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
------------------------------------------
Found VLAN Default with VID 1
Found VLAN H2G2 with VID 42
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
------------------------------------------
Found VLAN Default with VID 1
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
```

On the switch, you can see the actions have happened, assuming your Python application (from chapter 3.1.2) is still running.

```
sw1.10 # sh log
06/09/2020 23:17:01.39 <Info:AAA.logout> Administrative account (admin)
logout from app (192.168.56.1)
06/09/2020 23:16:49.58 <Info:System.userComment> Ohoh! VLAN H2G2 has been
deleted
06/09/2020 23:16:41.39 <Info:AAA.authPass> Login passed for user admin
through app (192.168.56.1)
06/09/2020 23:16:30.52 <Noti:log.ClrLogMsg> User admin: Cleared the log
messages in memory-buffer.

A total of 4 log messages are displayed.
```

To change the configuration of an existing VLAN, use the PATCH HTTP method directly on the endpoint's config container to send the modified parameter.

Add the following piece of code to your example:

```
    # you add the vlan again
    r = rest.post(url, data)

    # you list the existing vlans after to check again
    print("-"*42)
    list_vlans(rest)

    # you change the name
    patch_url = url + "vlan=42" + "/config/"
    info = {}
    info["openconfig-vlan:config"] = {"name": "Zaphod"}
    rest.patch(patch_url, info)
    # you list the existing vlans after to check again
    print("-"*42)
    list_vlans(rest)
```

Adding this code results in re-creating VLAN “H2G2”, and then renaming it to Zaphod:

```
C:\Extreme API with Python> rest_example.py -i 192.168.56.121 -u admin
Found VLAN Default with VID 1
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
------------------------------------------
Found VLAN Default with VID 1
Found VLAN H2G2 with VID 42
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
------------------------------------------
Found VLAN Default with VID 1
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
------------------------------------------
Found VLAN Default with VID 1
Found VLAN H2G2 with VID 42
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
------------------------------------------
Found VLAN Default with VID 1
Found VLAN Zaphod with VID 42
Found VLAN VLAN_0054 with VID 54
Found VLAN interco with VID 4094
```

The result also appears for the switch:

```
sw1.11 # sh log
06/09/2020 23:30:29.75 <Info:AAA.logout> Administrative account (admin) logout from app
(192.168.56.1)
06/09/2020 23:30:17.91 <Info:System.userComment> Ohoh! VLAN H2G2 has been deleted
06/09/2020 23:30:09.74 <Info:AAA.authPass> Login passed for user admin through app (192.168.56.1)
06/09/2020 23:17:01.39 <Info:AAA.logout> Administrative account (admin) logout from app
(192.168.56.1)
06/09/2020 23:16:49.58 <Info:System.userComment> Ohoh! VLAN H2G2 has been deleted
06/09/2020 23:16:41.39 <Info:AAA.authPass> Login passed for user admin through app (192.168.56.1)
06/09/2020 23:16:30.52 <Noti:log.ClrLogMsg> User admin: Cleared the log messages in memory-
buffer.

A total of 7 log messages are displayed.
* sw1.11 #
* sw1.11 # sh vlan
Untagged ports auto-move: Inform
-------------------------------------------------------------------------------------------------
Name            VID  Protocol Addr       Flags                         Proto  Ports  Virtual
                                                                              Active router
                                                                              /Total
-------------------------------------------------------------------------------------------------
```

```
Default         1    -----------------------------------------------    ANY    0 /0   VR-Default
interco         4094 10.1.1.2       /24  -f-------------------------    ANY    1 /1   VR-Default
Mgmt            4095 192.168.56.121 /24  --------------------------    ANY    1 /1   VR-Mgmt
VLAN_0054       54   -----------------------------------------------    ANY    0 /0   VR-Default
Zaphod          42   -----------------------------------------------    ANY    0 /0   VR-Default
-----------------------------------------------------------------------------------------------
Flags : (B) BFD Enabled, (c) 802.1ad customer VLAN, (C) EAPS Control VLAN,
[…]


Total number of VLAN(s) : 5
```

The same logic applies to any datastore and allows you to manage switches in a programmatic way, using an open standard.

3.2.2 JSON-RPC API

The JSON-RPC API offers another way to interact with EXOS switches. To see documentation, visit this link:

HTTPS://documentation.extremenetworks.com/app_notes/MMI/121152_MMI_Application_Release_Notes.pdf

This document also exists in html:

HTTPS://api.extremenetworks.com/EXOS/ClientApplications/JSONRPC/

***Note**: This capability was introduced with EXOS 21.1 and requires that the web server be enabled. This is the default behavior for EXOS.*

You can also find information and examples of JSON-RPC on the Extreme Networks github:

HTTPS://github.com/extremenetworks/EXOS_Apps/tree/master/JSONRPC

3.2.2.1 JSON-RPC Overview

JSON-RPC is a Remote Procedure CALL (RPC) returning JSON formatted information. It allows you to send CLI commands, run scripts remotely, or run a Python application via HTTP and receive a response formatted in JSON.

The main benefits are ease-of-use, and the lack of a requirement for strict data modeling on the system. Using JSON-RPC with EXOS allows you to send any valid CLI command, meaning that all features are accessible immediately.

3.2.2.2 EXOS JSON-RPC

The EXOS JSON-RPC implementation supports both HTTPS and HTTP protocols. It requires basic authentication (login and password) but supports a token for subsequent requests. The token is added as a cookie in the request header. The duration of the token defaults to 86400 seconds, which is 1 day.

3.2.2.3 Using JSON-RPC with Python

As with the RESTCONF API, a Python class is proposed on the Extreme Networks github to facilitate its use.

HTTPS://github.com/extremenetworks/EXOS_Apps/blob/master/JSONRPC/jsonrpc.py

***Note**: At the time of writing of this document, the latest version of the JSON-RPC class is 2.0.0.4.*

The class uses different methods, depending on the use case. The most common method is using CLI commands. This is not the only solution, however, and you can also use it to remotely run scripts on a switch or run a Python application. Scripts that you run remotely on a switch are not present on the switch but live instead in your system. This method handles the transfer to the switch for you.

This section describes the CLI method, which is the most common method.

Note: The JSON-RPC Python class is included by default with XMC Scripting Engine since XMC 8.2.

First, create a few VLANs on a switch (or VM) using the provided Python class. To make things a bit different from previous examples, in this example, you manipulate a file as the input for your application. The file must contain the CLI commands, one per line, that you want to run on a switch.

Name the CLI commands file `cmds.txt`:

```
create vlan 10-15
config vlan 10-15 add port 1 tag
show vlan port 1
```

The following example shows one way to code your application:

```python
from jsonrpc import JsonRPC
import argparse
import getpass
import json

# manage your arguments
def get_params():
    parser = argparse.ArgumentParser(prog = 'JSONRPCDemo')
    parser.add_argument('-i', '--ip',
            help='IP Address of the switch',
            required=True)
    parser.add_argument('-f', '--filename',
            help='Filename with valid EXOS CLI commands',
            required=True)
    parser.add_argument('-u', '--username',
            help='Login username for the remote system')
    parser.add_argument('-p', '--password',
            help='Login password for the remote system',
            default='')
    args = parser.parse_args()
    return args
def main():
    args = get_params()

    if args.username is None:
        # prompt for username
        args.username = input('Enter remote system username: ')
        # also get password
        args.password = getpass.getpass('Remote system password: ')

    with open(args.filename, "r") as f:
        cmds = f.read().splitlines()

    # you open a jsonrpc session to the switch
    jsonrpc = JsonRPC(args.ip, args.username, args.password)

    # you execute the CLI commands from the file
    for cmd in cmds:
        response = jsonrpc.cli(cmd)
        rslt = response.get('result')
        print("Executed CLI command {}".format(cmd))
        print("result: {}".format(rslt[0].get('CLIoutput')))
main()
```

The goal is to present the concepts and show how to manipulate APIs. As a result, these code examples are not meant to be the most efficient or handle all exceptions and errors.

In this example you send the CLI commands to the switch using JSON-RPC and print the CLI output from the response. The CLI output is a string of what is displayed on the switch if you are connected to it via Console, Telnet or SSH. It is normal for some of the CLI commands to have no output.

When you run the application, you should see output similar to this:

```
C:\Extreme API with Python> jsonrpc_example.py -f cmds.txt -i 192.168.56.121 -u admin
Executed CLI command create vlan 10-15
result:
Executed CLI command config vlan 10-15 add port 1 tag
result:
Executed CLI command show vlan port 1
result: Untagged ports auto-move: Inform
-----------------------------------------------------------------------------------
```

```
Name              VID  Protocol Addr       Flags                          Proto  Ports  Virtual
                                                                                 Active router
                                                                                 /Total
-----------------------------------------------------------------------------------------------
interco           4094 10.1.1.2       /24  -f------------------------     ANY    1 /1   VR-Default
VLAN_0010         10   ----------------------------------------------     ANY    1 /1   VR-Default
VLAN_0011         11   ----------------------------------------------     ANY    1 /1   VR-Default
VLAN_0012         12   ----------------------------------------------     ANY    1 /1   VR-Default
VLAN_0013         13   ----------------------------------------------     ANY    1 /1   VR-Default
VLAN_0014         14   ----------------------------------------------     ANY    1 /1   VR-Default
VLAN_0015         15   ----------------------------------------------     ANY    1 /1   VR-Default
-----------------------------------------------------------------------------------------------
Flags : (B) BFD Enabled, (c) 802.1ad customer VLAN, (C) EAPS Control VLAN,
        (d) Dynamically created VLAN, (D) VLAN Admin Disabled,
        (E) ESRP Enabled, (f) IP Forwarding Enabled,
        (F) Learning Disabled, (i) ISIS Enabled,
        (I) Inter-Switch Connection VLAN for MLAG, (k) PTP Configured,
        (l) MPLS Enabled, (L) Loopback Enabled, (m) IPmc Forwarding Enabled,
        (M) Translation Member VLAN or Subscriber VLAN, (n) IP Multinetting Enabled,
        (N) Network Login VLAN, (o) OSPF Enabled, (O) Virtual Network Overlay,
        (p) PIM Enabled, (P) EAPS protected VLAN, (r) RIP Enabled,
        (R) Sub-VLAN IP Range Configured, (s) Sub-VLAN, (S) Super-VLAN,
        (t) Translation VLAN or Network VLAN, (T) Member of STP Domain,
        (v) VRRP Enabled, (V) VPLS Enabled, (W) VPWS Enabled,
        (Y) Policy Enabled

Total number of VLAN(s) : 9 (7 displayed)
```

However, the real focus is to work with JSON output, which is easier from a programming perspective.

***Note**: The JSON output is not documented, you must test your CALLs prior to writing your application.*

The following example lists all the VLANs from two switches, and extracts and displays information about these VLANs. For simplicity, hard code the information about the switches and the CLI command you want to use.

```python
from jsonrpc import JsonRPC

IPS = ["192.168.56.121", "192.168.56.122"]
USER = "admin"
PW = ""

def main():
    vlans = []

    for ip in IPS:
        # you open a jsonrpc session to the switch
        jsonrpc = JsonRPC(ip, USER, PW)

        response = jsonrpc.cli("show vlan")
        sw = {}
        sw['ip'] = ip
        sw['vlans'] = []
        for vlan in response.get('result'):
            if vlan.get('status') in ["MORE", "SUCCESS"]:
                info = {}
                data = vlan.get('vlanProc')
                info['ip'] = data.get('ipAddress')
                info['netmask'] = data.get('maskForDisplay')
                info['name'] = data.get('name1')
                info['vid'] = data.get('tag')

                sw['vlans'].append(info)

        vlans.append(sw)

    for entry in vlans:
        print("\nSwitch {} has {} VLANs".format(entry.get('ip'), len(entry.get('v
lans'))))
        print("data structure of the vlans:\n{}".format(entry.get('vlans')))
main()
```

The result:

```
C:\Extreme API with Python> jsonrpc_example.py

Switch 192.168.56.121 has 9 VLANs
data structure of the vlans:
[{'ip': '0.0.0.0', 'netmask': 0, 'name': 'Default', 'vid': 1}, {'ip':
'10.1.1.2', 'netmask': 24, 'name': 'interco', 'vid': 4094}, {'ip':
'192.168.56.121', 'netmask': 24, 'name': 'Mgmt', 'vid': 4095}, {'ip':
'0.0.0.0', 'netmask': 0, 'name': 'VLAN_0010', 'vid': 10}, {'ip': '0.0.0.0',
'netmask': 0, 'name': 'VLAN_0011', 'vid': 11}, {'ip': '0.0.0.0', 'netmask':
0, 'name': 'VLAN_0012', 'vid': 12}, {'ip': '0.0.0.0', 'netmask': 0, 'name':
'VLAN_0013', 'vid': 13}, {'ip': '0.0.0.0', 'netmask': 0, 'name': 'VLAN_0014',
'vid': 14}, {'ip': '0.0.0.0', 'netmask': 0, 'name': 'VLAN_0015', 'vid': 15}]

Switch 192.168.56.122 has 8 VLANs
data structure of the vlans:
[{'ip': '0.0.0.0', 'netmask': 0, 'name': 'Default', 'vid': 1}, {'ip':
'192.168.10.1', 'netmask': 24, 'name': 'foo1', 'vid': 4093}, {'ip':
'192.168.20.1', 'netmask': 24, 'name': 'foo2', 'vid': 4092}, {'ip':
'192.168.30.1', 'netmask': 24, 'name': 'foo3', 'vid': 4091}, {'ip':
'192.168.56.122', 'netmask': 24, 'name': 'Mgmt', 'vid': 4095}, {'ip':
'0.0.0.0', 'netmask': 0, 'name': 'VLAN_0020', 'vid': 20}, {'ip': '21.1.1.1',
'netmask': 24, 'name': 'VLAN_0021', 'vid': 21}, {'ip': '0.0.0.0', 'netmask':
0, 'name': 'VLAN_0022', 'vid': 22}]
```

The JSON output is a result of the EXOS CLI command shows the data structures that have been used to create this display on EXOS. It can be sometimes difficult to find the exact information for a given feature or protocol parameter.

***Note**: JSON output is created with the cli2json.py embedded Python script in EXOS. You can use it directly to see the output for any given command. The output cannot be formatted, so you must create a script to improve readability in printed output.*

Another method is to use the undocumented `debug cfgmgr show` commands. These commands directly access the CM objects in the backend. These commands can be very helpful and can be used with on-switch Python scripting, however their use is not always straightforward, and may require parameters that are impossible for you to find or to guess.

4 VOSS API

VOSS offers a RESTCONF API using the Openconfig model. VOSS powers the VSP product family.

***Note**: RESTCONF was added to VOSS starting with version 8.0.*

4.1 VOSS RESTCONF Documentation

The *Configuring User Interfaces and Operating Systems for VOSS* document is provided with any new release of the OS. This document uses VOSS 8.1.5 for the examples.

The link to this document for VOSS 8.1.5 is:

HTTPS://documentation.extremenetworks.com/VOSS/SW/81x/ConfigUIOSVOSS_8.1.5_CG.pdf

On-switch documentation is also available when the feature is enabled on the switch.

4.2 Enable RESTCONF

The RESCONF server is not enabled by default with VOSS. You must configure it on the switch.

```
voss01:1>enable
voss01:1#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
voss01:1(config)#application
voss01:1(config-app)#restconf enable
```

When this is done, you can access on-switch documentation for RESTCONF here:

http://<Switch_IP>:8080/apps/restconfdoc/

As with EXOS, both HTTPS and HTTP protocols are supported for RESTCONF on VOSS. To use HTTPS, you must enable TLS and install a certificate, using the following procedure:

```
Switch:1>enable
Switch:1#configure terminal
Enter configuration commands, one per line. End with CTRL/Z.
Switch:1(config)#application
Switch:1(config-app)#no restconf enable
Switch:1(config-app)#restconf install-cert-file /intflash/.cert/restconf-
cert.pem
Switch:1(config-app)#restconf tls
Switch:1(config-app)#restconf enable
```

4.3 Use RESTCONF with Python

Extreme Networks provides a Python class called RESTCONF, which is similar to the one for EXOS.

The way to use RESTCONF is identical to that of EXOS, and the endpoints follow the same logic. Be careful to use the default port used in VOSS for RESTCONF, which is 8080 for HTTP. It must be provided along with the IP address.

The RESTCONF Python class is available on the Extreme Networks github:

HTTPS://github.com/extremenetworks/ExtremeScripting/blob/master/VOSS/restconf.py

***Note**: Starting with XMC 8.5, the restconf_voss.py Python class is shipped by default with XMC.*

The following example retrieves all existing VLANs on a VOSS switch (or a VM), then adds one and then deletes it. To better illustrate how similar this process is to EXOS and VOSS, the same code base is reused, except the data required to create a VLAN on VOSS is modified to add a new parameter.

```
from restconf_voss import Restconf
import getpass
import argparse

DEFAULT_TCP_PORT = '8080'

# manage your arguments
def get_params():
    parser = argparse.ArgumentParser(prog = 'VossRestDemo')
    parser.add_argument('-i', '--ip',
            help='IP Address of the switch',
            required=True)
    parser.add_argument('-u', '--username',
            help='Login username for the remote system')
    parser.add_argument('-p', '--password',
            help='Login password for the remote system',
            default='')
    args = parser.parse_args()
    return args

def list_vlans(rest):
    # you make a GET API call for all the vlans
    info = rest.get('data/openconfig-vlan:vlans')
    data = info.json()

    vlans = data.get('openconfig-vlan:vlans').get('vlan')
    for vlan in vlans:
        print("Found VLAN {} with VID {}".format(vlan.get('state').get('name'), v
lan.get('vlan-id')))

def main():
    args = get_params()

    if args.username is None:
        # prompt for username
        args.username = input('Enter remote system username: ')
        # also get password
        args.password = getpass.getpass('Remote system password: ')

    # open a restconf session - you are assuming http
    rest = Restconf(args.ip + ':' + DEFAULT_TCP_PORT, args.username, args.passwor
d)

    # you list the existing vlans prior adding one
    list_vlans(rest)

    # you prepare the data to send
    url = "data/openconfig-vlan:vlans/"
    data = {}
    vlan = {}

    vlan["config"] = {"extreme-mod-oc-vlan:stg-id": 1, "name": "H2G2", "vlan-
id": 42}
    data["openconfig-vlan:vlans"] = [vlan]

    # you make a POST API call
    r = rest.post(url, data)

    # you list the existing vlans after adding one to check
    print("-"*42)
    list_vlans(rest)

    # you delete it now
    del_url = url + "vlan=42"
    rest.delete(del_url)
    # you list the existing vlans after to check again
    print("-"*42)
    list_vlans(rest)

main()
```

The result:

```
C:\Extreme API with Python> rest_voss.py -i 192.168.56.141
Enter remote system username: rwa
Remote system password:
Found VLAN Test with VID 40
Found VLAN Default with VID 1
------------------------------------------
Found VLAN Test with VID 40
Found VLAN H2G2 with VID 42
Found VLAN Default with VID 1
------------------------------------------
Found VLAN Test with VID 40
Found VLAN Default with VID 1
```

4.4 EXOS & VOSS Restconf Python Classes

On the Extreme Networks github, both Restconf Python classes share the same name (restconf.py) but have a different name in XMC 8.5. If you plan to use both Python classes together, you can rename one and import it with the “as” keyword to differentiate it, as shown here:

```
from restconf import Restconf as EXOSRestconf
from restconf_voss import Restconf as VOSSRestconf
```

Or place them in different sub-directories with `__init__.py` (empty) file, as shown here:

```
~/extreme $ tree
VOSS/
├── README.md
├── __init__.py
└── restconf.py
EXOS/
├── README.md
├── __init__.py
└── restconf.py
```

It would then resemble this:

```
from EXOS.restconf import Restconf as exos_restconf
from VOSS.restconf import Restconf as voss_restconf
```

5 XMC API

XMC (Extreme Management Center) uses several APIs, and the focus in this section is on the most recent addition with GraphQL support. This is also referred to as the NBI API (NorthBound Interface), through the extensive use of the Python capability built into XMC. This API can be accessed either externally or internally via the Python Scripting Engine.

***Note**: GraphQL is a query language developed by Facebook, before becoming public in 2015. It accesses data via HTTP and receives the content formatted in JSON. It is very similar to a REST API but has the benefit of sending only the information requested, instead of the entire tree. It provides a more efficient system, which is very appealing when manipulating large databases.*

This section is an updated (with XMC 8.4.4) and summary of the document available here:

HTTPS://api.extremenetworks.com/XMC/Scripting/Python_with_XMC_8.1_v0.94.pdf

5.1 Python Scripting Engine

XMC includes a Python Scripting Engine (Tasks > Scripts) based on Jython and running Jython 2.7.0.8. Support for Jython has been included with XMC 8.0.4 and several modules have been installed in addition to the standard library, such as requests and pip utility.

5.1.1 Default Location for Scripts

When you create or modify a script in the XMC UI, the script is saved in the following location:

```
/usr/local/Extreme_Networks/NetSight/appdata/scripting/overrides/
```

5.1.2 Add a User-Created Script

To add a user-created script, copy the Python script to this directory:

```
/usr/local/Extreme_Networks/NetSight/appdata/scripting/extensions/
```

From the embedded Python scripts, import the module.

Note: This directory doesn't exist by default. When created, it is automatically added to the system path and so becomes available for importing.

5.1.3 Python Modules Shipped with XMC

With the release of XMC 8.1.2, some default Python modules ship with XMC. They are located into the following directory:

```
/usr/local/Extreme_Networks/NetSight/appdata/scripting/system/
```

This is also where jsonrpc.py, restconf.py and restconf_voss.py are located. While Extreme Networks has an ongoing effort to update the included versions, the timing of releases may prevent their ability to ship the latest revision of the modules. These are available on the Extreme Networks Github, and it is advisable to update them to the latest version available.

***Note**: At the time of writing of this document, latest versions are v2.0.0.4 for jsonrpc.py, v1.2.0.0 for restconf.py and v1.0.0.1 for restconf_voss.py.*

5.1.4 System Path and Precedence

The following paths are automatically added to the system path:

```
appdata/scripting/overrides
appdata/scripting/extensions
appdata/scripting/system
appdata/scripting/
NetSight/jython/Lib
NetSight/jython/Lib/site-packages
NetSight/jython
```

If identical Python modules are found, the expected precedence is that `overrides` is used first.

5.1.5 Install a Library

To install a library, the easiest way is to use PIP. Starting with XMC 8.1.2, the PIP utility is part of the default XMC server installation. The commands to use the PIP utility to install a library are:

```
cd /usr/local/Extreme_Networks/NetSight/jython/bin
export JAVA_HOME=/usr/local/Extreme_Networks/NetSight/java
sudo chmod a+x pip
sudo chmod a+x jython
./pip install <module>
```

5.1.6 XMC Python Module

5.1.6.1 emc_vars

When you are writing Python scripts to be run directly from XMC, you can use a global variable named **emc_vars**. This variable is a Python dictionary containing all global variables in the system.

It is important to understand this key element. When a Python script must be executed on a device, this global variable can provide a great deal of useful information about that device, such as IP address, vendor profile, product family, etc.

With information easily accessible, you can create powerful scripts to run on different products. Another benefit of XMC is that you do not need to manage device access or store login credentials.

The XMC 8.4.4 list of variables in the emc_vars dictionary is shown below, as returned from this script executed in XMC:

```
for key,value in emc_vars.iteritems():
    print key
```

You can sort the output by category for easier reading, with explanations for the variable use.

```
time                              current time at server (HH:mm:ss z)
date                              current date at server (yyyy-MM-dd)
```

```
userDomain                        XMC user domain name
userName                          XMC user name
username
domain
```

```
serverVersion                     server version
serverIP                          server IP address
serverName                        server host name
auditLogEnabled                   True/False if audit log is supported
```

```
isExos                            True/False. Is this device an EXOS device?
vendor                            vendor name
family                            device family name
```

```
deviceConfigPwd
deviceASN                         AS number of the selected device
deviceSysOid                      device system object id
devicePwd                         login password for the selected device
deviceLogin                 login user for the selected device
deviceId                          device DB ID
deviceName                        DNS name of selected device
deviceVR                          device virtual router name
deviceSoftwareVer           software image version number on the device
deviceType                        device type of the selected device
deviceIP                          IP address of the selected device
deviceCliType                     method used to connect (Telnet/SSH)
deviceEnablePwd
```

```
managementPorts                   all ports with config role management
ports                             all device ports
accessPorts                 all ports with config role access
interSwitchPorts                  all ports with config role interswitch
```

```
scriptTimeout                     max script timeout in secs
abort_on_error                    True/False
scriptOwner                 scripts owner
```

```
javax.script.name
javax.script.engine_version
javax.script.language
javax.script.filename
javax.script.engine
```

```
jboss.http.port
jboss.server.log.dir
jboss.bind.address
jboss.bind.address.management
jboss.HTTPS.port
```

```
STATUS
USE_IPV6
extreme.hideLegacyDesktopApps
```

The `ports` variable returns a string containing all the ports, separated by commas.

5.1.6.2 emc_cli.send()

Another tool provided by XMC is the **emc_cli.send()** Python object. This object accepts several parameters. The first parameter is a string containing the CLI command, the second parameter is a Boolean value that enables you to choose to wait for a system or shell prompt, or not wait. If you set the Boolean value to False, no CLI output is returned. The Boolean value is optional, and the default is True. A third (optional) parameter is a timer, in seconds, to wait for information if needed.

There are several ways to use this Python object to retrieve information from CLI command execution:

- **isSucces()**: Boolean to represent outcome of the last command
- **getError()**: if it fails, contains the error as a string
- **getOutput()**: output captured or echoed back from the device (including the CLI command prompt) as a string

`isSuccess()` does not indicate whether the CLI command was successful or not, but it does show whether the `send()` has been completed correctly. The script handles the result of this CLI command by analyzing the CLI output.

For example:

```
# executes a show vlan command and prints the output
cli_results = emc_cli.send("show vlan")
cli_output = cli_results.getOutput()
print cli_output

# creates a dummy UPM profile
emc_cli.send("create upm profile \"Test\"", False)
emc_cli.send("Test", False)
cli_results = emc_cli.send(".")

# example of using timer - waiting for 3 seconds
emc_cli.send("show config", False, 3)
```

In this example, EXOS is the NOS. This is not restricted to one specific NOS. Any other NOS is eligible, if the device is accessible from XMC with a correct CLI Profile.

Because the emc_cli object connects to the device using either Telnet or SSH, any device from any vendor is accessible, however login banners and sub-prompts can vary from one vendor to another. XMC has a list of CLI rules to access the device.

Starting with XMC 8.1.2, you can customize the CLI rules or the regular expressions for prompt detection, by creating a file named `myCLIRules.xml`, located in the same directory as the `CLIRules.xml` file (names are case-sensitive).

```
/usr/local/Extreme_Networks/NetSight/appdata/scripting/
```

This file should be divided into sections containing regular expressions per vendor, in a similar fashion to that of the `CLIRules.xml` file. Typically, BOSS and VOSS access also uses this file.

Note*: CLI scripting for BOSS and VOSS is very inconsistent. Devices have a variety of different login banners and subprompts. Make sure that the CLI profile for a device is correct, as emc_cli relies on the CLI profile that is set for that device. By default, emc_cli will try to use the regular expressions defined in* `CLIRules.xml` *under the "Avaya" section, but because not all commands and prompts have been added. As a result, this might be the reason the script fails even if your CLI profile is correct.*

When you create the `myCLIRules.xml` file, the following logic applies when XMC tries to connect to a device:

- Checks if myCLIRules.xml exists. If it does, use the cliRule name in it.
- Checks if cliRule name exists in CLIRules.xml, if yes use this one.
- Finally, use the default rule name of “*”

The cliRule name normally comes from the device vendor profile. Each device (family, subfamily or device type) should have a property called cliRuleFileName (this name is misleading, it is really the cliRuleName, not a file name).

Note*: To set the cliRuleName dynamically from Python, invoke* `emc_cli.setCliRule`*.*

For example:
```
# must be called before using emc_cli.send
emc_cli.setCliRule("ruleName")
```

The CLI output returned by `emc_cli.send()` is a string that contains the CLI command used (first line).

Note*: For XMC 8.0.4 up to XMC 8.1.1, the string returned also included the trailing CLI prompt. XMC 8.1.2 removed it, and XMC 8.1.3 brought it back. You may need to update existing Python scripts.*

One way to remove extra lines, which is especially important if you are waiting for JSON-formatted output, is shown here:

```
import re

RegexPrompt = re.compile('.*[\?\$%#>]\s?$')
```

```
# Remove echoed command and final prompt from output
def cleanOutput(outputStr):
    lastLine = outputStr.splitlines()[-1:][0]
    if RegexPrompt.match(lastLine):
        lines = outputStr.splitlines()[1:-1]
    else:
        lines = outputStr.splitlines()[1:]
    return '\n'.join(lines)
```

5.1.6.3 Additional emc_cli Methods

This section has covered the most common method used for the emc_cli Python object. However, there are several other methods that can be useful. Examples of these are shown below.

You can list all the functions and methods provided with this object using a very basic Python code. The output with XMC 8.4.4 is shown below.

```
print dir(emc_cli)
```

The result:

```
['SSHEnabled', '__class__', '__copy__', '__deepcopy__', '__delattr__',
'__doc__', '__ensure_finalizer__', '__eq__', '__format__',
'__getattribute__', '__hash__', '__init__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'__subclasshook__', '__unicode__', 'class', 'cliRule', 'cliRuleObject',
'close', 'commandPrompt', 'commandReply', 'commandTimeout',
'commandTimeoutInMillis', 'connect', 'connected', 'equals', 'errorMessage',
'getClass', 'getCliRule', 'getCliRuleObject', 'getCommandPrompt',
'getCommandReply', 'getCommandTimeout', 'getIpAddress', 'getMaxCliOut',
'getPasswd', 'getPort', 'getSSHEnabled', 'getSaveCommand',
'getSessionTimeout', 'getShellPrompt', 'getUser', 'getVendor', 'hashCode',
'ipAddress', 'isConnected', 'logDebugMessage', 'logErrorMessage',
'logMessage', 'maxCliOut', 'notify', 'notifyAll', 'passwd', 'port', 'read',
'saveCommand', 'send', 'sessionTimeout', 'setCliRule', 'setCommandPrompt',
'setCommandReply', 'setCommandTimeout', 'setCommandTimeoutInMillis',
'setErrorMessage', 'setIpAddress', 'setMaxCliOut', 'setPasswd', 'setPort',
'setSSHEnabled', 'setSaveCommand', 'setSessionTimeout', 'setShellPrompt',
'setUser', 'setVendor', 'shellPrompt', 'toString', 'user', 'vendor', 'wait']
```

You can also find many set methods to use with emc_cli, for example, the session timeout.

```
# set the session timeout to 80 seconds
emc_cli.setSessionTimeout(80)
```

5.1.6.4 Add User-Input Variables to a Script

As of XMC 8.0.4, the metadata used with TCL can still be used as is, even if the syntax is more TCL-centric than it is compliant with Python. Starting with XMC 8.1.2, the Metadata fields with Python scripting have evolved so that the name field and the value field can be referenced directly.

***Note**: The legacy "set var name value" syntax is still supported for backward compatibility.*

Script interaction must be defined between the Metadata tags.

```
#@MetaDataStart
…
#@MetaDataEnd
```

You can add a description, but the most important part is the user-input variable definition. You must use the specific meta data shown below to define a variable that the user is prompted to set at when the script is executed.

```
#@VariableFieldLabel (description = "Enter Tag Type",
#                      type = String,
#                      required = yes,
#                      validValues = [tag,untag],
#                      readOnly = no,
#                      name = "myVar",
#                      value = "42"
#                     )
```

To specify multiple variables if needed, repeat the above definition.

You can specify multiple values in the VariableFieldLabel metadata.

- **description**: this is displayed before the value field
- **type**: the data format of data.
- **scope**: global (default) or device specific.
- **required**: yes or no
- **validValues**: a list of possible values, inside square brackets and comma-separated
- **readOnly**: access privilege setting, yes or no
- **name**: the name of the variable
- **value**: the default value of the variable, which you can override

As of XMC 8.4.4, the data type is string only. In this example, you define the user-input variable fields and their scope with a description.

```
#@MetaDataStart
#@DetailDescriptionStart
###############################################################################
#
# This script setups Fabric Connect for VOSS. It assumes reachabilty to each
# Fabric node (VSP) is available (via OoB or else)
#
#################################################################################
#@DetailDescriptionEnd

#@SectionStart (description = "Service Definition to create")
#    @VariableFieldLabel (description = "BVLAN 1",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "bvlan1",
#                     value = "4051"
#                     )

#    @VariableFieldLabel (description = "BVLAN 2",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "bvlan2",
#                     value = "4052"
#                     )

#    @VariableFieldLabel (description = "AREA",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "area",
#                     value = "49.0000"
#                     )

#    @VariableFieldLabel (description = "Nickname",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     validValues = [auto,custom],
#                     name = "nickname",
#                     value = "auto"
#                     )
#    @VariableFieldLabel (description = "Multicast Enable",
#                         type = string,
#                         required = yes,
#                         readOnly = no,
#                         validValues = [yes,no],
#                         name = "multicast",
#                         value = "no"
#                         )
#@SectionEnd

#@SectionStart (description = "Device Specific Data")
#    @VariableFieldLabel (description = "NNI Fabric Port List",
#                         type = string,
#                         required = yes,
#                         readOnly = no,
#                         name = "portlist",
#                         value = "1/1-1/3",
#                         scope = device
#                         )

#    @VariableFieldLabel (description = "Nickname Custom",
#                         type = string,
#                         required = no,
#                         readOnly = no,
#                         name = "nicknameCustom",
#                         value = "",
#                         scope = device
#                         )
#@SectionEnd
#@MetaDataEnd
```

These variables are then accessible from the emc_vars dictionary, which uses the name as the key.

This code snippet illustrates this principle:

```
def main():
    # you first perform some sanity checks
    familyType = emc_vars["family"]
    if familyType != "VSP Series":
        raise RuntimeError('Error: This script needs to be executed on a VSP')
    if int(emc_vars["bvlan1"]) > 4094 or int(emc_vars["bvlan1"]) < 2:
        raise RuntimeError('BVLAN 1 Id is out of range')
    if int(emc_vars["bvlan2"]) > 4094 or int(emc_vars["bvlan2"]) < 2:
        raise RuntimeError('BVLAN 2 Id is out of range')
    if int(emc_vars["bvlan1"]) == int(emc_vars["bvlan2"]):
        raise RuntimeError('Error: BVLAN 1 Id is identical than BVLAN 2 Id')
```

Writing a script from XMC is simple. The following example validates BGP information on an EXOS switch, using Restconf:

```
from restconf import Restconf

if emc_vars['family'] != 'Summit Series':
    print 'Must be run on EXOS'
    exit(0)

r = Restconf(emc_vars['deviceIP'], emc_vars['deviceLogin'], emc_vars['devicePwd'])

data = r.get('data/openconfig-bgp:bgp/neighbors')
bgp_data = data.json()

if bgp_data:
    bgp = bgp_data.get('openconfig-bgp:neighbors').get('neighbor')
    print 'Found {} BGP neigbhors'.format(len(bgp))

    for neighbor in bgp:
        print 'Received {} prefixes from {} in ASN {}'.format(
            neighbor['afi-safis']['afi-safi'][0]['state']['prefixes'].get('received'),
            neighbor['neighbor-address'],
            neighbor['state'].get('peer-as'))
```

When you run the script against a BGP router, you will see a result similar to:

```
Script Name: BGP Test
Date and Time: 2020-06-21T02:20:34.617
XMC User: root
XMC User Domain:
IP: 192.168.56.121
Found 1 BGP neighbors
Received 3 prefixes from 10.0.0.1 in ASN 65002
```

5.2 Workflow Engine

The Workflow Engine, Introduced with XMC 8.2, enables you to create complex actions based on events or alarms or that can be manually triggered. These actions execute several tasks in a logical progression, depending on the result of the previous task. This improvement enables you to create custom features.

The Workflow Engine relies on Python scripting, with some extra parameters and a few differences, which are described in the following sections.

5.2.1 emc_vars

With XMC 8.4.4, the emc_vars Python dictionary returns the following keys:

```
time
date
userDomain
userName
domain
username
serverVersion
serverIP
serverHTTPSPort
serverName
hostName
isExos
family
vendor
vrName
deviceNosIdName
deviceConfigPwd
devicePwd
deviceName
deviceVR
deviceType
deviceEnablePwd
deviceNosId
deviceASN
deviceSysOid
deviceLogin
deviceSoftwareVer
deviceCliType
deviceIP
devices
workflowPath
workflowStatus
workflowCreatedDateTime
workflowMessage
workflowCategory
```

```
workflowUpdatedBy
workflowexecutionId
workflowUpdatedDateTime
workflowDescription
workflowTimeout
workflowNosIds
workflowCreatedBy
workflowName
workflowVersion
activityMessage
activityDescription
activityCustomId
activityName
activityNosIds

scriptOwner
scriptName
scriptAssignment
scriptTimeout
scriptType
abort_on_error

javax.script.name
javax.script.engine_version
javax.script.language
javax.script.engine
output
STATUS
auditLogEnabled
failFast
extreme.hideLegacyDesktopApps
status
ports
USE_IPV6
jboss.http.port
jboss.bind.address.management
jboss.server.log.dir
jboss.bind.address
jboss.HTTPS.port
```

As shown, there are more keys in the emc_vars dictionary. Using a Python script, you can see the differences between the two environments, using as XMC 8.4.4 as a reference:

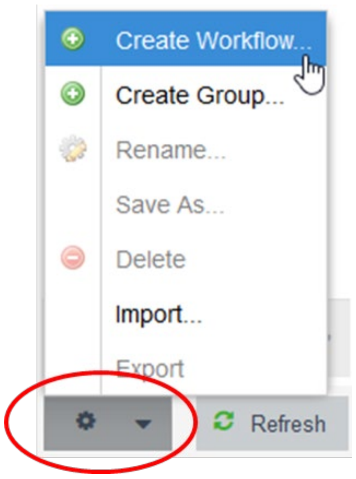
```
There are 46 entries in emc_vars in Scripting Engine
There are 72 entries in emc_vars in Workflow Engine
emc_vars not in Workflow Engine:
Not found: deviceId
Not found: managementPorts
Not found: accessPorts
Not found: interSwitchPorts
Not found: javax.script.filename
emc_vars not in Scripting Engine:
Not found: serverHTTPSPort
Not found: hostName
Not found: vrName
Not found: deviceNosIdName
Not found: deviceNosId
Not found: devices
Not found: workflowPath
Not found: workflowStatus
Not found: workflowCreatedDateTime
Not found: workflowMessage
Not found: workflowCategory
Not found: workflowUpdatedBy
Not found: workfloyouxecutionId
Not found: workflowUpdatedDateTime
Not found: workflowDescription
Not found: workflowTimeout
Not found: workflowNosIds
Not found: workflowCreatedBy
Not found: workflowName
Not found: workflowVersion
Not found: activityMessage
Not found: activityDescription
Not found: activityCustomId
Not found: activityName
Not found: activityNosIds
Not found: scriptName
Not found: scriptAssignment
Not found: scriptType
Not found: output
Not found: failFast
Not found: status
```

As a result, when you work with Python scripts that could be used in both environments, be careful when collecting information from emc_vars.

5.2.2 Create Workflows

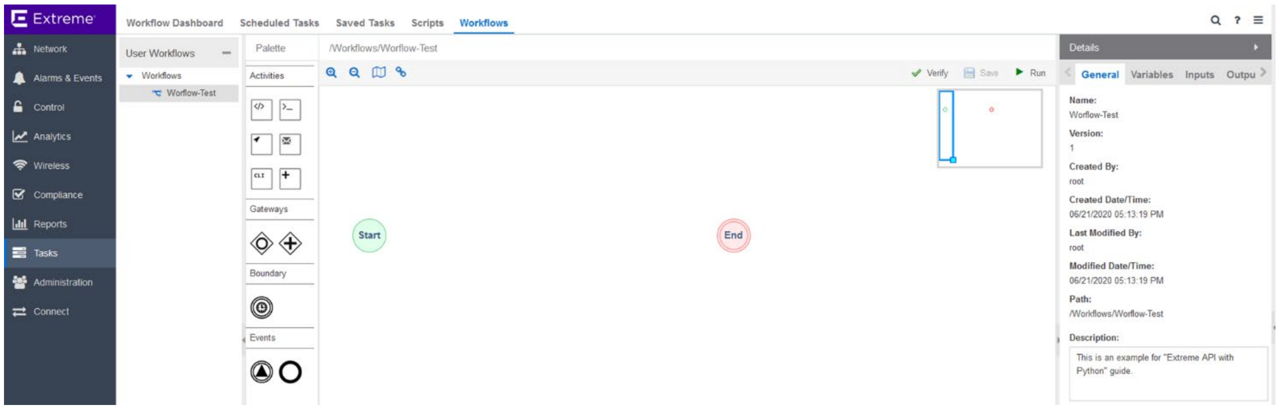
The Workflow Engine is accessible from the Tasks menu (Tasks > Workflows). You will need an Advanced license to create User-Workflows.

Select the gear icon, at the bottom of the page, then select “Create Workflow”. The new workflow appears in the User Workflows tab.



***Note**: You can also select an existing workflow and save it with a different name to use as a template for a new workflow.*

After you have created a new workflow, and entered a name and a description for it, you are ready to start editing it. Several windows are displayed.



Name:
Worflow-Test
Version:
1
Created By:
root
Created Date/Time:
06/21/2020 05:13:19 PM
Last Modified By:
root
Modified Date/Time:
06/21/2020 05:13:19 PM
Path:
/Workflows/Worflow-Test
Description:
This is an example for "Extreme API with Python" guide.

Next to the Menu bar, on the left side, you can see the Workflow List, the Palette, the Designer and finally the Details window.

From the Workflow List you can select any workflows available on the system. The Palette and the Designer panels allow you to create the logic of the workflow in a graphical and intuitive way.

From the Palette, select an item to place in the Designer by dragging and dropping the item between the Start and End buttons. The available items are grouped in categories, depending on their purpose:

- **Activities** are piece of code or actions that produce something. The Script Activity is most often used, but other activities are also available.
    Script Activity
    Shell Activity

HTTP Activity
Mail Activity
CLI Activity
Activity Group

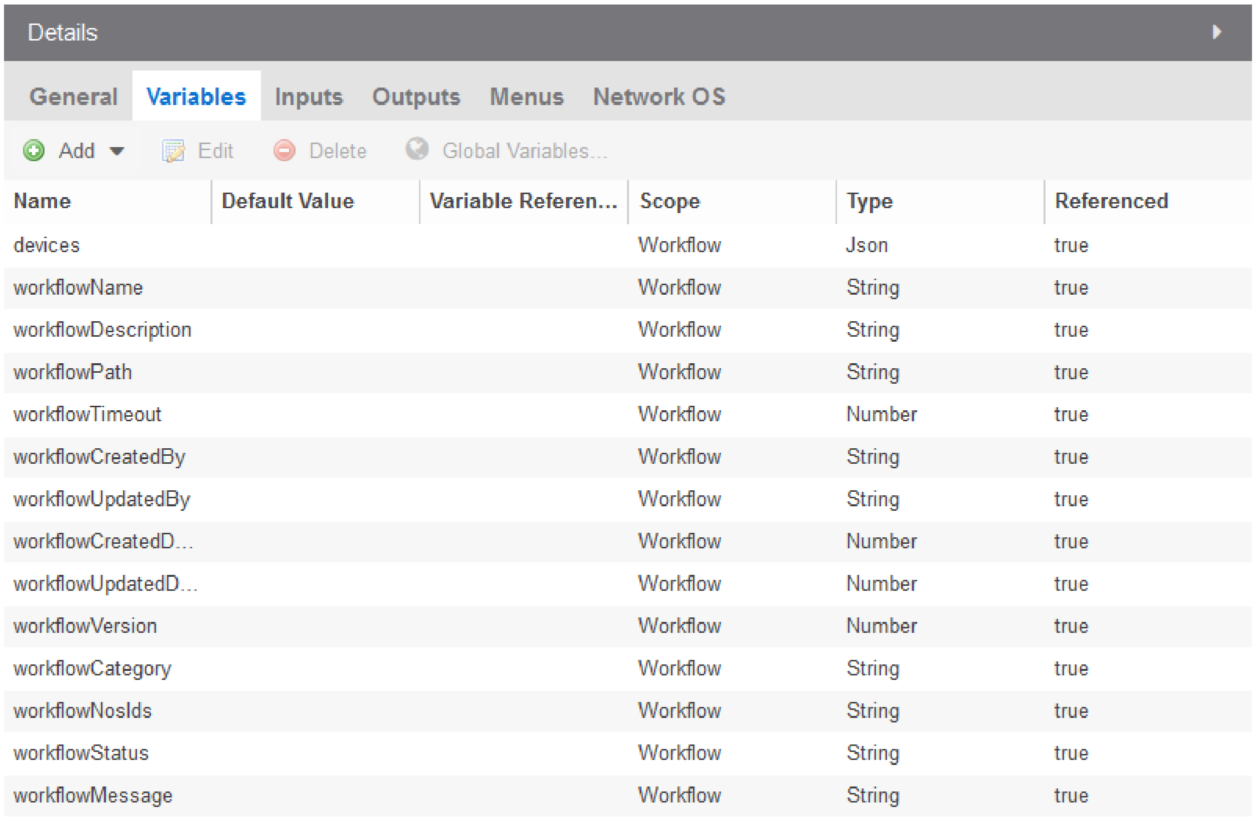
- **Gateways** are objects that allow different execution paths.
  Inclusive Parallel
  Parallel
- **Boundary** is a timer object that can be executed if an activity does not complete during a specified time. This allows you to follow a given path in the Workflow if this happens. The Workflow Engine checks every 10 seconds and triggers a timer in a range of N to N+10 seconds.
  Events allow you to end a path or generate an event when reached.

The Details panel is where you configure these settings.

5.2.3 Create Variables

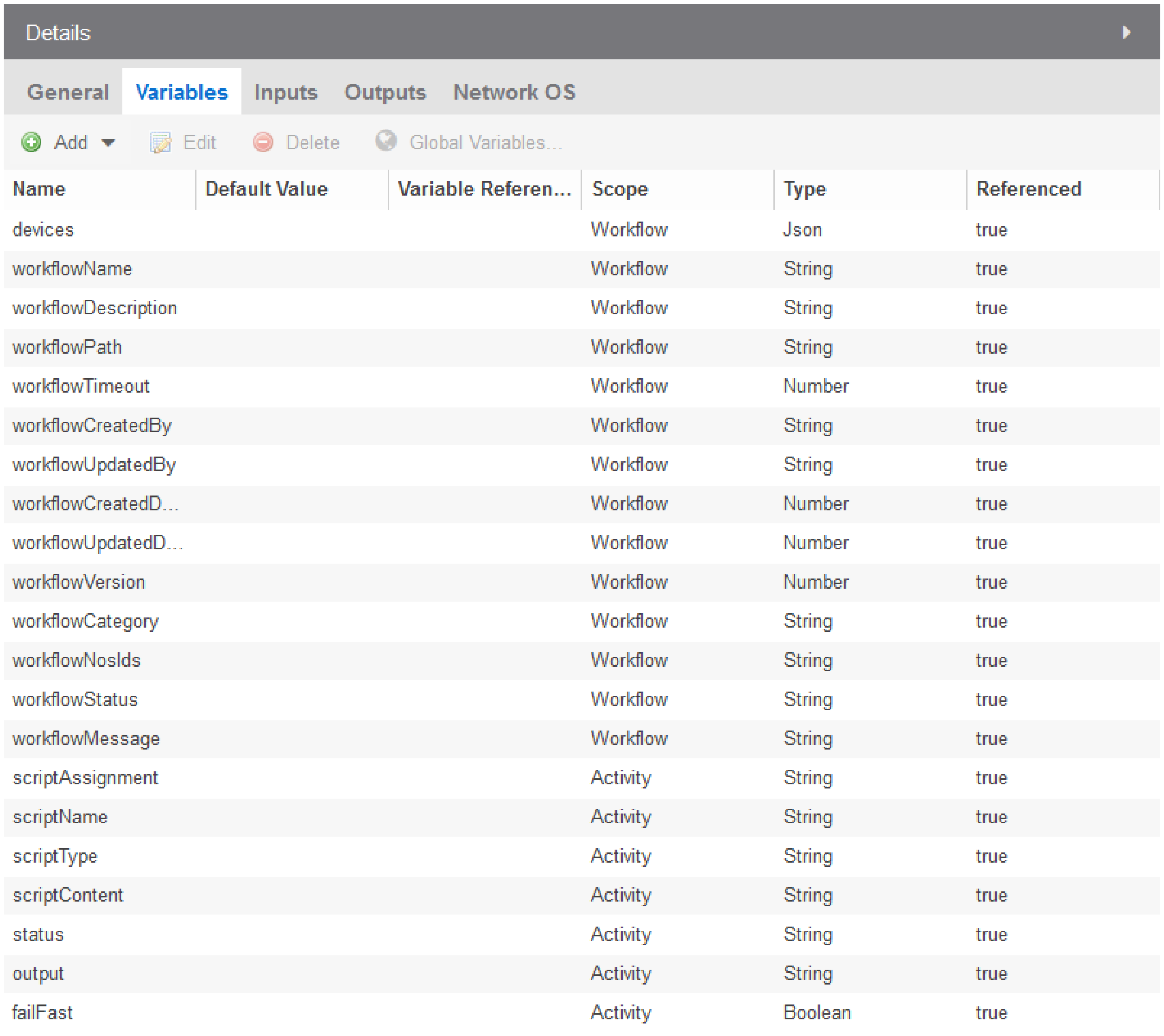
In Workflows, you can create variables to manipulate them. From the Details panel, in the Variables tab, you will see a list of all available variables. The items that appear in are dependent on the activity selected.

Using XMC 8.4.4 as an example, the list of variables that appears when you have no activity selected in a workflow is shown here:



Name	Default Value	Variable Referen...	Scope	Type	Referenced
devices			Workflow	Json	true
workflowName			Workflow	String	true
workflowDescription			Workflow	String	true
workflowPath			Workflow	String	true
workflowTimeout			Workflow	Number	true
workflowCreatedBy			Workflow	String	true
workflowUpdatedBy			Workflow	String	true
workflowCreatedD...			Workflow	Number	true
workflowUpdatedD...			Workflow	Number	true
workflowVersion			Workflow	Number	true
workflowCategory			Workflow	String	true
workflowNosIds			Workflow	String	true
workflowStatus			Workflow	String	true
workflowMessage			Workflow	String	true

If you select a Python script, the list expands to include the variables shown here:



Details

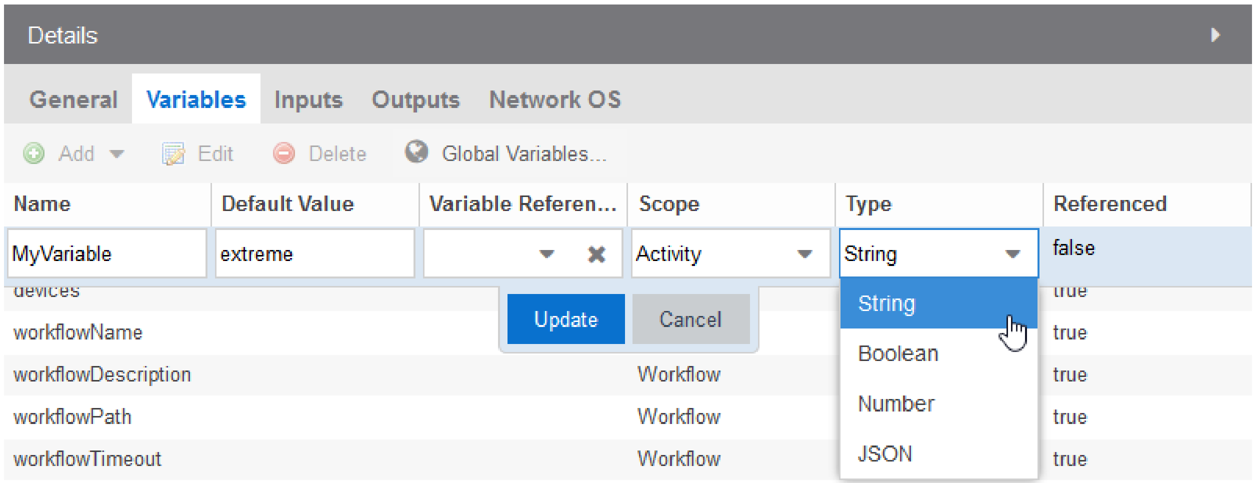
General | Variables | Inputs | Outputs | Network OS

Add | Edit | Delete | Global Variables...

Name	Default Value	Variable Referen...	Scope	Type	Referenced
devices			Workflow	Json	true
workflowName			Workflow	String	true
workflowDescription			Workflow	String	true
workflowPath			Workflow	String	true
workflowTimeout			Workflow	Number	true
workflowCreatedBy			Workflow	String	true
workflowUpdatedBy			Workflow	String	true
workflowCreatedD...			Workflow	Number	true
workflowUpdatedD...			Workflow	Number	true
workflowVersion			Workflow	Number	true
workflowCategory			Workflow	String	true
workflowNosIds			Workflow	String	true
workflowStatus			Workflow	String	true
workflowMessage			Workflow	String	true
scriptAssignment			Activity	String	true
scriptName			Activity	String	true
scriptType			Activity	String	true
scriptContent			Activity	String	true
status			Activity	String	true
output			Activity	String	true
failFast			Activity	Boolean	true

These variables are from your emc_vars dictionary, and some are only significant in some situations, running with a given activity.

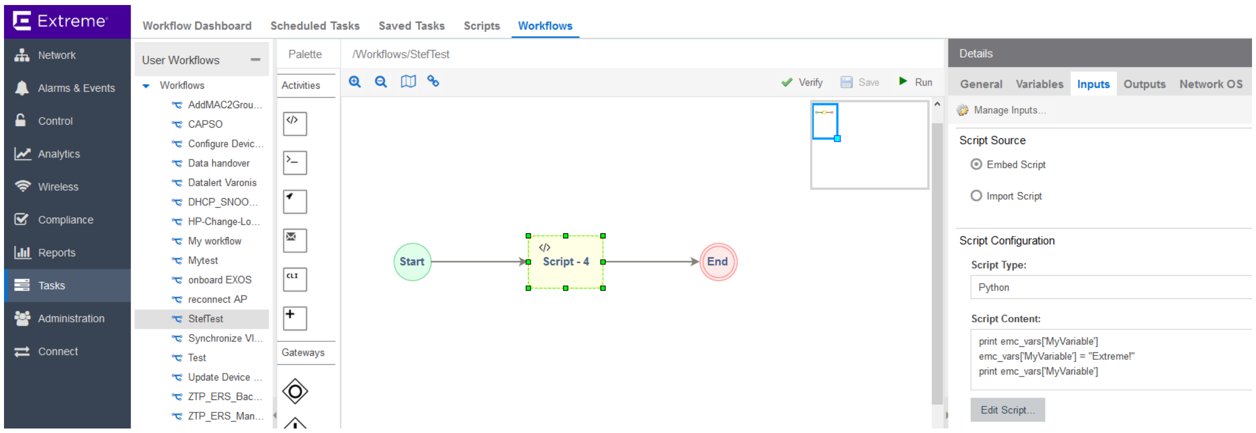
To create a new variable, select the Add button at the top of the Details panel. You can set the default value, type, and scope.



Name	Default Value	Variable Referen...	Scope	Type	Referenced
MyVariable	extreme		Activity	String	false
devices					true
workflowName					true
workflowDescription			Workflow		true
workflowPath			Workflow		true
workflowTimeout			Workflow		true

When created, in this example as a string type, the variable becomes accessible from the Python Script through the emc_vars dictionary.

Here's a quick example, creating a Python Script in a workflow. You connect the script with the Start and End gateways, using the arrows from one object to the other, then you can click on the Run button to execute the workflow, after a save.



Script Source

(● Embed Script) ○ Import Script

Script Configuration

Script Type: Python

Script Content:

```
print emc_vars['MyVariable']
emc_vars['MyVariable'] = "Extreme!"
print emc_vars['MyVariable']
```

The script is:

```
print emc_vars['MyVariable']

emc_vars['MyVariable'] = "Extreme!"

print emc_vars['MyVariable']
```

***Note**: Directly modifying an entry of emc_vars is not recommended.*

The output:

```
Script Name: StefTest_Script_-_4
Date and Time: 2020-06-26T19:32:36.361
XMC User: root
XMC User Domain:
IP:
extreme
Extreme!
```

***Note**: If an activity does not need to be run against a device, delete the* `devices` *variable so that the engine will not ask you to provide this input.*

5.2.4 emc_results

When you work with Workflows that have Inclusive Parallel gateways, you should provide the outcome of the action to select the path to follow using the emc_results Python object.

This Python object contains the following methods and functions:

```
print dir(emc_results)
```

The output, from XMC 8.4.4:

```
['DATE_FORMAT', 'DATE_FORMAT_STRING', 'ResultType', 'Status',
'TIMESTAMP_FORMAT', 'TIMESTAMP_FORMAT_STRING', 'TIMESTAMP_FORMAT_STRING_24',
'TIME_FORMAT', 'TIME_FORMAT_STRING', '__class__', '__copy__', '__deepcopy__',
'__delattr__', '__doc__', '__ensure_finalizer__', '__eq__', '__format__',
'__getattribute__', '__hash__', '__init__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'__subclasshook__', '__unicode__', 'addResult', 'batchId', 'childResults',
'class', 'clear', 'deviceIP', 'deviceId', 'elapsedTimeInMillis',
'elaspsedTimeInSecs', 'equals', 'errorMessage', 'fileName', 'get',
'getBatchId', 'getChildResults', 'getClass', 'getDeviceIP', 'getDeviceId',
'getElapsedTimeInMillis', 'getElaspsedTimeInSecs', 'getErrorMessage',
'getFileName', 'getId', 'getMessage', 'getName', 'getOutput',
'getOutputType', 'getResultUrl', 'getStartTime', 'getStatus', 'getStopTime',
'getVariables', 'hasErrors', 'hashCode', 'id', 'isOutputTruncated',
'message', 'name', 'notify', 'notifyAll', 'output', 'outputTruncated',
'outputType', 'put', 'putAll', 'resultUrl', 'setBatchId', 'setDeviceIP',
'setDeviceId', 'setErrorMessage', 'setFileName', 'setId', 'setMessage',
'setName', 'setOutput', 'setOutputTruncated', 'setOutputType',
'setResultUrl', 'setStartTime', 'setStatus', 'setStopTime', 'startTime',
'status', 'stopTime', 'toString', 'variables', 'wait']
```

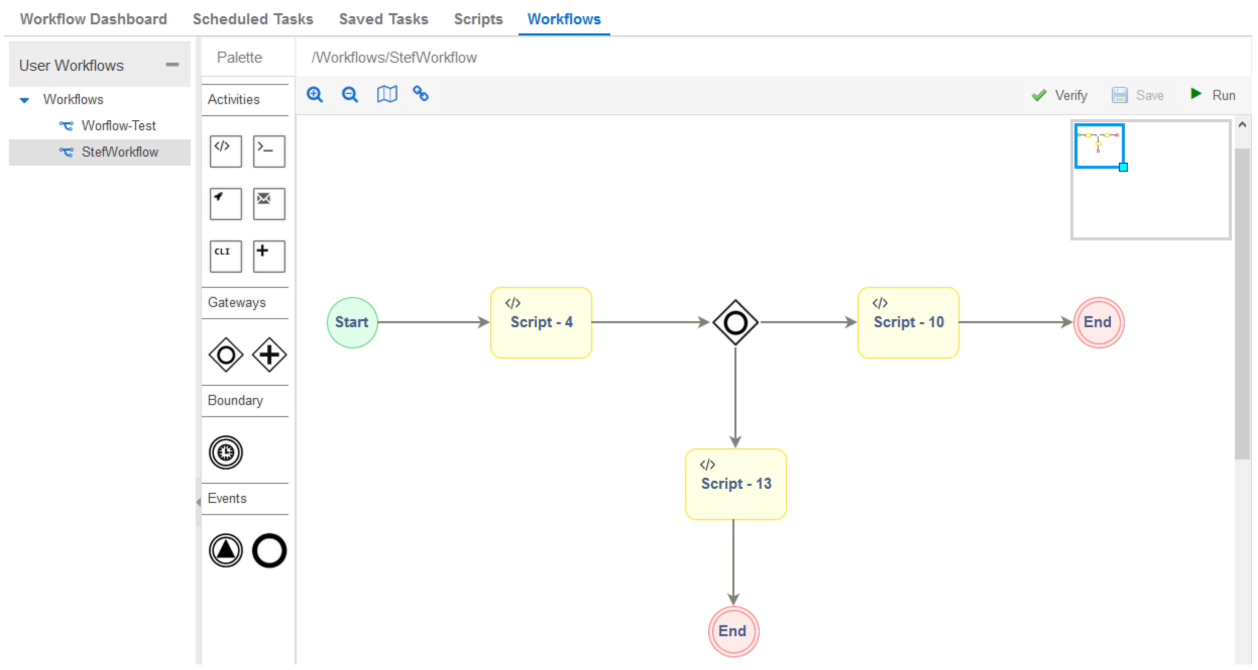
PUT is the most common method you will use.

With this method, you pass an argument variable as a string that you created previously in the activity, and its value, also as a string. You can test this value with gateways, and other activities. This is the correct way to change a variable value (if the variable has a scope of workflow).

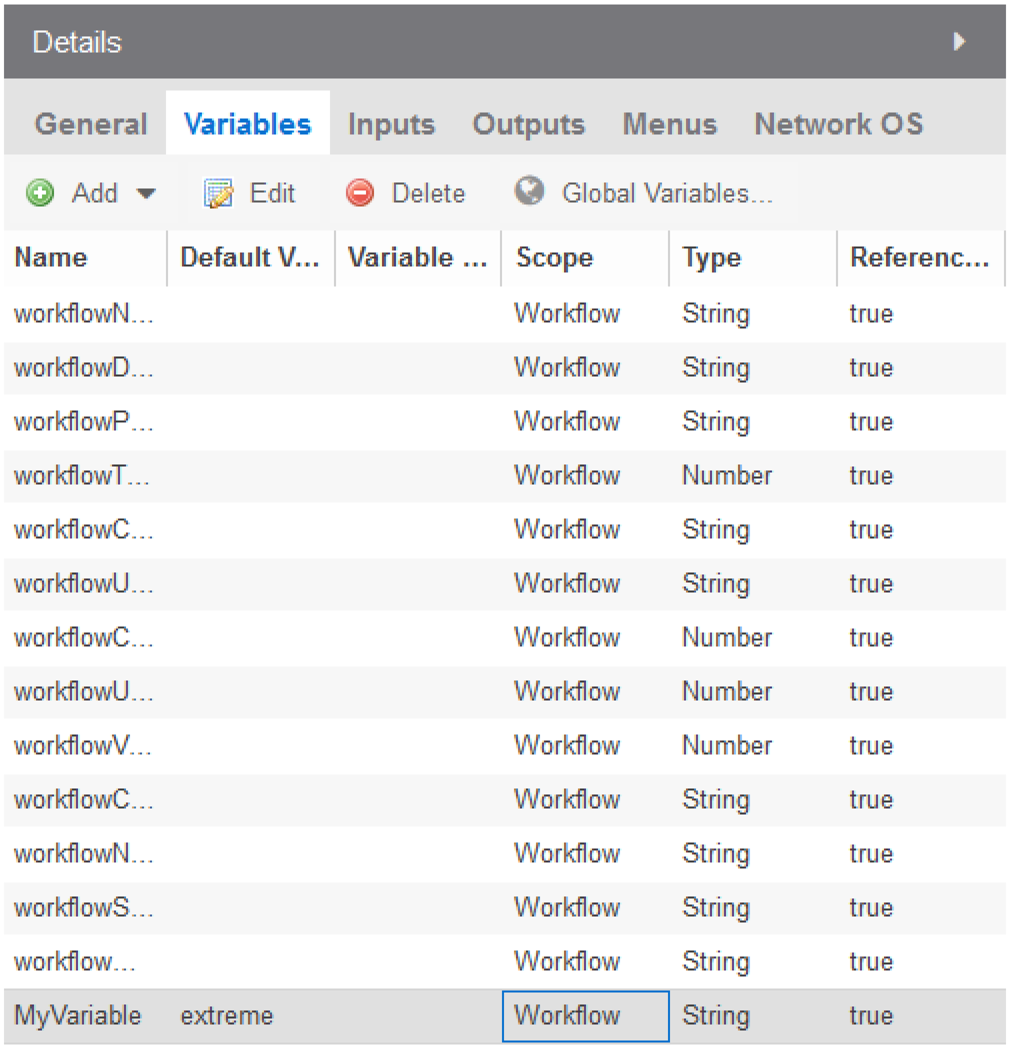
***Note**: You can also use this method to pass JSON data, using json.dumps().*

```
emc_results.put("MyVariable", "true")
```

To better illustrate this, modify your workflow so that you select a path based on the value of a variable. Delete the link between the script and the "End" gateway by selecting it then selecting on the trash can icon. Then add an inclusive parallel gateway leading to two new scripts that will end the workflow by adding another "End" gateway and connecting the scripts to it.



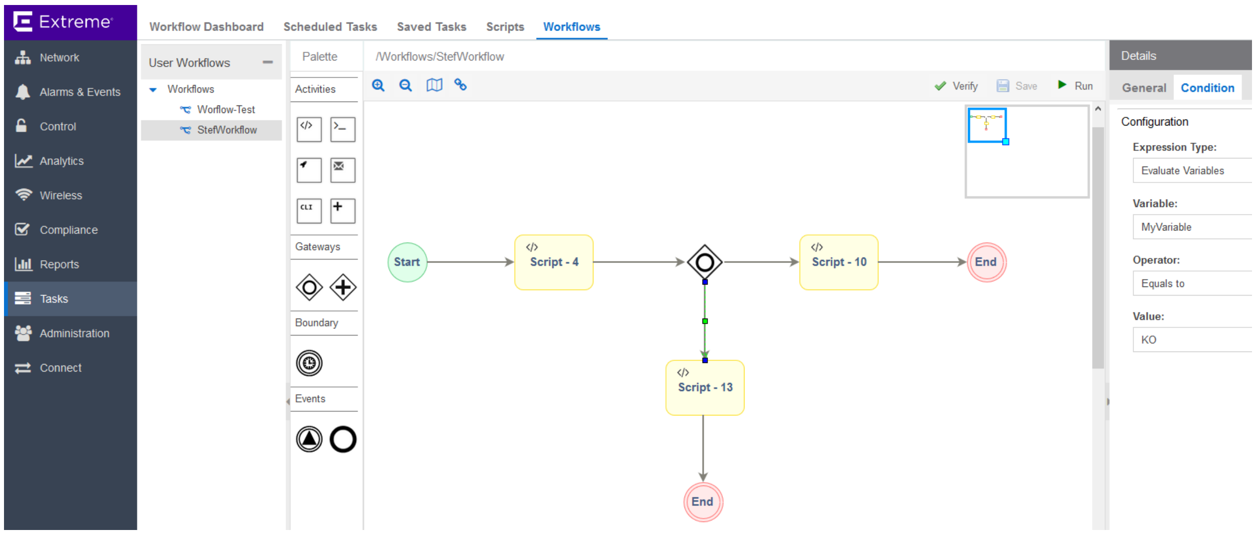
To evaluate the value of MyVariable, first change its scope. In the previous example, this variable was defined with a scope of Activity. This means its value is not accessible outside of the activity. To check it with the inclusive parallel gateway, you need to increase its scope by setting it to Workflow.



Name	Default V...	Variable ...	Scope	Type	Referenc...
workflowN...			Workflow	String	true
workflowD...			Workflow	String	true
workflowP...			Workflow	String	true
workflowT...			Workflow	Number	true
workflowC...			Workflow	String	true
workflowU...			Workflow	String	true
workflowC...			Workflow	Number	true
workflowU...			Workflow	Number	true
workflowV...			Workflow	Number	true
workflowC...			Workflow	String	true
workflowN...			Workflow	String	true
workflowS...			Workflow	String	true
workflow...			Workflow	String	true
MyVariable	extreme		Workflow	String	true

After you have done this, the variable is accessible throughout the workflow. To test it with the inclusive parallel gateway, select one of the output links and set the condition. This dictates what must be met to follow this path.

In your example, you defined two paths; one that checks if “My Variable” is equal to “OK”, and the other to “KO”. The resulting script prints the path has been followed.



Configuration

Expression Type: Evaluate Variables

Variable: MyVariable

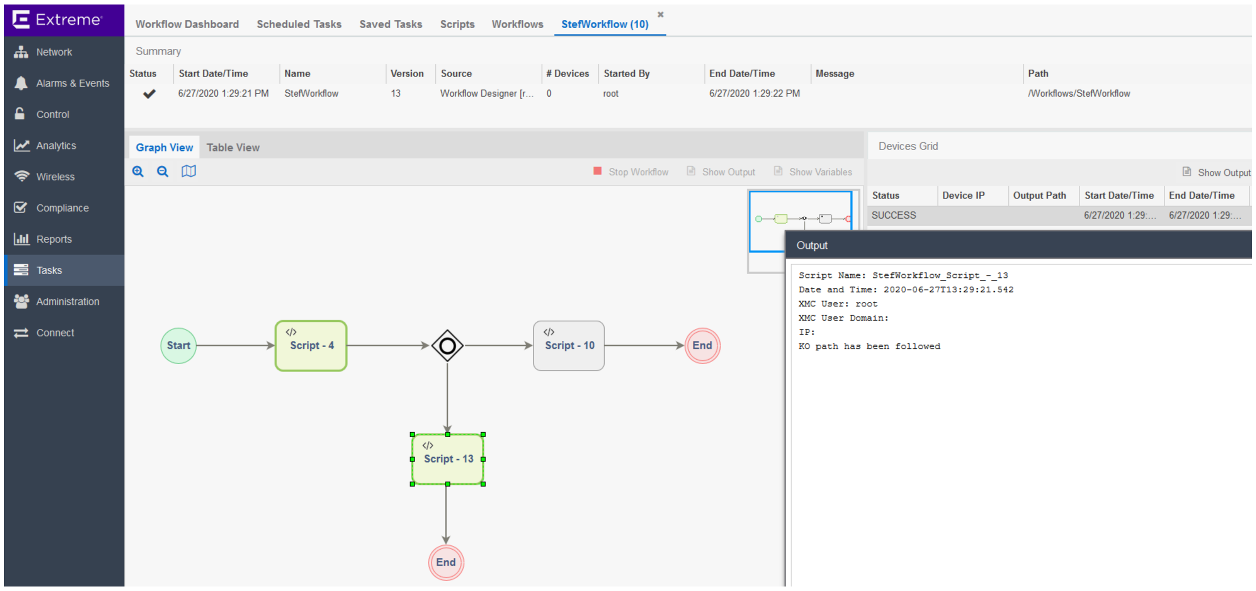
Operator: Equals to

Value: KO

The first script (Script-4) sets the value to the variable for the rest of the workflow, using emc_results.put().

```
print emc_vars['MyVariable']

emc_results.put("MyVariable", "KO")
```

After you save and run your workflow, you can watch a visual representation of execution of the workflow. Every step that is completed successfully turns green, while failed steps appear red.



Summary

Status	Start Date/Time	Name	Version	Source	# Devices	Started By	End Date/Time	Message	Path
✓	6/27/2020 1:29:21 PM	StefWorkflow	13	Workflow Designer [r...	0	root	6/27/2020 1:29:22 PM		/Workflows/StefWorkflow

Devices Grid

Status	Device IP	Output Path	Start Date/Time	End Date/Time
SUCCESS			6/27/2020 1:29:...	6/27/2020 1:29:...

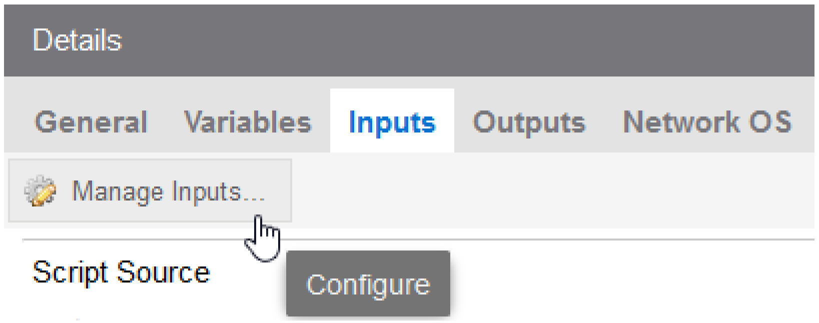
Output

```
Script Name: StefWorkflow_Script_-_13
Date and Time: 2020-06-27T13:29:21.542
XMC User: root
XMC User Domain:
IP:
KO path has been followed
```

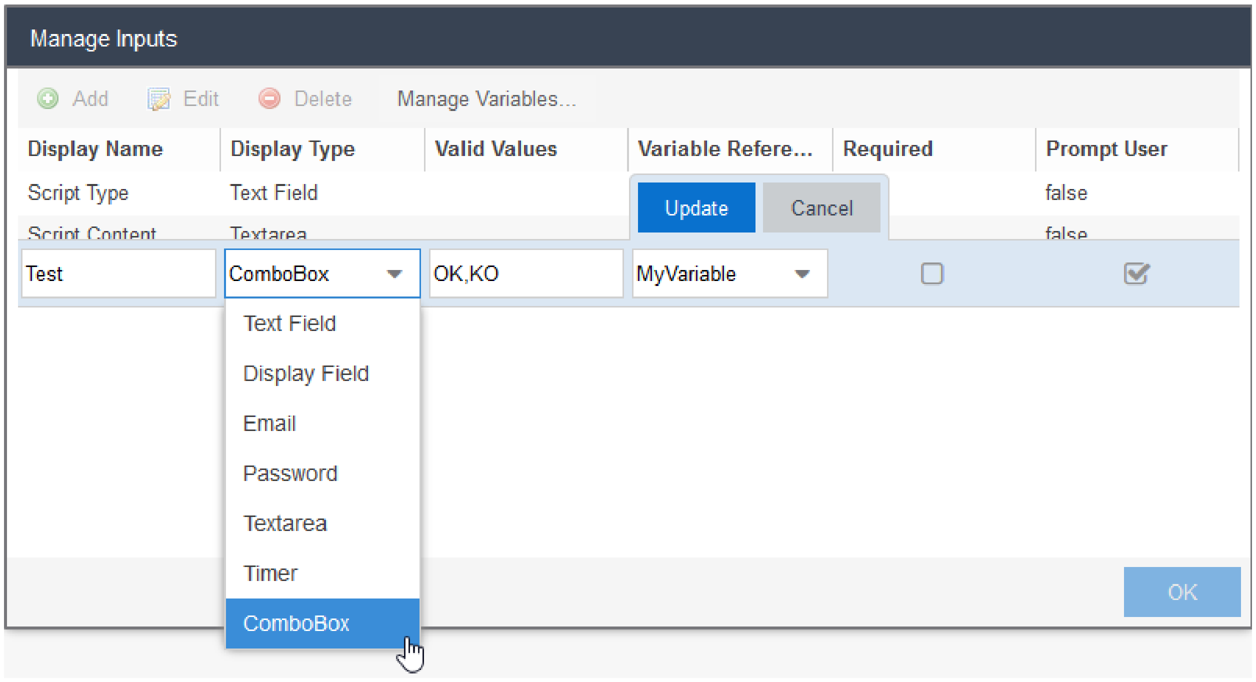
5.2.5 Add User Inputs

Workflows also allows for users to be prompted for inputs. In the previous example, you replaced the hardcoded value of My Variable with a field to select the value you want to use and made sure only valid values are available.

Select your script (Script-4) and select the Inputs tab in the Details panel. At the top of the Inputs tab, select the gear icon (Manage Inputs…).



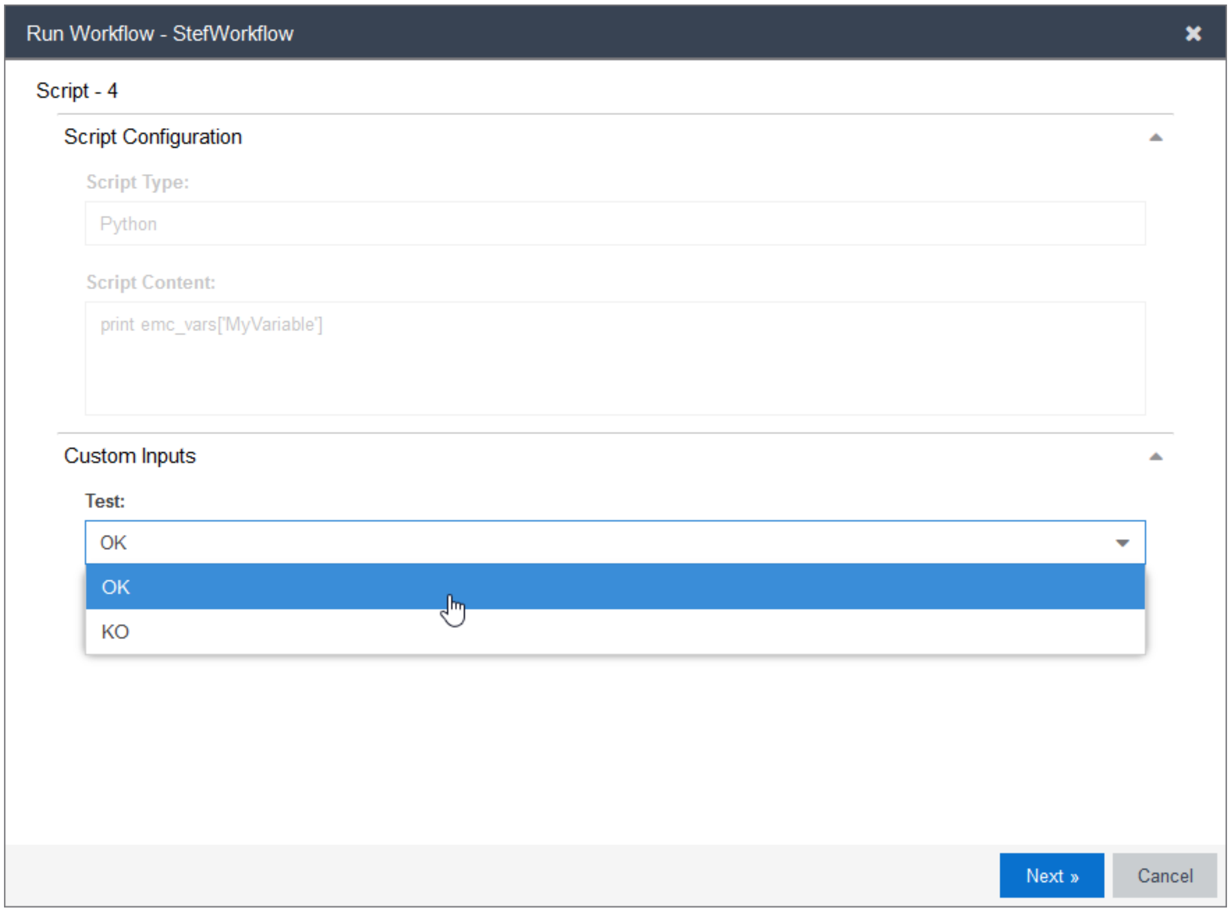
Create a new Input by setting a name, a type, the valid values to be entered and the variable that will be set to this value. You can also choose to make this input required and prompt the user for it.



Display Name	Display Type	Valid Values	Variable Refere...	Required	Prompt User
Script Type	Text Field				false
Script Content	Textarea				false
Test	ComboBox	OK,KO	MyVariable	☐	✓

Select a ComboBox with Valid Values of OK and KO. This is what you expect for your inclusive parallel gateway. Then select the correct variable (My Variable). Your script will no longer modify the variable.

When you run your workflow, you must choose between OK and KO. The path that is followed depends on your choice.



Script Type: Python

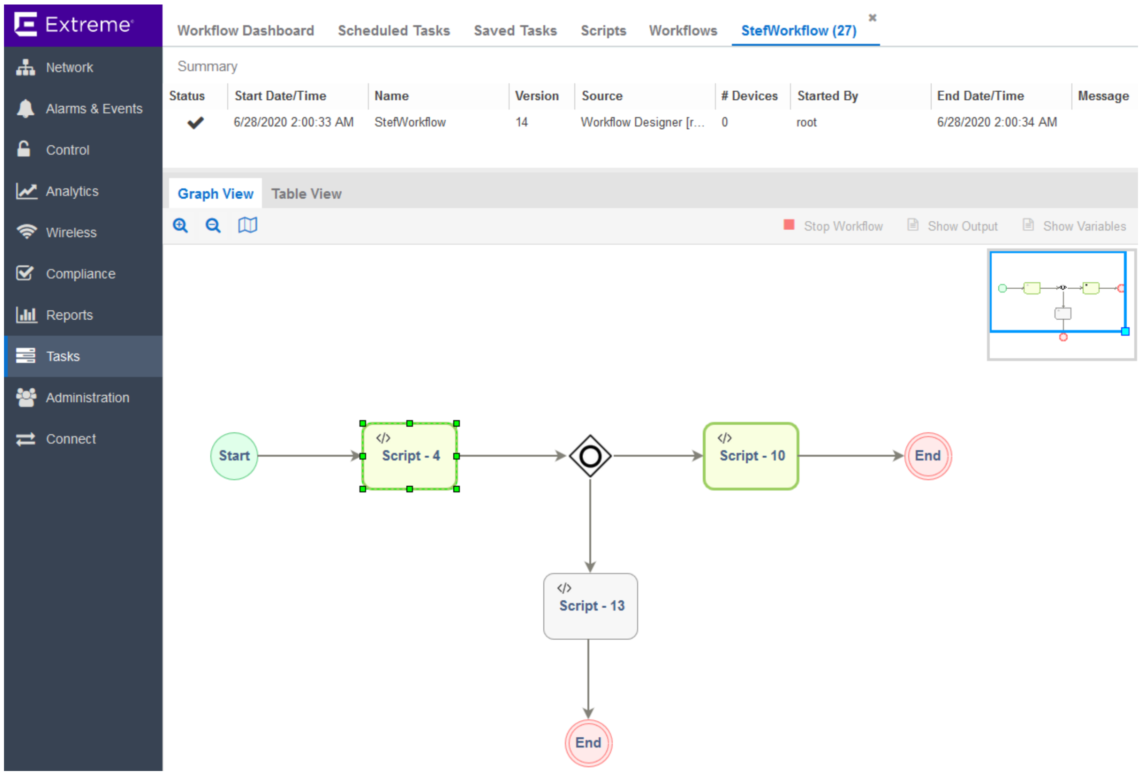
Script Content:

```
print emc_vars['MyVariable']
```

Test: OK

- OK
- KO

The result:



Summary

Status	Start Date/Time	Name	Version	Source	# Devices	Started By	End Date/Time	Message
✓	6/28/2020 2:00:33 AM	StefWorkflow	14	Workflow Designer [r...	0	root	6/28/2020 2:00:34 AM	

5.2.6 Automate Workflow Execution

As with Python scripting with EXOS, automation is important because it provides dynamic triggers, based on specific events, in a workflow.

Workflows in XMC can be triggered by multiple events, such as:

- Alarms
- API CALLs
- ExtremeControl
- Scheduler
- Manually by right-clicking on a port, a device, or a group of devices

You can find a description of how a workflow is executed in the workflow, if you are using a script, by accessing the emc_vars Python dictionary.

For example, using XMC 8.4.4, the list shown below displays the keys in the emc_vars Python dictionary when a workflow is triggered by an Alarm, compared to a script in a workflow:

```
There are 92 entries in emc_vars in Alarm Workflow Engine
There are 72 entries in emc_vars in Workflow Engine

emc_vars not in Workflow Engine:
Not found: deviceIp
Not found: eventCategory
Not found: sysDescr
Not found: chassisId
Not found: alarmSource
Not found: sysName
Not found: deviceFirmware
Not found: deviceIpCtx
Not found: eventClient
Not found: alarmName
Not found: eventType
Not found: sysUpTime
Not found: deviceStatus
Not found: eventTitle
Not found: deviceBootProm
Not found: alarmId
Not found: eventUser
Not found: severity
Not found: eventSeverity
Not found: deviceNickName
Not found: message
Not found: chassisType
Not found: sysContact

emc_vars not in Alarm Workflow Engine:
Not found: hostName
Not found: extreme.hideLegacyDesktopApps
Not found: ports
```

In this example, the Alarm context adds context-specific variables in the emc_vars dictionary. Notice that there is also a `deviceIp` variable that appears beside the usual `deviceIP` variable.

This section describes a simple workflow to show how to configure an Alarm that can execute a specific workflow, using the new variables.

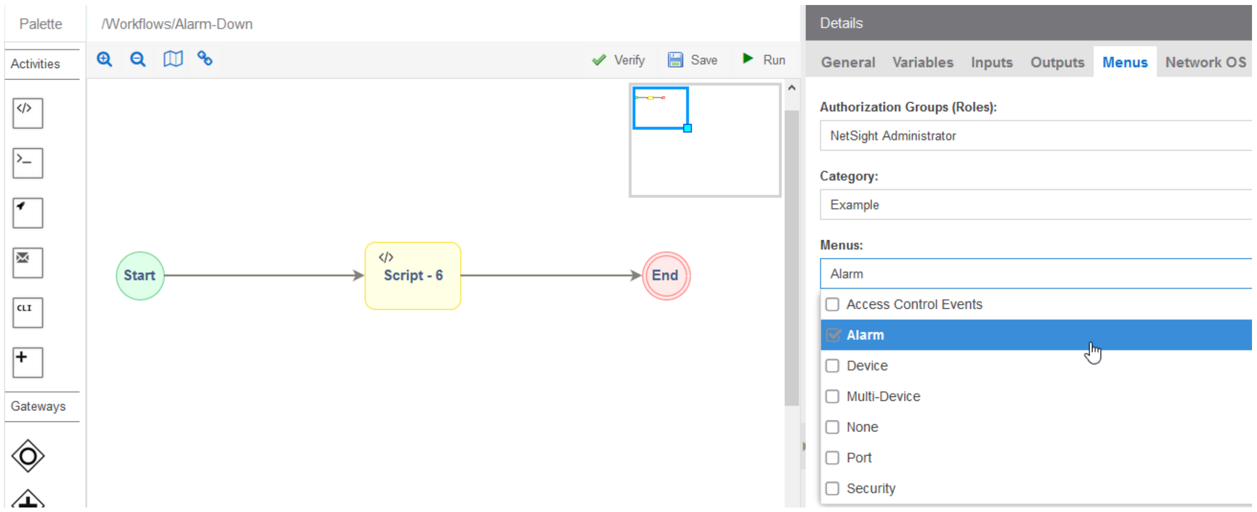
Start with a regular workflow, and run a Python script when an alarm for a Device Down appears in XMC. This script is not especially useful, but it illustrates the framework.

The script you run first checks to make sure it is triggered by the Alarms and Events process and that it can detect specific device types.

```
# You want to be notified by this workflow only if specific network devices are down
# You could also look at the model type to narrow even further the worflow
```

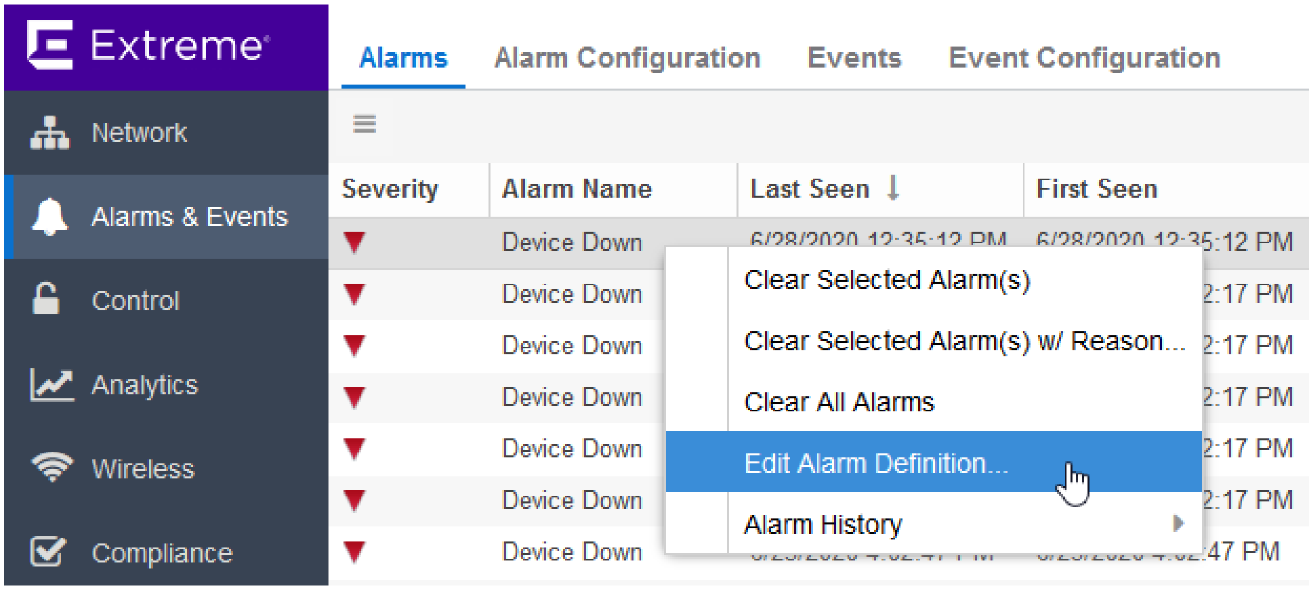
```
if emc_vars['family'] not in ["VSP Series", "Summit Series"]:
    print "This type of device is not supported by this workflow ({})".format(emc_var
s['family'])
    exit(0)

# This is one way to make sure the workflow is triggered from alarm
# This key is alarm context-specific
if 'alarmName' not in emc_vars:
    print "This workflow must be executed within an Alarm"
    exit(0)

# A not very useful script, just for illustration
print "You received an alarm that device {}, with the IP {} is {}.".format(emc_vars['
deviceNickName'], emc_vars['deviceIp'], emc_vars['message'])
```

After you have created this workflow, configure the rights and menus where this workflow can be executed in the Details panel, in the Menus tab of the global workflow. Make sure no activities are selected.



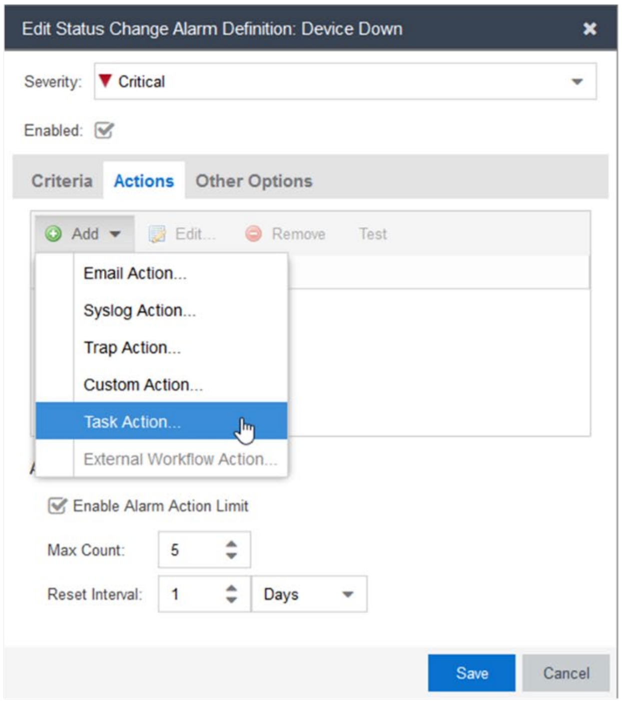
Authorization Groups (Roles): NetSight Administrator
Category: Example
Menus: Alarm
[✓ Alarm]

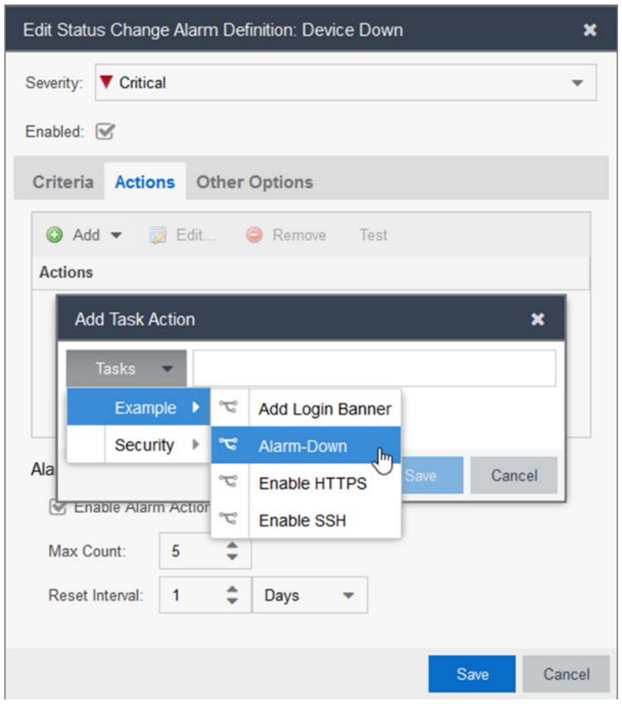
Next, select the Alarm menu. From the Alarms & Events XMC menu, select the alarm type you want to modify. You can create a new alarm or edit an existing one, depending on the use case. In this example, you will edit the existing Device Down alarm.



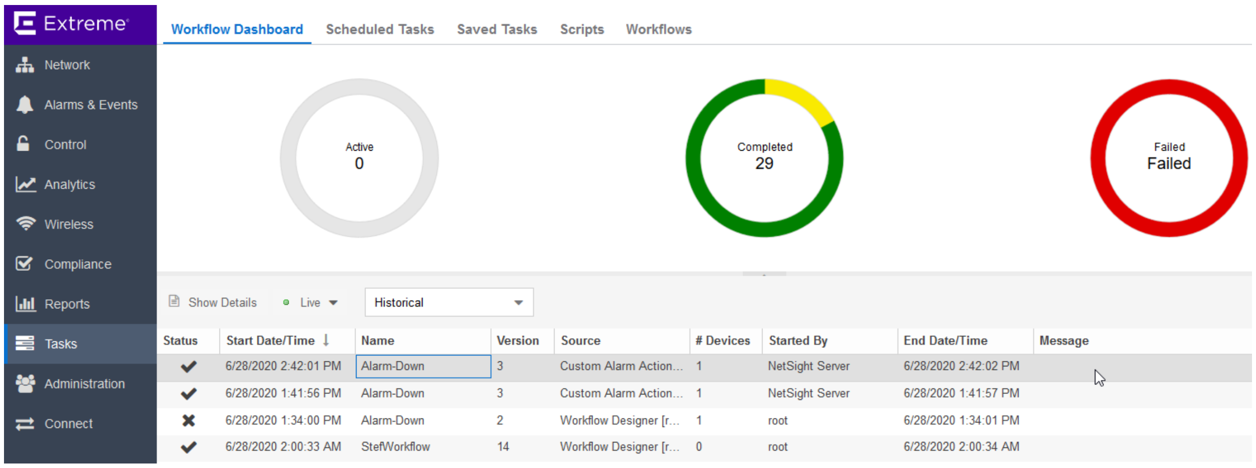
Severity	Alarm Name	Last Seen	First Seen
▼	Device Down	6/28/2020 12:35:12 PM	6/28/2020 12:35:12 PM
▼	Device Down		2:17 PM
▼	Device Down		2:17 PM
▼	Device Down		2:17 PM
▼	Device Down		2:17 PM
▼	Device Down		2:17 PM
▼	Device Down		:47 PM

From the Actions tab, add a new Task Action and select your workflow.





Select Save. As soon as a Device Down alarm is received by XMC, your workflow is executed. You can track it from the Workflow Dashboard.



Status	Start Date/Time	Name	Version	Source	# Devices	Started By	End Date/Time	Message
✓	6/28/2020 2:42:01 PM	Alarm-Down	3	Custom Alarm Action...	1	NetSight Server	6/28/2020 2:42:02 PM	
✓	6/28/2020 1:41:56 PM	Alarm-Down	3	Custom Alarm Action...	1	NetSight Server	6/28/2020 1:41:57 PM	
✗	6/28/2020 1:34:00 PM	Alarm-Down	2	Workflow Designer [r...	1	root	6/28/2020 1:34:01 PM	
✓	6/28/2020 2:00:33 AM	StefWorkflow	14	Workflow Designer [r...	0	root	6/28/2020 2:00:34 AM	

Double-click the workflow to see details and confirm the output of your script.

```
Script Name: Alarm-Down_Script_-_6
Date and Time: 2020-06-28T15:01:56.478
```

```
XMC User: NetSight Server
XMC User Domain:
IP: 192.168.56.121
You received an alarm that device IP Campus_08:00:27:35:2A:E4, with the IP
192.168.56.121 is SNMP Contact Lost: No SNMP reply from device 192.168.56.121
caused by SNMP Error: Timeout[4098], last uptime was 0 Days 00:02:56.
```

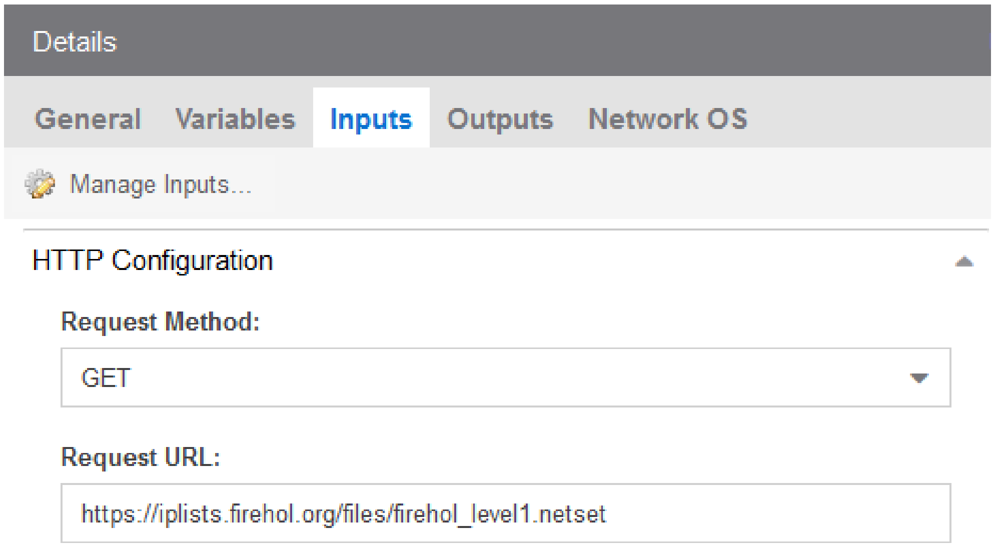
5.2.7 Workflow Example

In this example, you will create a new workflow, using different resources for illustration. You will retrieve data from a web server and use it in a script.

To see a list of blocked IP addresses, go to this link:

HTTPS://iplists.firehol.org/files/firehol_level1.netset

You will use this for your workflow, using the HTTP Activity. You must do a GET on this URL.



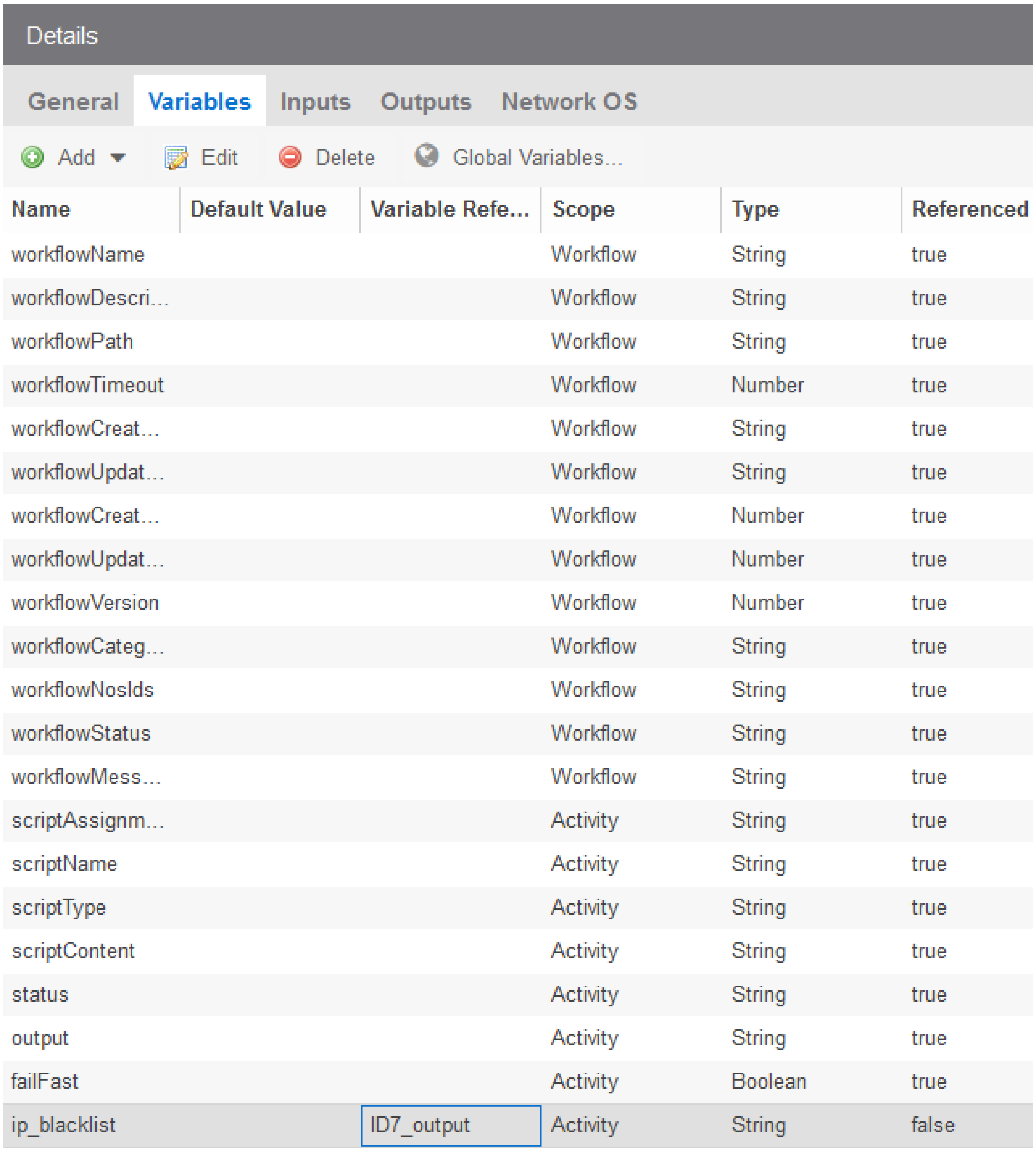
Request Method: GET

Request URL: https://iplists.firehol.org/files/firehol_level1.netset

To use the content returned by this URL, in the script following the HTTP activity, define a new variable and set the “Variable Reference” field to the output of the ID of the HTTP Activity.

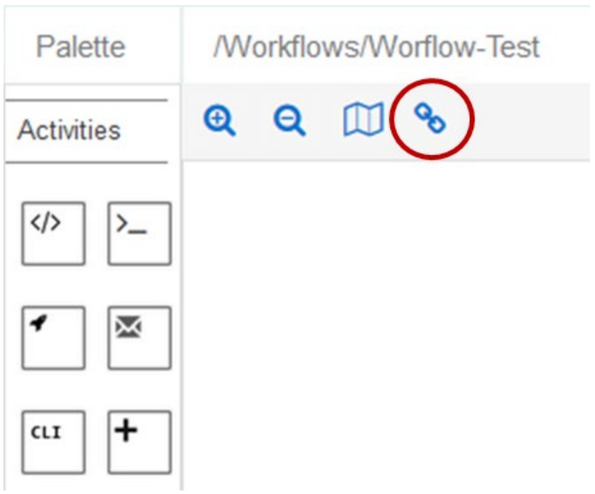
***Note**: This ID can be found in the General tab of an activity, in is the “custom ID” field.*

Limit the scope of the new variable to the activity since you will not be using it elsewhere.

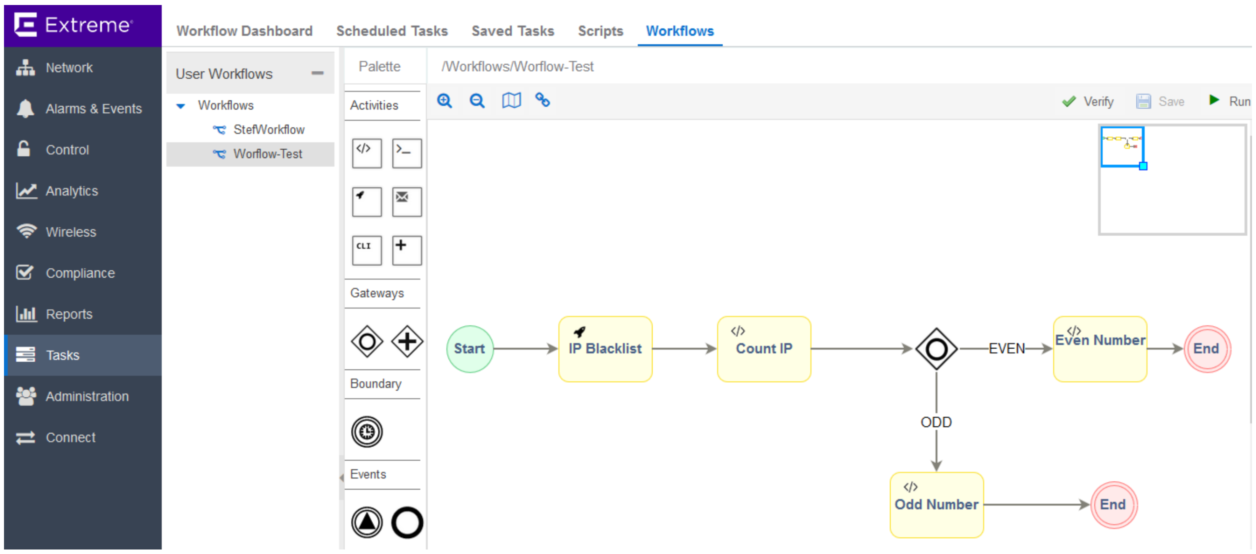


Name	Default Value	Variable Refe...	Scope	Type	Referenced
workflowName			Workflow	String	true
workflowDescri...			Workflow	String	true
workflowPath			Workflow	String	true
workflowTimeout			Workflow	Number	true
workflowCreat...			Workflow	String	true
workflowUpdat...			Workflow	String	true
workflowCreat...			Workflow	Number	true
workflowUpdat...			Workflow	Number	true
workflowVersion			Workflow	Number	true
workflowCateg...			Workflow	String	true
workflowNosIds			Workflow	String	true
workflowStatus			Workflow	String	true
workflowMess...			Workflow	String	true
scriptAssignm...			Activity	String	true
scriptName			Activity	String	true
scriptType			Activity	String	true
scriptContent			Activity	String	true
status			Activity	String	true
output			Activity	String	true
failFast			Activity	Boolean	true
ip_blacklist		ID7_output	Activity	String	false

Your script counts the number of IPs and, based on this, will trigger a different script. To make the workflow easier to read, enable the link edit mode to you can add a description to each test.



Your workflow should look like this:

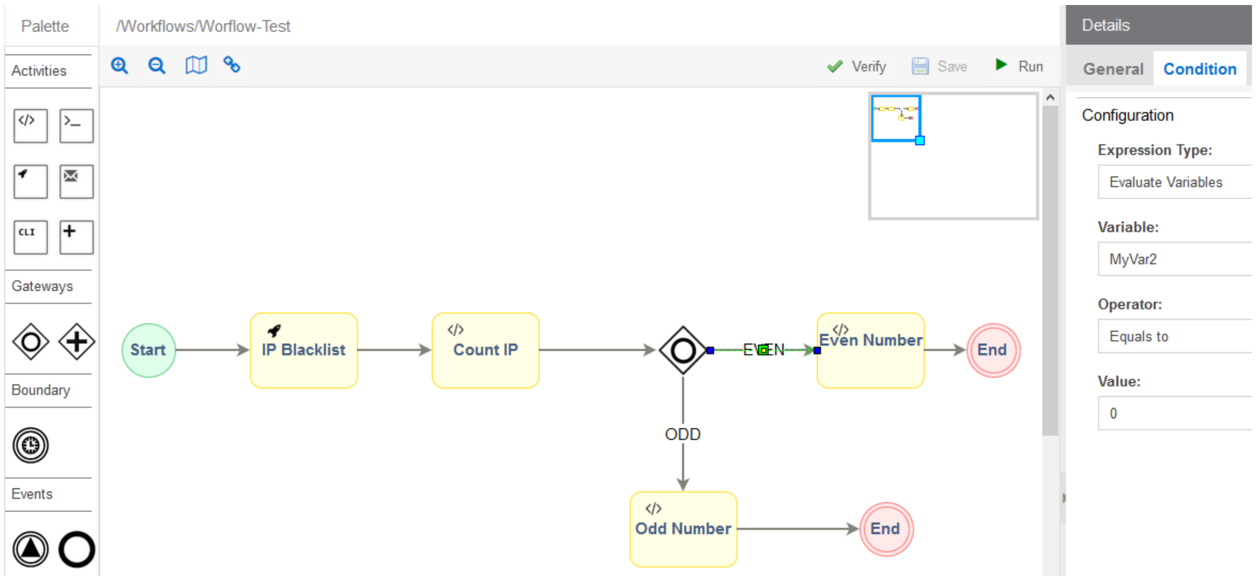


The simple Count IP script counts all the IPs in the file received, and removes the comment lines.

Create a second variable called MyVar2 to test for path selection.

```
received_blacklist = emc_vars['ip_blacklist']

blist = received_blacklist.splitlines()

i = 0

for ip in blist:
    if ip.startswith("#"):
        continue
    else:
        i += 1

if (i % 2):
    emc_results.put("MyVar2", "1")
else:
    emc_results.put("MyVar2", "0")

emc_results.put("IPCount", str(i))
```

The inclusive parallel gateway tests if the MyVar2 variable is equal to 0 or 1.



Configuration

Expression Type: Evaluate Variables

Variable: MyVar2

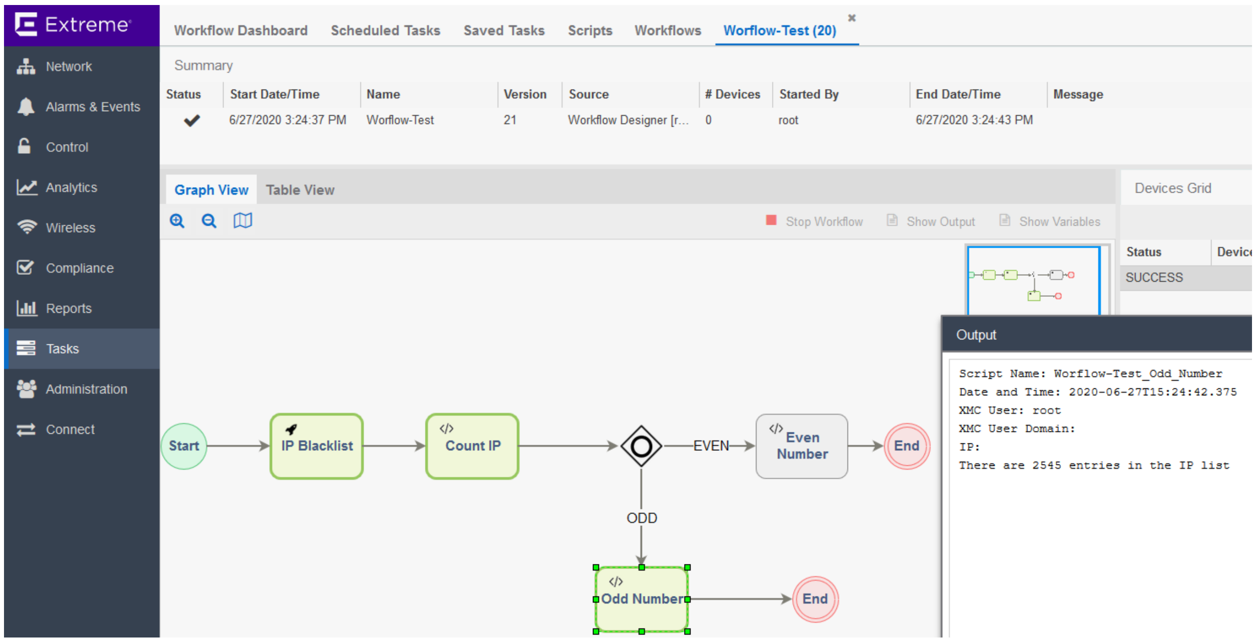
Operator: Equals to

Value: 0

The final scripts do the same thing, but for the purpose of this example, this is duplicated for each path.

```
print "There are {} entries in the IP list".format(emc_vars["IPCount"])
```

Run your workflow to see the output:



Summary

Status	Start Date/Time	Name	Version	Source	# Devices	Started By	End Date/Time	Message
✓	6/27/2020 3:24:37 PM	Worflow-Test	21	Workflow Designer [r...	0	root	6/27/2020 3:24:43 PM	

Status	Devic
SUCCESS	

Output

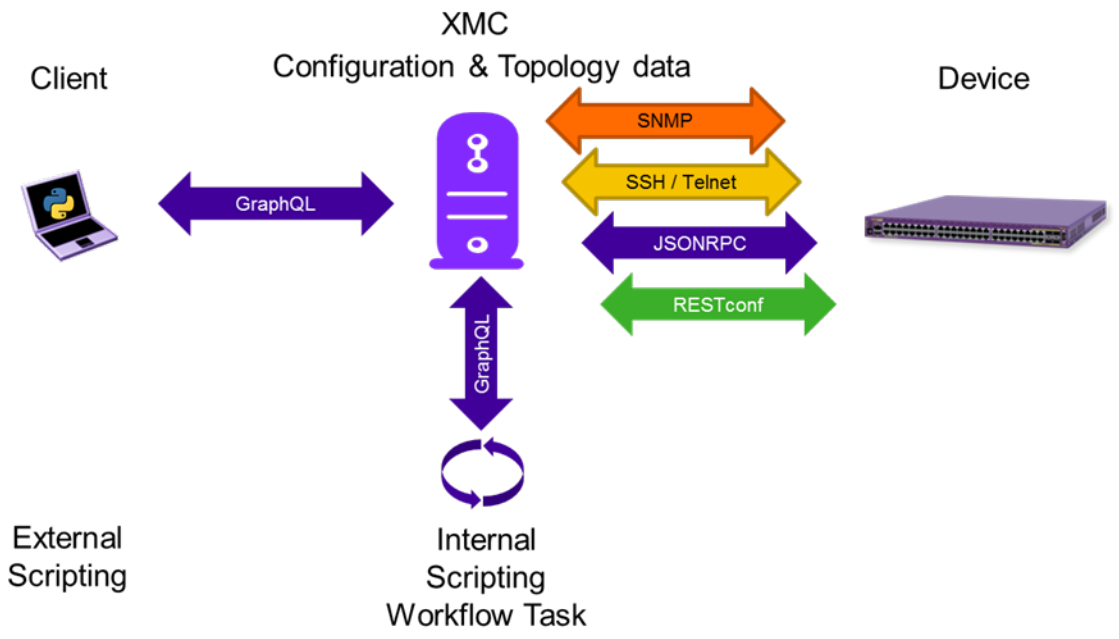
```
Script Name: Worflow-Test_Odd_Number
Date and Time: 2020-06-27T15:24:42.375
XMC User: root
XMC User Domain:
IP:
There are 2545 entries in the IP list
```

5.3 NorthBound Interface API

The NBI API is based on GraphQL. Through the NBI, you can access data stored in the database of XMC, which means virtually everything that XMC manages in the network.

XMC provides the emc_nbi Python object to interact with the NBI API.

This NBI is accessible both internally through the Python Scripting Engine (and by extension the Workflow Engine), and externally to any authorized application requesting data.



GraphQL is a query language that allows for extremely efficient data transfers where only the necessary information is transmitted, unlike plain JSON. You can both read and write data from and to the database. In GraphQL vocabulary, a *read* is a query and a *write* is a mutation.

This chapter explores in detail the GraphQL capabilities.

5.3.1 emc_nbi

The emc_nbi Python object includes several functions and methods. As of XMC 8.4.4, the example below shows what is available:

```
print dir(emc_nbi)
```

The output is:

```
['__class__', '__copy__', '__deepcopy__', '__delattr__', '__doc__',
'__ensure_finalizer__', '__eq__', '__format__', '__getattribute__',
'__hash__', '__init__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__str__', '__subclasshook__', '__unicode__',
'class', 'equals', 'example1', 'example2', 'getClass', 'getName', 'hashCode',
'mutation', 'name', 'notify', 'notifyAll', 'query', 'toString', 'wait']
```

The two methods that you will use the most are query and mutation.

***Note**: It is possible to use the query method for mutation, but you will use each method for their respective usage for clarity. However, only the query method supports graphql variables, as presented here: HTTPS://graphql.org/learn/queries/#variables.*

Use query to access the XMC database as read-only, and mutation to edit (write) to the database.

***Note**: When using mutation, be careful not to corrupt the XMC database. The best practice is to always have a backup of your database.*

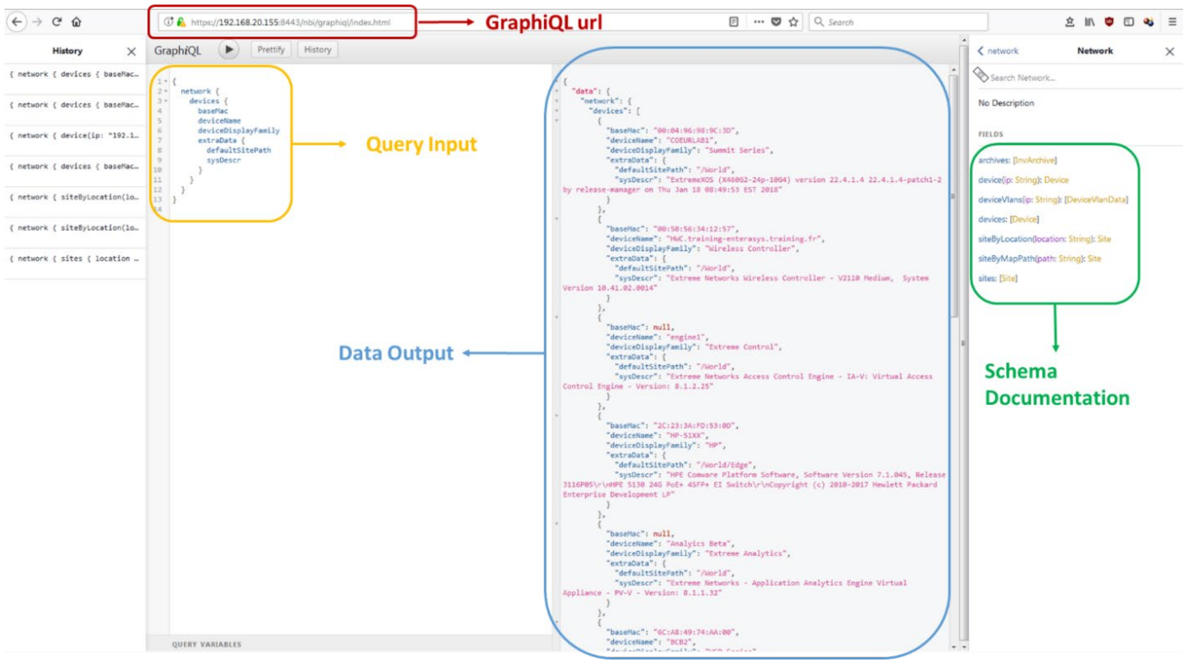
5.3.2 GraphQL Query

XMC integrates GraphQL, an interface that lets you test queries and view the output. You can access GraphQL at the following URL on a given XMC server:

HTTPS://<xmc server IP>:8443/nbi/graphiql/index.html

You use the GraphQL NBI to create queries that retrieve information.

***Note**: The GraphiQL interface can also be accessed from XMC, in the following menu: Administration > Diagnostics > Server > Server Utilities > NBI Explorer.*



```
{ network { devices { baseMac…
{ network { devices { baseMac…
{ network { device(ip: "192.1…
{ network { devices { baseMac…
{ network { siteByLocation(lo…
{ network { siteByLocation(lo…
{ network { sites { location …
```

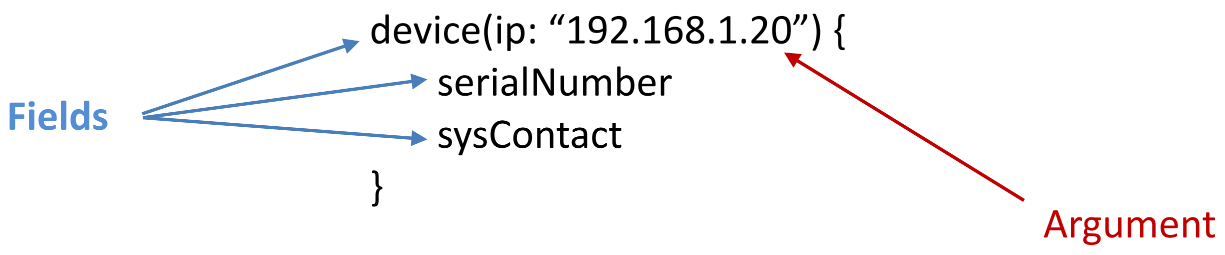
```
{
  network {
    devices {
      baseMac
      deviceName
      deviceDisplayFamily
      extraData {
        defaultSitePath
        sysDescr
      }
    }
  }
}
```

```
{
  "data": {
    "network": {
      "devices": [
        {
          "baseMac": "00:04:96:98:9C:3D",
          "deviceName": "COEURLAB1",
          "deviceDisplayFamily": "Summit Series",
          "extraData": {
            "defaultSitePath": "/World",
            "sysDescr": "ExtremeXOS (X460G2-24p-10G4) version 22.4.1.4 22.4.1.4-patch1-2
by release-manager on Thu Jan 18 08:49:53 EST 2018"
          }
        },
        {
          "baseMac": "00:50:56:34:12:57",
          "deviceName": "HWC.training-enterasys.training.fr",
          "deviceDisplayFamily": "Wireless Controller",
          "extraData": {
            "defaultSitePath": "/World",
            "sysDescr": "Extreme Networks Wireless Controller - V2110 Medium,  System
Version 10.41.02.0014"
          }
        },
        {
          "baseMac": null,
          "deviceName": "engine1",
          "deviceDisplayFamily": "Extreme Control",
          "extraData": {
            "defaultSitePath": "/World",
            "sysDescr": "Extreme Networks Access Control Engine - IA-V: Virtual Access
Control Engine - Version: 8.1.2.25"
          }
        },
        {
          "baseMac": "2C:23:3A:FD:53:0D",
          "deviceName": "HP-51XX",
          "deviceDisplayFamily": "HP",
          "extraData": {
            "defaultSitePath": "/World/Edge",
            "sysDescr": "HPE Comware Platform Software, Software Version 7.1.045, Release
3116P05\r\nHPE 5130 24G PoE+ 4SFP+ EI Switch\r\nCopyright (c) 2010-2017 Hewlett Packard
Enterprise Development LP"
          }
        },
        {
          "baseMac": null,
          "deviceName": "Analyics Beta",
          "deviceDisplayFamily": "Extreme Analytics",
          "extraData": {
            "defaultSitePath": "/World",
            "sysDescr": "Extreme Networks - Application Analytics Engine Virtual
Appliance - PV-V - Version: 8.1.1.32"
          }
        },
        {
          "baseMac": "6C:A8:49:74:AA:00",
          "deviceName": "BCB2",
```

```
archives: [InvArchive]
device(ip: String): Device
deviceVlans(ip: String): [DeviceVlanData]
devices: [Device]
siteByLocation(location: String): Site
siteByMapPath(path: String): Site
sites: [Site]
```

A query is a read-only operation. This has been supported since XMC 8.1.2 as a beta feature and in 8.2 as GA code.

A query is a string that can be formatted as a JSON object. The most important part of a query are the fields. Each field is defined in a schema that is dynamically created by the runtime. Some arguments may be used with some fields.



```
device(ip: “192.168.1.20”) {
    serialNumber
    sysContact
}
```

The top-level field stipulates a query or a mutation. The sub-systems supported are:

```
accessControl,
administration,
inventory,
network,
policy,
wireless,
workflows
```

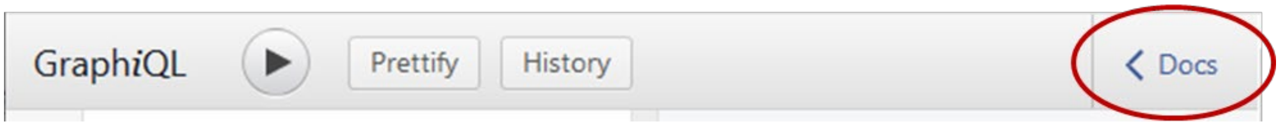
You can access the GraphQL schema description as an IDL file or a JSON file, at the following URLs on an XMC server:

HTTPS://<xmc-ip-address>:8443/nbi/graphql/schema.idl

HTTPS://<xmc-ip-address>:8443/nbi/graphql/schema.json

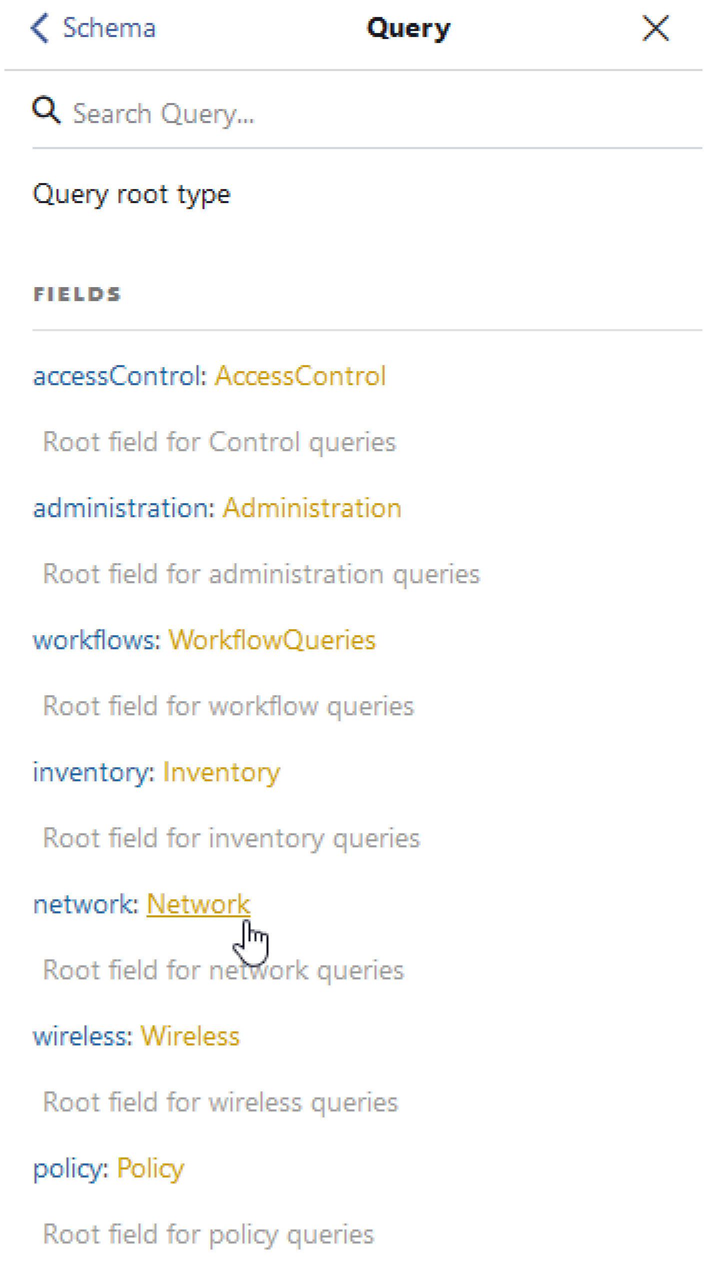
Using the GraphiQL interface, you can browse the fields available to us. Let's have an example and say you want to retrieve the list of devices managed by XMC, and have their MAC address, firmware version and site location.

By expanding the Docs panel (on the far right), you have access to all the information available.



Here you can find available fields, field types and sub-fields. Typically, the field type is on the left (in blue) and the field value is on the right (in orange).

Start at the top with the Query field, then browse the sub-fields to find the Network section.



Query root type

FIELDS

accessControl: AccessControl

Root field for Control queries

administration: Administration

Root field for administration queries

workflows: WorkflowQueries

Root field for workflow queries

inventory: Inventory

Root field for inventory queries

network: Network

Root field for network queries

wireless: Wireless

Root field for wireless queries

policy: Policy

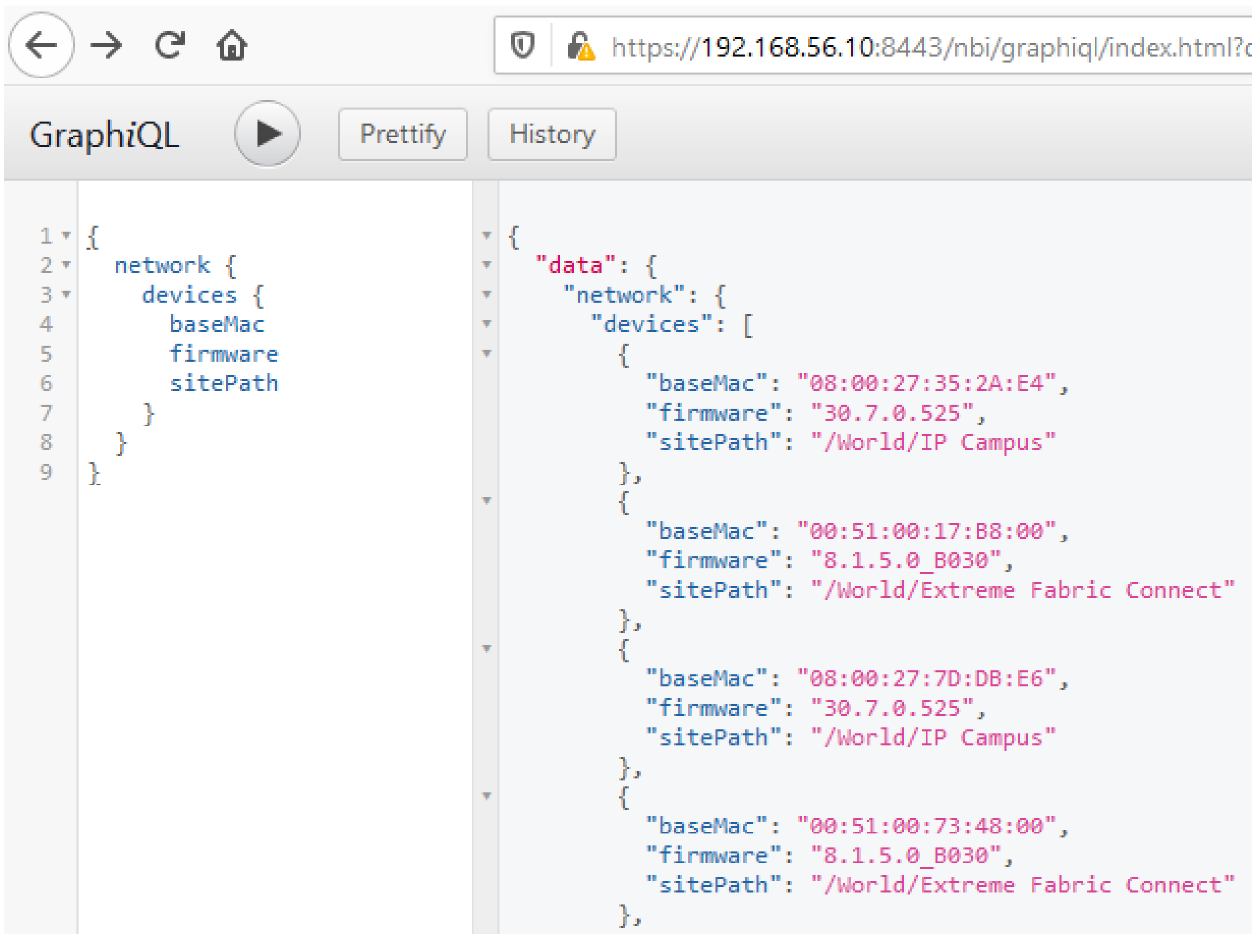
Root field for policy queries

The Device field contains several sub-fields. Build your query in the Network input panel and include the fields you want to search between curly brackets. Some fields are mandatory, which is indicated by a !.

A query example is shown here:

```
{
  network {
    devices {
      baseMac
      firmware
      sitePath
    }
  }
}
```

If you run your query from GraphiQL, you will see this (truncated) output from your XMC server:



```
{
  network {
    devices {
      baseMac
      firmware
      sitePath
    }
  }
}
```

```
{
  "data": {
    "network": {
      "devices": [
        {
          "baseMac": "08:00:27:35:2A:E4",
          "firmware": "30.7.0.525",
          "sitePath": "/World/IP Campus"
        },
        {
          "baseMac": "00:51:00:17:B8:00",
          "firmware": "8.1.5.0_B030",
          "sitePath": "/World/Extreme Fabric Connect"
        },
        {
          "baseMac": "08:00:27:7D:DB:E6",
          "firmware": "30.7.0.525",
          "sitePath": "/World/IP Campus"
        },
        {
          "baseMac": "00:51:00:73:48:00",
          "firmware": "8.1.5.0_B030",
          "sitePath": "/World/Extreme Fabric Connect"
        },
```

***Note**: The list for this device consists of VMs running on a PC, with XMC also running on the same PC along with a VM. This is a practical and safe way to test queries, workflows, and scripts.*

This example clearly shows the GraphQL query syntax and the expected output to expect. The output is formatted in JSON in GraphiQL, but the data is returned as a Java hashmap within a script. This should be treated as a regular Python dictionary. From here, you have all the tools you need to include NBI CALLs within your Python scripts and workflows.

Using this same query example in a Python script is easy:

```
query = '''
{
  network {
    devices {
      baseMac
      firmware
      sitePath
    }
  }
}
'''

res = emc_nbi.query(query)

i = 0
j = 0
for item in res['network']['devices']:
    if item['sitePath'] == "/World/Extreme Fabric Connect":
        i += 1
    elif item['sitePath'] == "/World/IP Campus":
        j += 1
    else:
        continue

print "There are {} devices in Extreme Fabric Connect site".format(i)
print "There are {} devices in IP Campus site".format(j)
```

Copy and paste your query from GraphiQL as a string in your Python code, and then CALL it using the provided emc_nbi Python object, with the query method.

The hashmap data format is natively converted to a Python dictionary. This data should not be treated as JSON, which will trigger an error.

The output is:

```
Script Name: NBI Test
Date and Time: 2020-06-27T17:56:20.166
```

```
XMC User: root
XMC User Domain:
IP: 192.168.56.121
There are 5 devices in Extreme Fabric Connect site
There are 9 devices in IP Campus site
```

5.3.3 GraphQL Mutation

A mutation lets you add, delete, or modify content in the XMC database. This can be potentially risky, so be sure you are not deleting important data, or corrupting the database content.

***Note**: GraphQL mutation is supported in XMC 8.3 and later.*

The mutation framework is the same as for query: browse the GraphiQL interface to find the format of your CALL, and the fields to complete.

When you created a query, you did not specify the top query field in your CALL, which is the default and as such implied. For mutation, however, you must start your CALL with the top mutation field.

To illustrate mutation, create a Python script that creates a new site. Because this is a simple script is for demonstration, it is short. A real script and workflow would be much longer.

```
SiteName = "/World/NewSite"

mutation = '''
  mutation {
    network {
      createSite(input: {siteLocation: "%s"}) {
        status
        message
        siteId
      }
    }
  }
''' % SiteName

res = emc_nbi.mutation(mutation)

print "\nData returned: ", res

if res['network']['createSite']['status'] != "SUCCESS":
    print "\nCannot create Site {} because {}".format(
        SiteName,
        res['network']['createSite']['message'])
```

```
else:
    print "\nSuccessfully created Site {}".format(SiteName)
```

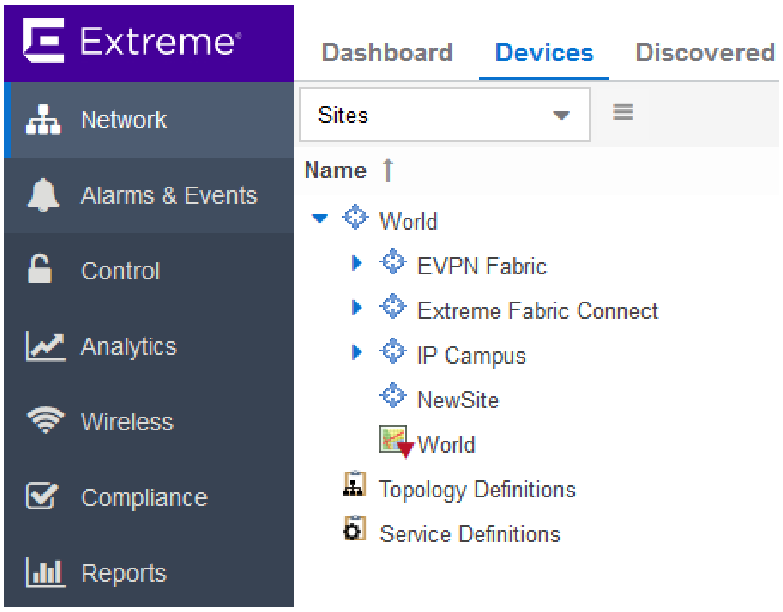
The string starts with the mutation field. Browse the GraphiQL interface to the createSite field. Enter the required arguments, such as the siteLocation (which is mandatory when creating a new site).

Specify the output you want to receive from this action. Select from the GraphiQL list.

Print the result to display the data returned, and print a comment depending on the outcome.

If you run this script before the new site exists, you can also confirm that the site has been created.

```
Script Name: NBI Mutation
Date and Time: 2020-06-28T16:00:20.726
XMC User: root
XMC User Domain:
IP: 192.168.56.122

Data returned:  {network: {createSite={status=SUCCESS, message=, siteId=10}}}

Successfully created Site /World/NewSite
```



When the site exists, if you run the script again, you will see an error:

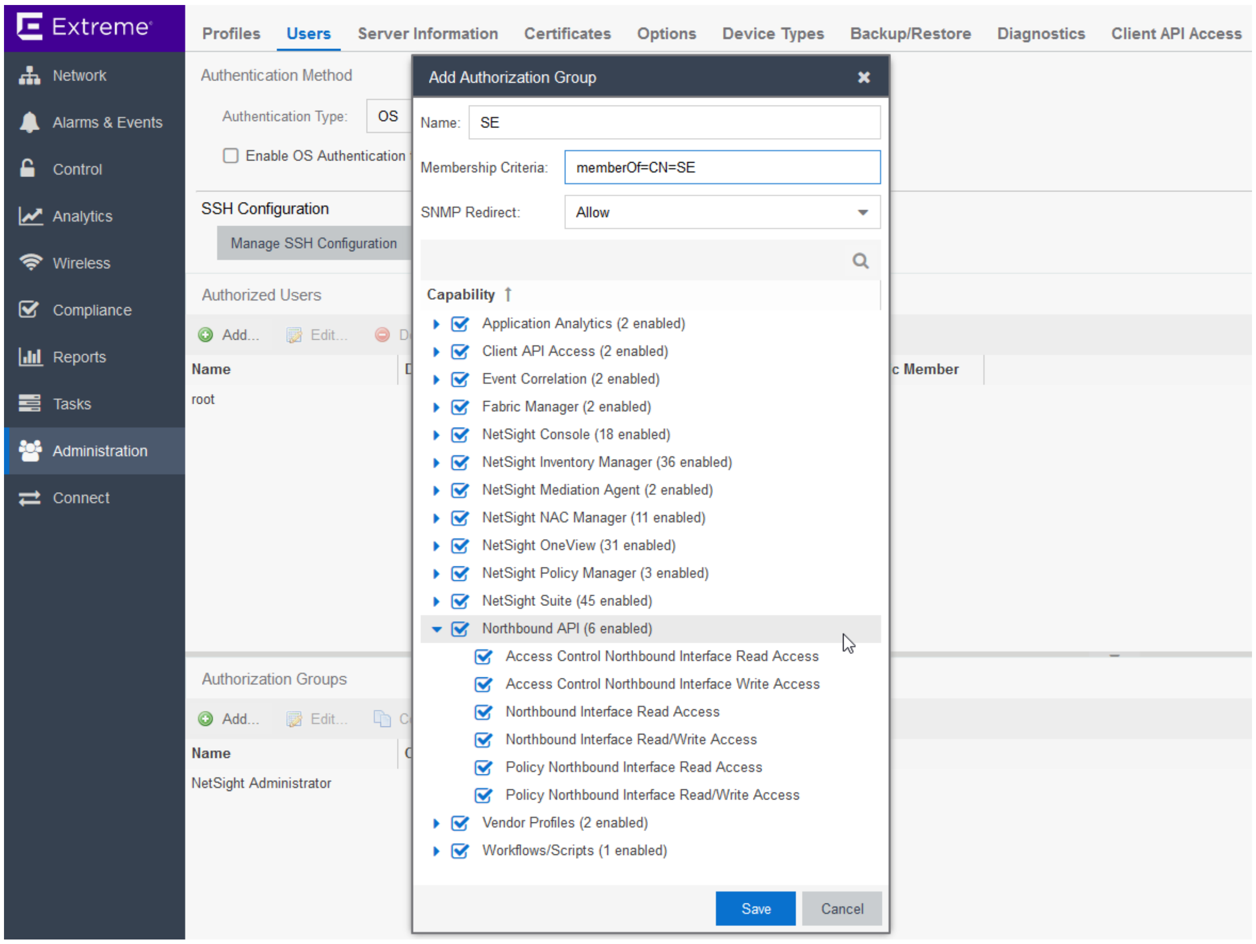
```
Script Name: NBI Mutation
Date and Time: 2020-06-28T16:02:23.390
XMC User: root
XMC User Domain:
IP: 192.168.56.121
```

```
Data returned:  {network: {createSite={status=ERROR, message=Site already
exists 'NewSite', siteId=null}}}

Cannot create Site /World/NewSite because Site already exists 'NewSite'
```

5.3.4 RBAC for API Usage

Another feature of XMC is the ability to limit access to the API based on user requests. Administrators may have a root access, and thus all access privileges, but you can use XMC to create separate users and groups, with specific privileges.

The Northbound API access rights, as with many other capabilities, can be applied very precisely.



Add Authorization Group

Name: SE

Membership Criteria: memberOf=CN=SE

SNMP Redirect: Allow

Capability
- ✓ Application Analytics (2 enabled)
- ✓ Client API Access (2 enabled)
- ✓ Event Correlation (2 enabled)
- ✓ Fabric Manager (2 enabled)
- ✓ NetSight Console (18 enabled)
- ✓ NetSight Inventory Manager (36 enabled)
- ✓ NetSight Mediation Agent (2 enabled)
- ✓ NetSight NAC Manager (11 enabled)
- ✓ NetSight OneView (31 enabled)
- ✓ NetSight Policy Manager (3 enabled)
- ✓ NetSight Suite (45 enabled)
- ✓ Northbound API (6 enabled)
  - ✓ Access Control Northbound Interface Read Access
  - ✓ Access Control Northbound Interface Write Access
  - ✓ Northbound Interface Read Access
  - ✓ Northbound Interface Read/Write Access
  - ✓ Policy Northbound Interface Read Access
  - ✓ Policy Northbound Interface Read/Write Access
- ✓ Vendor Profiles (2 enabled)
- ✓ Workflows/Scripts (1 enabled)

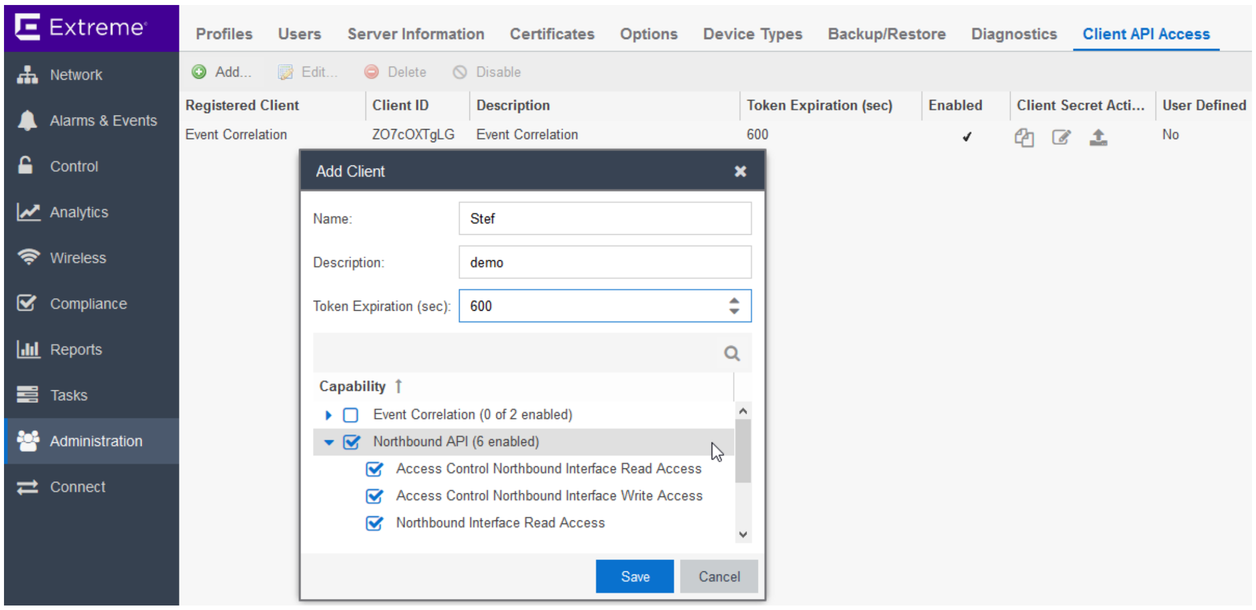
Scripts, workflow, external NBI CALLs can be limited to specific user groups.

There are two authorization methods for external NBI access:

- basic authorization
- OAuth 2.0.

Basic authorization is the standard approach, using the authorization header in HTTP or HTTPS. However, this simple access method is not very secure, as discussed in chapter 2.3.

The OAuth 2.0 method is recommended when security is an important factor. This method requires you to create a client in the Client API Access tab in the Administration > Users menu.



Registered Client	Client ID	Description	Token Expiration (sec)	Enabled	Client Secret Acti...	User Defined
Event Correlation	ZO7cOXTgLG	Event Correlation	600	✓		No

Name: Stef

Description: demo

Token Expiration (sec): 600

Capability

- Event Correlation (0 of 2 enabled)
- ✓ Northbound API (6 enabled)
  - ✓ Access Control Northbound Interface Read Access
  - ✓ Access Control Northbound Interface Write Access
  - ✓ Northbound Interface Read Access

This generates a Client ID and a Client Secret that can be used for NBI CALLs only.

Use this Client ID and Client Secret to receive an access token from the oAuth server that can be used to make valid NBI CALLs.

To receive the access token, initiate a POST to this URL:

HTTPS://<xmc-ip-address>:8443/oauth/token/access-token?grant_type=client_credentials

Include the Content-Type header and set it to application/x-www-form-urlencoded. Set the authorization header with the Client ID and the Client Secret as password.

The response is a JSON-formatted data with the access_token key containing the value expected.

Additional CALLs use Accept and Content-Type headers set to application/json and Authorization set to Bearer <access_token>.

5.3.5 External Access to the NBI API

Here is a simple example that accesses XMC via GraphQL from an external program. This example uses the basic authentication method for simplicity.

```
#!/usr/bin/env python

import json
import requests
from requests import Request, Session
from requests.auth import HTTPBasicAuth
from requests.packages.urllib3.exceptions import InsecureRequestWarning
import argparse
import getpass

def get_params():
    parser = argparse.ArgumentParser(prog = 'nbi')
    parser.add_argument('-u', '--username',
            help='Login username for the remote system')
    parser.add_argument('-p', '--password',
            help='Login password for the remote system',
            default='')
    parser.add_argument('-i', '--ip',
            help='IP of the XMC 8.1.2+ server')
    args = parser.parse_args()
    return args

args = get_params()
if args.username is None:
    # prompt for username
    args.username = input('Enter remote system username: ')
    # also get password
    args.password = getpass.getpass('Remote system password: ')

if args.ip is None:
    #prompt for XMC's IP
    args.ip = input('Enter IP of the XMC server: ')

# To disable SSL certificate verification
requests.packages.urllib3.disable_warnings( InsecureRequestWarning )

# prepare HTTPS session
session         = Session()
```

```
session.verify  = False
session.timeout = 10
session.auth    = (args.username, args.password)
session.headers.update(
    { 'Accept':        'application/json',
      'Content-type':  'application/json',
      'Cache-Control': 'no-cache',
    }
)

# define XMC-NBI query
nbiQuery = '{ network{ devices { ip nickName } } }'

# execute NBI call
nbiUrl   = 'HTTPS://' + args.ip + ':8443/nbi/graphql'
response = session.post(nbiUrl, json= {'query': nbiQuery} )

if response.status_code != 200:
    print('ERROR: HTTP ' + response.reason + '(' + str(response.status_code) + ')
')
else:
    # convert JSON string to a data structure
    inbound_data = json.loads(response.text)

    for device in inbound_data['data']['network']['devices']:
        print(device['ip'] + ' \t' + device['nickName'])
```

The output is:

```
C:\Extreme API with Python> nbi.py
Enter remote system username: root
Remote system password:
Enter IP of the XMC server: 192.168.56.10
192.168.56.121  IP Campus_08:00:27:35:2A:E4
192.168.56.143  voss03
192.168.56.125  IP Campus_08:00:27:7D:DB:E6
192.168.56.142  voss02
192.168.56.129  IP Campus_08:00:27:AD:C4:CB
192.168.56.145  voss05
192.168.56.126  IP Campus_08:00:27:07:56:3D
192.168.56.144  voss04
192.168.56.124  IP Campus_08:00:27:DD:A3:DA
192.168.56.12   FabricManager
```

```
192.168.56.123  IP Campus_08:00:27:C5:83:32
192.168.56.127  sw7
192.168.56.128  sw8
192.168.56.11   192.168.56.11
192.168.56.141  voss01
192.168.56.122  IP Campus_08:00:27:2A:B1:DF
```

5.3.6 Use NBI to Execute a Workflow

So far, NBI examples have been created mostly from the network field. This example uses the workflows field.

To execute a workflow from the NBI, you need to know the ID of the workflow you want to run, and you need to do a mutation.

Initiate a query to see the IDs for all existing workflows, as shown:

```
{
  workflows {
    allWorkflows {
      id
      name
    }
  }
}
```

Even if there is a limited number of user workflows, this query returns most of them, including system workflows.

Reuse your workflow with the “OK/KO” path selection, but this time the you execute the script from an NBI CALL. In this workflow, initially you only print the path that has been followed. Instead of doing a print of the message, set workflowMessage to the value you need. This message is displayed on the Workflow Dashboard when a workflow is successfully executed.

Rewrite the line of code in the output script of OK path to say:

```
emc_results.put("workflowMessage", "OK path has been followed")
```

Modify the KO path as well. Now, each time this workflow is successfully executed, it will print the corresponding message in the message column in the Workflow Dashboard.

Now write a quick script using the NBI to execute this workflow. First find the ID of your workflow, then execute your workflow, setting MyVariable to either OK or KO. Next, validate the execution of your workflow and print the result.

The example below uses the Python string replace method to adapt your GraphQL CALLs with your dynamic values.

```
from time import sleep

idQuery = '''
{
  workflows {
    allWorkflows {
      id
      name
    }
  }
}
'''

exeMutation = '''
mutation {
  workflows {
    startWorkflow (input: { id: <id>
                            variables: {
                              MyVariable: "<state>"
                            }
                          } ) {
      status
      errorCode
      executionId
      message
    }
  }
}
'''

messageQuery = '''
{
  workflows {
    execution(executionId: <id>) {
      variables
    }
  }
}
```

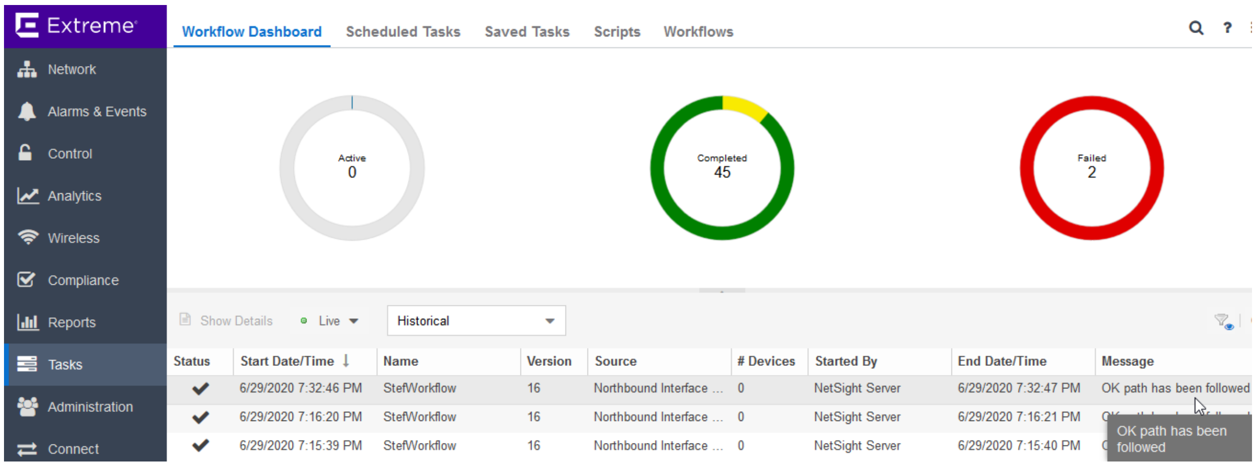
```
'''

WorkflowName = "StefWorkflow"
WorkflowID = 0
Action = "OK"
WAIT = 1

res = emc_nbi.query(idQuery)

for workflow in res['workflows']['allWorkflows']:
    if workflow['name'] == WorkflowName:
        WorkflowID = workflow['id']
        break

exeMutation = exeMutation.replace("<id>", str(WorkflowID)).replace("<state>", Action)
res = emc_nbi.mutation(exeMutation)

sleep(WAIT)

if res['workflows']['startWorkflow']['status'] == "SUCCESS":
    execId = res['workflows']['startWorkflow']['executionId']
    messageQuery = messageQuery.replace("<id>", str(execId))
    info = emc_nbi.query(messageQuery)
    print info['workflows']['execution']['variables'].get('workflowMessage')
else:
    print res
```

You must include a wait timer after the execution of your workflow, so that it has enough time to complete.

When you run your script, you should see the following output:

```
Script Name: NBI Workflow
Date and Time: 2020-06-29T19:32:46.221
XMC User: root
XMC User Domain:
IP: 192.168.56.126
"OK path has been followed"
```

On the Workflow Dashboard, you can see the same output.



Status	Start Date/Time	Name	Version	Source	# Devices	Started By	End Date/Time	Message
✓	6/29/2020 7:32:46 PM	StefWorkflow	16	Northbound Interface ...	0	NetSight Server	6/29/2020 7:32:47 PM	OK path has been followed
✓	6/29/2020 7:16:20 PM	StefWorkflow	16	Northbound Interface ...	0	NetSight Server	6/29/2020 7:16:21 PM	O
✓	6/29/2020 7:15:39 PM	StefWorkflow	16	Northbound Interface ...	0	NetSight Server	6/29/2020 7:15:40 PM	O

5.4 Axis API

Prior to the NBI API, XMC offered the Axis API. The Axis API is also sometimes called XMC Web Services. The Axis API is a REST API, returning data formatted as XML.

This API is still available, but not recommended for use anymore, with the exception of ExtremeAnalytics, because the NBI API does not provide an interface to ExtremeAnalytics.

Note*: The ExtremeAnalytics web service is called Purview, which was the original name of the solution.*

To access the ExtremeAnalytics web service description language, go to:

HTTPS://<XMC-IP-Address>:8443/axis/services/PurviewWebService?wsdl

To connect to the ExtremeAnalytics web service, go to:

HTTPS://<XMC-IP-Address>:8443/axis/services/PurviewWebService

Methods are available to query specific resources that the API exposes. The method is added to the URL Each method has parameters that must be transmitted to receive the appropriate data. The API supports Basic authentication.

5.4.1 Analytics Methods

Analytics methods are described in the following sections.

5.4.1.1 addLocation

This command creates a new location with the specified name.

Parameters:

Name		Type	Description
locationGroup		string	Location group name

Name		Type	Description
name		string	Name of new location
description		string	Location description
masks		string	IP subnets and masks of location

Returns: A string status.

5.4.1.2 *addLocationGroup*

This function creates a new location group.

Parameters:

Name	Type	Description
name	string	Name of new location group
description	string	Description of location group

Returns: A string status.

5.4.1.3 *getAppliances*

Retrieve the list of Extreme Appliances.

Returns: A list of Extreme appliances in JSON format.

5.4.1.4 *getApplicationBrowserTableData*

Retrieve data from the application browser.

Parameters:

Name	Type	Description
tableId	int	The table to retrieve the data from, available options are: 0 – appid_attribute (client & server data) 1 – appid_datapoint (application data) 2 – topn_tables 3 – application_usage_default (hourly application data) 4 – application_usage_hr_default (high rate application data)

Name	Type	Description
target	string	The target to retrieve data from, available options are: application application_group location profile target_address client target source target_type datafamily user_data TopN specific targets: appsByClient server
statistics	string	The statistic to retrieve, available options are: byte_count – total byte count flow_count – total flow count target_address – client/server IP address app_rsp_time – application response time tcp_rsp_time – network response time total – total clients, used with TopN tx_byte_count – transmit byte count rx_byte_count – receive byte count tx_flow_count – transmit flow count rx_flow_count – receive flow count client_count – client count server_count – server count application_count – application count user_data – user data contains different fields based on the tableId all_stats – all the above stats
searchCriteria	string	Key value (key=value) pair used in the database query. The available targets, with the exception of TopN, and statistics can be used as a key.
start	long	Starting timestamp for the query in milliseconds
end	long	Ending timestamp for the query in milliseconds
limit	int	Number of results to return
queryType	string	Query type, available options are: grid chartovertime

Name	**Type**	**Description**
aggType	string	Aggregation type, available options are: SUM – sum AVG - average

Returns: TableData with a structure defined by the following table.

Name	Type	Description
extraData	anyType	Additional data from the operation
lastChange	long	Timestamp of last valid data
noChange	boolean	True if the data is being stored
success	boolean	True if operation is successful
tableData	string	JSON data

5.4.1.5 getBidirectionalFlowsData

Retrieve the latest filtered bidirectional flow data from an Extreme Analytics appliance.

Parameters:

Name	**Type**	**Description**
maxRows	int	Maximum number of flows to return
searchString	string	Search string used to query the data
source	string	Extreme Analytics appliance IP address

Returns: flow data in JSON format.

5.4.1.6 getLocations

Retrieve the list of location groups and locations.

Returns: A list of location groups and locations in JSON format.

5.4.1.7 getUnidirectionalFlowsData

Retrieve the latest flow data from an Extreme Analytics appliance.

Parameters:

Name	**Type**	**Description**
maxRows	int	Maximum number of flows to return
searchString	string	Search string used to query the data

Name	Type	Description
source	string	Extreme Analytics appliance IP address

Returns: flow data in JSON format.

5.4.1.8 getVersion

Retrieve Extreme Analytics version.

Returns: Version as a string.

5.4.1.9 importLocationCSV

This creates locations with a provided CSV string.

Parameters:

Name	Type	Description
locationGroup	string	Location group name
csv	string	CSV data, data must be in a format where line 1 contains "name,ipmask" without quotes. Subsequent lines will contain the "<location name>,<IP subnet/mask>" without quotes.
overwrite	boolean	True to replace locations with the same name
purge	boolean	True to remove any locations not imported
protect	boolean	True to block operation if any locations would be overwritten

Returns: A string status.

5.4.2 Analytics API with Python

This section contains examples using the Axis API with Analytics web services. The examples retrieve data from the Applications tab as a bidirectional flow, and check the appliance version.

Although you must format the payload data as JSON, the API returns query results in XML format.

```
import xml.etree.ElementTree as ET
import json
import os
import requests
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

login = os.environ.get('xmclogin')
passw = os.environ.get('xmcpassw')

url = 'HTTPS://192.168.20.80:8443/axis/services/PurviewWebService/'
```

```
payload = {'maxRows': 2, 'searchString': 'Extreme', 'source': '192.168.20.88'}
getHeaders = {'Accept': 'application/json'}

r = requests.get(url + 'getBidirectionalFlowsData',
                 verify=False,
                 auth=(login, passw),
                 params=payload,
                 headers=getHeaders)

root = ET.fromstring(r.text)
data = json.loads(root[0].text)
print('json data: ', data)

r = requests.get(url + 'getVersion',
                 verify=False,
                 auth=(login, passw),
                 headers=getHeaders)

root = ET.fromstring(r.text)
print(root[0].text)
```

The result is:

```
C:\Extreme API with Python> analytics.py
json data:  {'root': [{'pp': '', 'fsip': '192.168.20.83', 'hn': '', 'dl':
'192.168.254.1/VMNIC_0_20.199 (ge.1.3)', 'acm': False, 'tid': -1, 'dt': '', 'du':
4349000874, 'scc': '', 'uk': 38356, 'ic': 14349, 'tapp': False, 'net': -1, 'app': -1,
'ag': 'Protocols', 'an': 'NetBIOS', 'et': 1593629388421, 'rb': 0, 'rc': 14323, 'stos':
'', 're': '', 'fsa': 'student2-o20vd8 (192.168.20.83)', 'ctos': '', 'fc': 14311,
'fsi': 'ge.1.3 [12003 - VMNIC_0_20.199]', 'rp': 0, 'fsm': '00:50:56:bf:11:c2', 'np':
'Pass Through NAC Profile', 'dcc': '', 'fst': 'NetFlow', 'omd':
'HalfSession=2\nuuid=b5df20ee', 'fdip': '192.168.20.56', 'sc': '', 'sl':
'/World/Extreme Networks France', 'ss': '192.168.254.1', 'st': 1589275522843, 'fda':
'192.168.20.56', 'bps': 0, 'aceg': '', 'fdi': 'ge.1.4 [12004 - VMNIC_0_20_198]',
'uav': '192.168.20.83\t192.168.20.56\t137\t17\tNetBIOS', 'l': '/World/Extreme Networks
France', 'ttl': '128 (C)', 'fdm': 'e8:fc:af:e7:3b:34', 'tb': 1632822, 'ct': '', 'r':
'', 'fdp': 'netbios-ns [137]', 'pd': '', 'u': '', 'tsloc': True, 'tloc': True, 'tp':
14323, 'pn': 'UDP', 'dc': ''}, {'pp': '', 'fsip': '192.168.254.161', 'hn': '', 'dl':
'', 'acm': False, 'tid': -1, 'dt': '', 'du': 5839574494, 'scc': '', 'uk': 38357, 'ic':
230146, 'tapp': False, 'net': 1805, 'app': 8540, 'ag': 'Web Applications', 'an':
'netsight.', 'et': 1593629370053, 'rb': 74163096.0, 'rc': 60244, 'stos': '', 're': '',
'fsa': '192.168.254.161', 'ctos': '', 'fc': 60104, 'fsi': 'ge.1.48 [12048]', 'rp':
299986, 'fsm': '00:04:96:9f:d8:b5', 'np': '', 'dcc': '', 'fst': 'NetFlow', 'omd':
'IssuerIdAtCommonName=Netsight
Enterasys\nSignatureAlgorithmId=shaWithRSAEncryption\nSSLVersion=TLS
1.0\nSubjectCount=6\nIssuerIdAtOrganizationName=Enterasys\nValidNotAfter=250112230000Z
\nPublicKeySize=2048\nCertificateVersion=v3\nValidNotBefore=150111230000Z\ncommonName=
Netsight.\nTLSServerName=extremecontrol\nIssuerCount=6\nSubjectOrganizationalUnitName=
NetSight
Server\nCertificateLength=798\nuuid=f7d61177\nHalfSession=0\nFlow_HostName=Netsight.\n
ServerIP=192.168.20.80', 'fdip': '192.168.20.80', 'sc': '', 'sl': '/World/Extreme
Networks France', 'ss': '192.168.254.1', 'st': 1590674777472, 'fda': 'xmc
(192.168.20.80)', 'bps': 0.0820000022649765, 'aceg': '', 'fdi': 'ge.1.3 [12003 -
VMNIC_0_20.199]', 'uav': '192.168.254.161\t192.168.20.80\t8443\t6\tnetsight.', 'l':
'', 'ttl': '63', 'fdm': '00:50:56:bf:0f:6b', 'tb': 405628416.0, 'ct': '', 'r': '',
'fdp': 'Alternate HTTPS [8443]', 'pd': '', 'u': '', 'tsloc': True, 'tloc': False,
'tp': 480195, 'pn': 'TCP', 'dc': ''}], 'count': 2}
8.4.4.26
```

6 ExtremeCloud IQ API

ExtremeCloud IQ (XIQ) is a 4th generation cloud-based network management system.

XIQ is entirely API driven internally, and offers an external API (xAPI) as an addition of the internal API to users. This is a REST API.

The XIQ xAPI is made up of four APIs:

- Identity Management
- Monitoring
- Configuration
- Presence and Location

There are dozens of endpoints to choose from.

Presence and Location require a streaming data service via a webhook.

6.1 Connect to the xAPI

To access the xAPI, first connect to HTTPS://developer.aerohive.com and register for a developer portal account. This account is free and self-approving. This is where you will access the documentation and manage tokens for authentication.

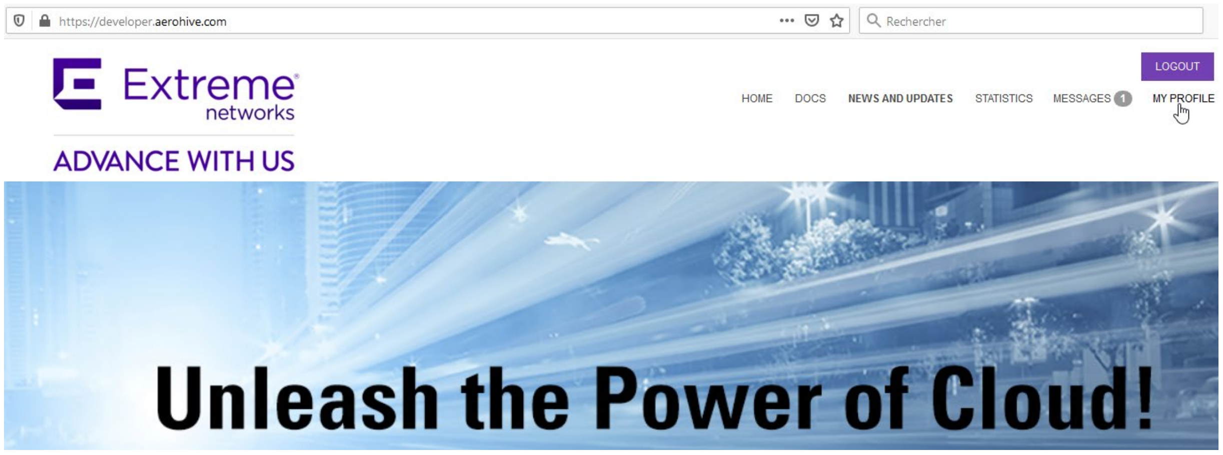
6.1.1 Create Tokens

There are two ways to authenticate to the xAPI:

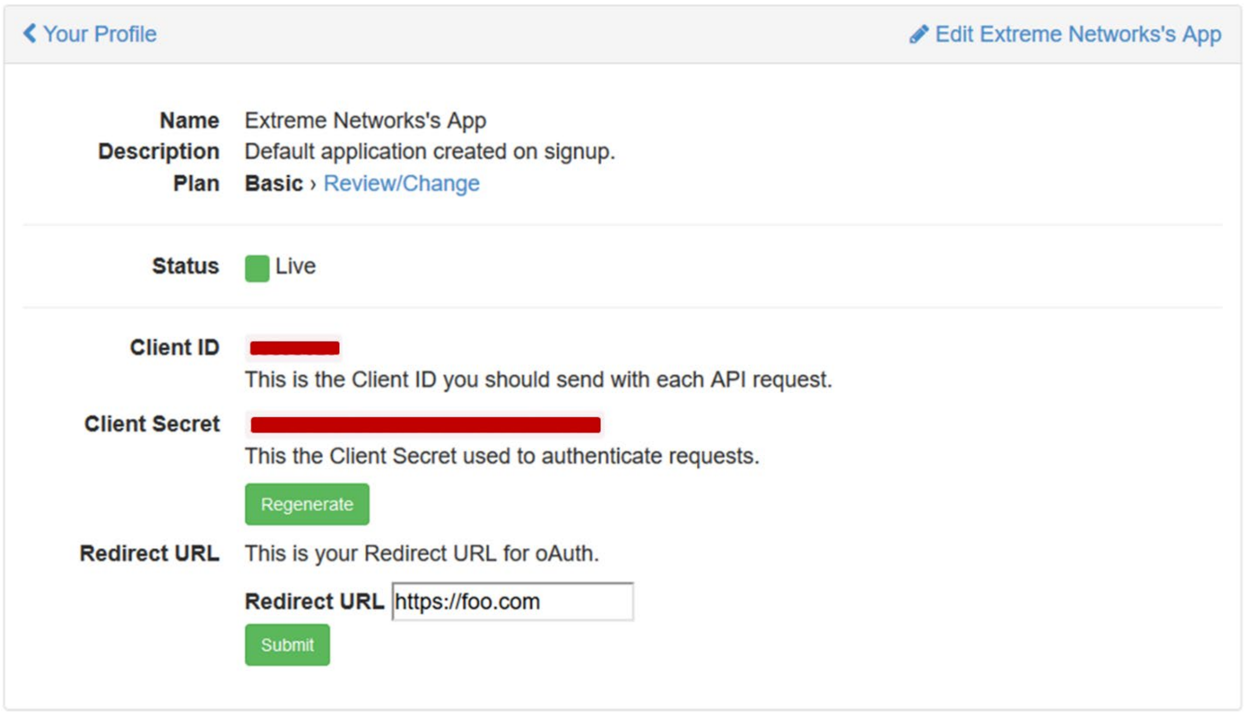
- Bearer Token
- OAuth 2.0

The easiest way is to use the Bearer token, although the OAuth 2.0 method is the most secure.

After you are registered and connected to your account, you can access your profile to create a token.



Create an app to obtain a client ID and a client secret.



< Your Profile — Edit Extreme Networks's App

Name	Extreme Networks's App
Description	Default application created on signup.
Plan	**Basic** › Review/Change
Status	🟩 Live
Client ID	[redacted] This is the Client ID you should send with each API request.
Client Secret	[redacted] This the Client Secret used to authenticate requests. Regenerate
Redirect URL	This is your Redirect URL for oAuth. **Redirect URL** https://foo.com Submit

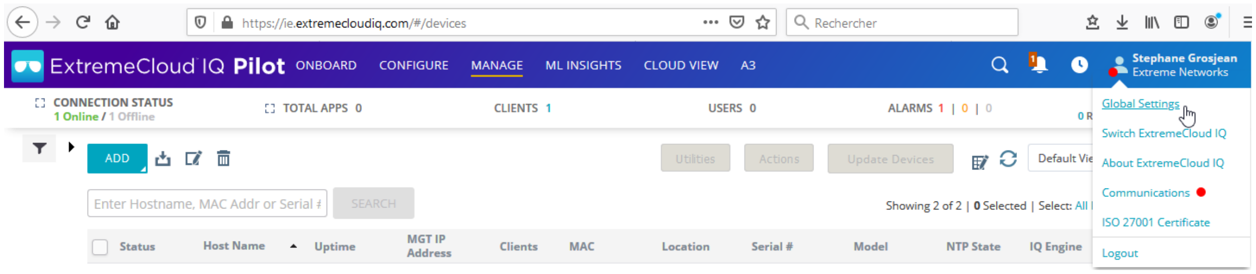
The Redirect URL is necessary regardless of the authentication method you use. XIQ offers two methods: Bearer Token (basic) and OAuth 2.0 (advanced).

With Bearer Token, the URL you use does not need to be real, but it must use HTTPS, and must match the URL in the headers of your requests.

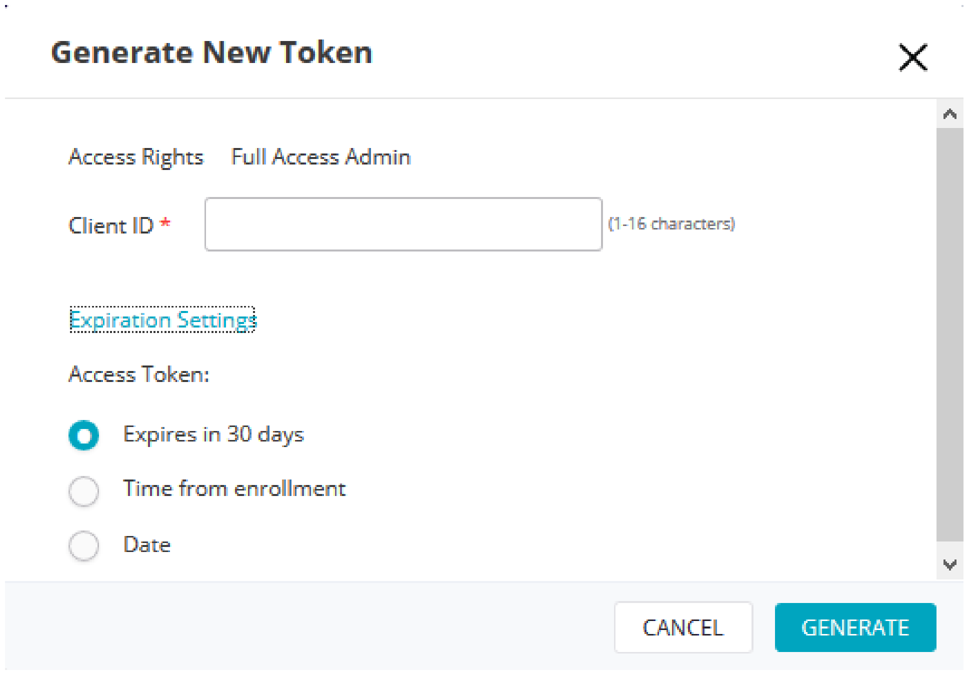
Connect to your XIQ account:

HTTPS://extremecloudiq.com

From here, navigate to the Global Settings in the upper right corner.



From the API menu, go to the API Token Management. Click **+** to create a new API Access token. You must provide your client ID.



Generate New Token

Access Rights Full Access Admin

Client ID * (1-16 characters)

Expiration Settings

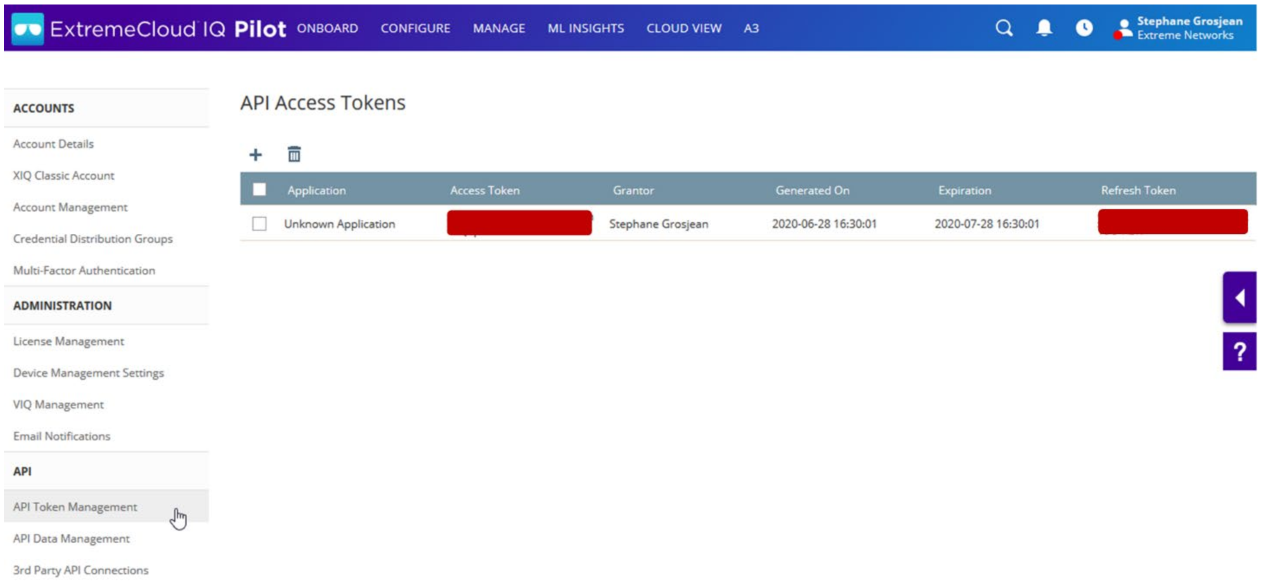
Access Token:

[✓ Expires in 30 days]

Time from enrollment

Date

By default, the token is valid for 30 days but can be extended up to 365 days.



API Access Tokens

Application	Access Token	Grantor	Generated On	Expiration	Refresh Token
Unknown Application	[redacted]	Stephane Grosjean	2020-06-28 16:30:01	2020-07-28 16:30:01	[redacted]

You should see two new tokens:

- The Access token that you will use to authenticate for every request. This is the Bearer token.
- The Refresh token.

6.1.2 Headers for the xAPI

Because XIQ uses a REST API, connections must be made via HTTPS. Specific headers are required when making a CALL to the xAPI.

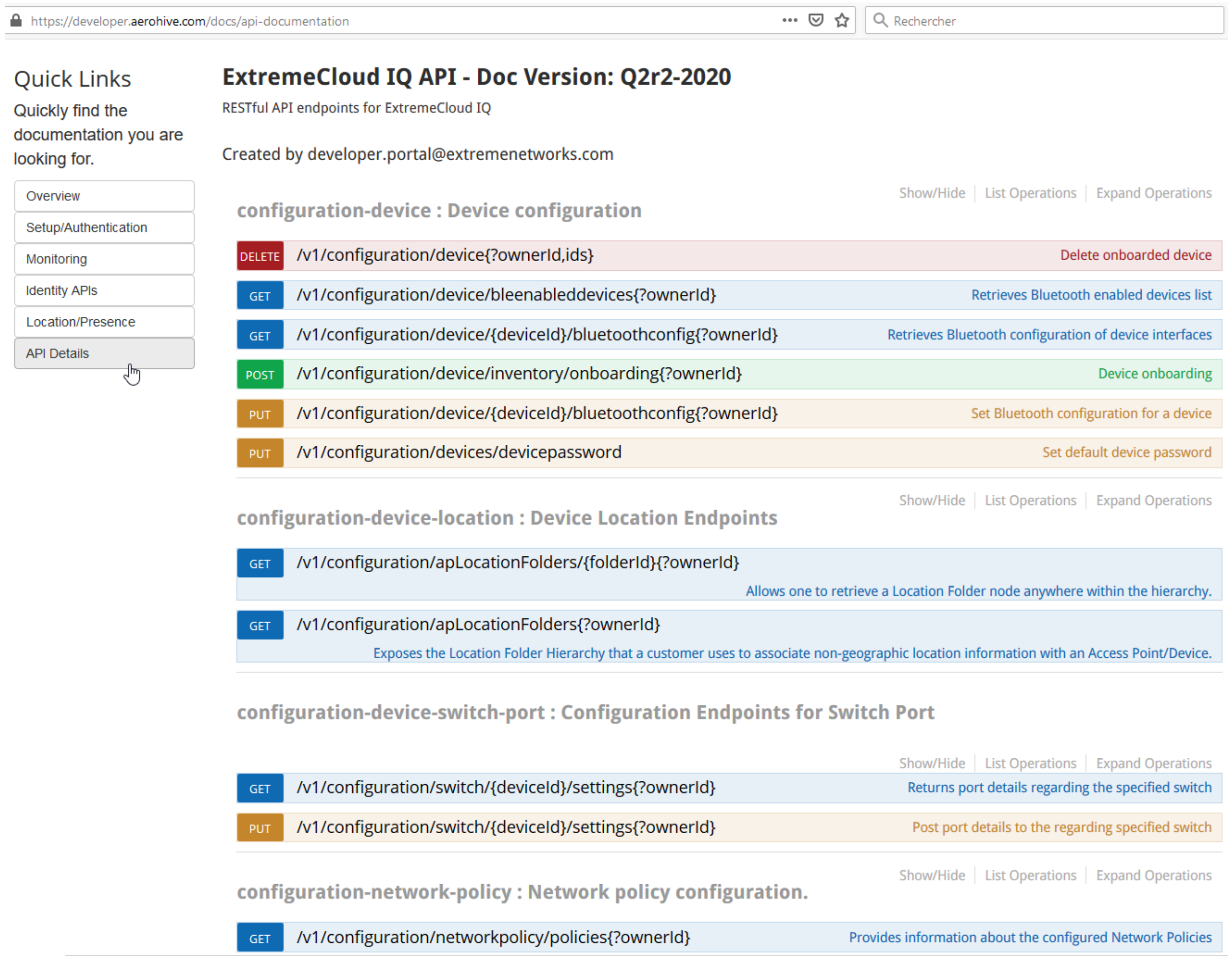
- X-AH-API-CLIENT-SECRET
- X-AH-API-CLIENT-ID
- X-AH-API-CLIENT-REDIRECT-URI
- Authorization

The first three headers are proprietary, and the last one is the regular authorization header.

As the names imply, the first three headers must be configured with, respectively, the Client Secret, the Client ID, and the Redirect URL generated on the developer portal account. The Authorization header is a Bearer token, that is, a string containing the word Bearer, with a trailing space, followed with the Access Token generated in your XIQ account.

6.1.3 xAPI Endpoints

The documentation on the developer portal is the best place to find the correct URL and HTTP methods supported for each endpoint. All the endpoints are documented with Swagger in the API Details menu.


https://developer.aerohive.com/docs/api-documentation

Quick Links
Quickly find the documentation you are looking for.

- Overview
- Setup/Authentication
- Monitoring
- Identity APIs
- Location/Presence
- API Details

ExtremeCloud IQ API - Doc Version: Q2r2-2020

RESTful API endpoints for ExtremeCloud IQ

Created by developer.portal@extremenetworks.com

configuration-device : Device configuration

DELETE	/v1/configuration/device{?ownerId,ids}	Delete onboarded device
GET	/v1/configuration/device/bleenableddevices{?ownerId}	Retrieves Bluetooth enabled devices list
GET	/v1/configuration/device/{deviceId}/bluetoothconfig{?ownerId}	Retrieves Bluetooth configuration of device interfaces
POST	/v1/configuration/device/inventory/onboarding{?ownerId}	Device onboarding
PUT	/v1/configuration/device/{deviceId}/bluetoothconfig{?ownerId}	Set Bluetooth configuration for a device
PUT	/v1/configuration/devices/devicepassword	Set default device password

configuration-device-location : Device Location Endpoints

GET	/v1/configuration/apLocationFolders/{folderId}{?ownerId}	Allows one to retrieve a Location Folder node anywhere within the hierarchy.
GET	/v1/configuration/apLocationFolders{?ownerId}	Exposes the Location Folder Hierarchy that a customer uses to associate non-geographic location information with an Access Point/Device.

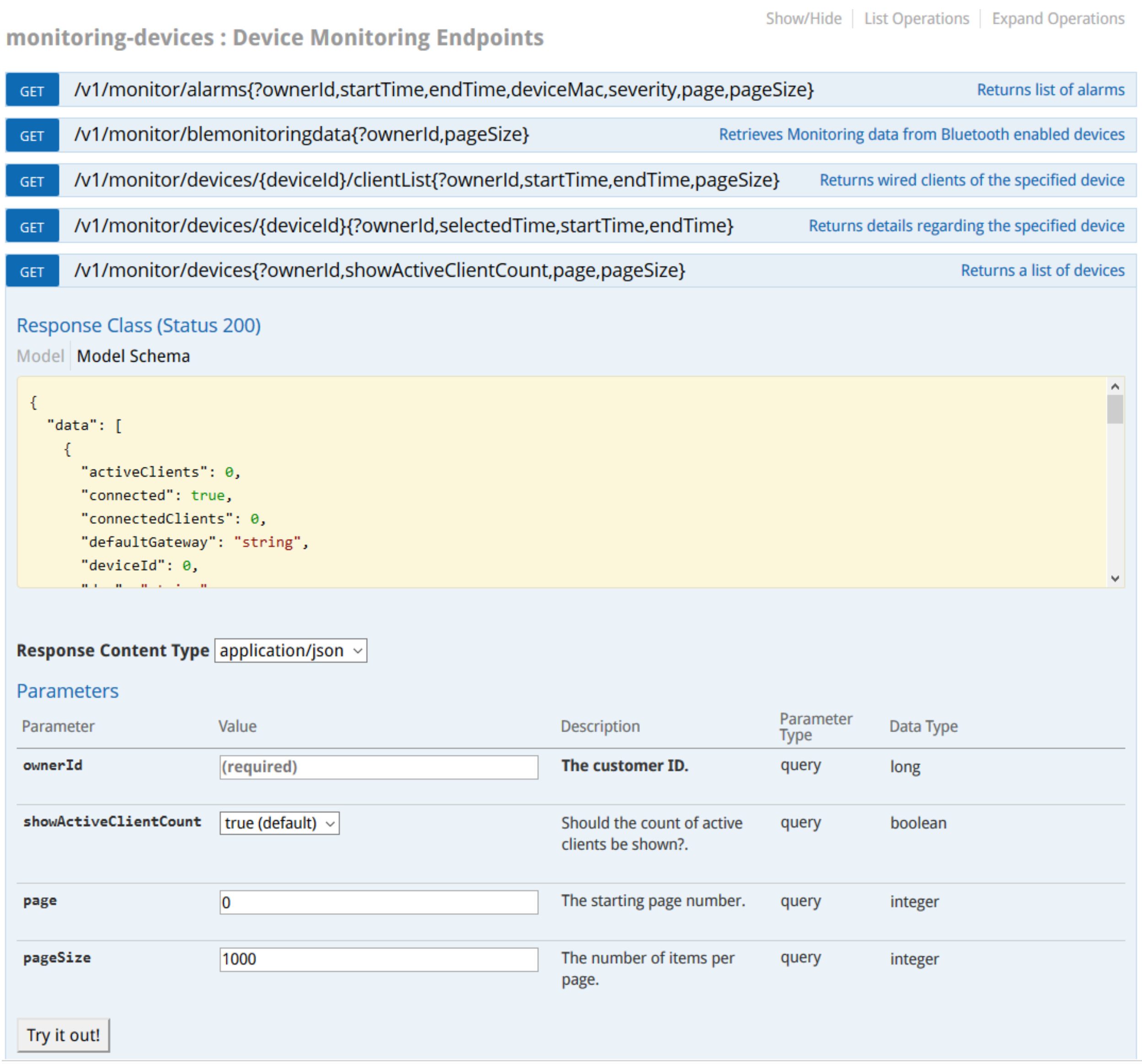
configuration-device-switch-port : Configuration Endpoints for Switch Port

GET	/v1/configuration/switch/{deviceId}/settings{?ownerId}	Returns port details regarding the specified switch
PUT	/v1/configuration/switch/{deviceId}/settings{?ownerId}	Post port details to the regarding specified switch

configuration-network-policy : Network policy configuration.

GET	/v1/configuration/networkpolicy/policies{?ownerId}	Provides information about the configured Network Policies

Swagger is easy to browse, and lists all URLs and the methods that are supported, including a brief description. Select an URL to expand it and see what type of data is expected and how the response will be displayed.



monitoring-devices : Device Monitoring Endpoints

GET	/v1/monitor/alarms{?ownerId,startTime,endTime,deviceMac,severity,page,pageSize}	Returns list of alarms
GET	/v1/monitor/blemonitoringdata{?ownerId,pageSize}	Retrieves Monitoring data from Bluetooth enabled devices
GET	/v1/monitor/devices/{deviceId}/clientList{?ownerId,startTime,endTime,pageSize}	Returns wired clients of the specified device
GET	/v1/monitor/devices/{deviceId}{?ownerId,selectedTime,startTime,endTime}	Returns details regarding the specified device
GET	/v1/monitor/devices{?ownerId,showActiveClientCount,page,pageSize}	Returns a list of devices

Response Class (Status 200)

Model | Model Schema

```
{
  "data": [
    {
      "activeClients": 0,
      "connected": true,
      "connectedClients": 0,
      "defaultGateway": "string",
      "deviceId": 0,
```

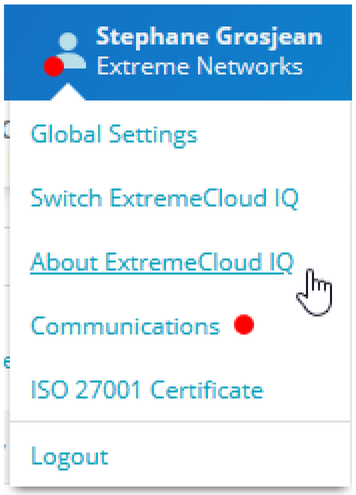
Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
ownerId	(required)	**The customer ID.**	query	long
showActiveClientCount	true (default)	Should the count of active clients be shown?.	query	boolean
page	0	The starting page number.	query	integer
pageSize	1000	The number of items per page.	query	integer

6.1.4 Parameters

The information between curly brackets in the URL is optional, with the exception of the ownerId parameter, which is mandatory. This parameter is the VIQ ID in your XIQ account, in the “About ExtremeCloud IQ” menu.



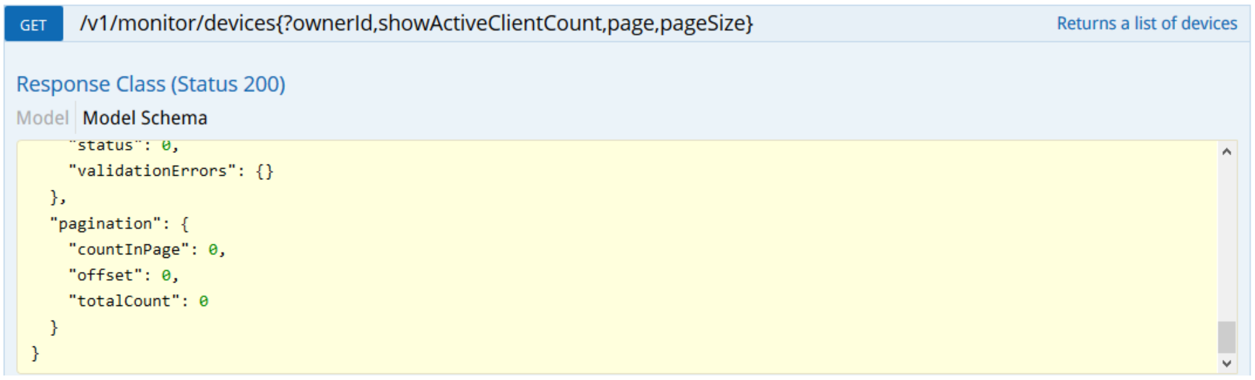
Nearly every endpoint has a pageSize parameter. By default, XIQ sends data at a maximum of 100 entries at a time, on one return, which can be too small when collecting the clients list, for example. The pageSize parameter lets you change this value to better match your needs.

The pagination entry in the data returned provides valuable information.

```
'pagination': {
    'offset': 0,
    'countInPage': 7,
    'totalCount': 7
}
```

In this example, you know that there are 7 objects and that you received 7, and you have received everything on page 0, so you don't need more pages to display the information.

If you need more than 100 objects, you must set the pageSize accordingly, or request the page numbers using the page parameter.



/v1/monitor/devices{?ownerId,showActiveClientCount,page,pageSize}

Response Class (Status 200)

```
    "status": 0,
    "validationErrors": {}
  },
  "pagination": {
    "countInPage": 0,
    "offset": 0,
    "totalCount": 0
  }
}
```

The **clientId** or **deviceId** optional parameters are unique IDs for each client or device. They are found when listing the appropriate data, and when they are added to the endpoint, where they point to more information.

6.2 Use Python with XIQ

After you have created all the tokens necessary to connect to XIQ, you can write a Python script to interact with and provide information about your XIQ account.

The best practice is to create environment variables on your OS, so that you can share your code without leaking secret information. This approach is described in chapter 2.3.5.

6.2.1 Use GET

The example below shows one way to format your headers and connect to XIQ using Python 3.x (3.7.7 in this example) and the requests library.

```
import requests
import os
import sys

baseURL = "HTTPS://ie.extremecloudiq.com/"

clientSecret = os.environ.get('clientSecret')
clientId = os.environ.get('clientId')
redirectURI = 'HTTPS://foo.com'
authToken = os.environ.get('authToken')
ownerID = os.environ.get('ownerID')

requestHeaders = {  'X-AH-API-CLIENT-SECRET': clientSecret,
                    'X-AH-API-CLIENT-ID': clientId,
                    'X-AH-API-CLIENT-REDIRECT-URI': 'HTTPS://foo.com',
                    'Authorization': authToken
                 }

params = {'ownerId': ownerID}

try:
    r = requests.get(baseURL + 'xapi/v1/monitor/devices', headers=requestHeaders, params=params)
    if r.status_code != 200:
        print("Connection failure! Unable to connect to API")
        sys.exit(1)
```

```
except requests.exceptions.RequestException as e:
    print("There was an error accessing XIQ API")
    sys.exit(1)

data = r.json()

EXOSList = []
VSPList = []
for device in data.get('data'):
    entry = {}
    if device.get('simType') == "SIMULATED":
        continue

    entry['model'] = device.get('model')
    entry['ip'] = device.get('ip')
    entry['firmware'] = device.get('osVersion')
    entry['deviceID'] = device.get('deviceId')
    entry['connected'] = device.get('connected')

    if device.get('model').startswith("X"):
        EXOSList.append(entry)
    elif device.get('model').startswith("VSP"):
        VSPList.append(entry)

print("Found {} EXOS switches and {} VSP switches".format(len(EXOSList), len(VSPList)))

if len(EXOSList):
    print("\nEXOS switches:")
    for exos in EXOSList:
        print('\t{} with IP {} running EXOS version {}'.format(exos['model'],
                                                       exos['ip'],
                                                       exos['firmware']))

        try:
            r = requests.get(baseURL + 'xapi/v1/monitor/devices/{}'.format(exos['deviceID']),
                                                                 headers=requestHeaders,
                                                                 params=params)

            if r.status_code != 200:
                print("Connection failure! Unable to connect to API")
                sys.exit(1)
        except requests.exceptions.RequestException as e:
            print("There was an error accessing XIQ API")
            sys.exit(1)

        data = r.json()
        up = 0
        down = 0
        for port in data.get('data').get('ports'):
            if port.get('status') == "UP":
                up += 1
            elif port.get('status') == "DOWN":
                down += 1
        print("\t{} ports UP and {} ports DOWN".format(up, down))

if len(VSPList):
    print("\nVSP switches:")
    for vsp in VSPList:
        print('\t{} with IP {} running VOSS version {}'.format(vsp['model'],
                                                               vsp['ip'],
                                                               vsp['firmware']))
```

It is important to note the following:

- The URL to XIQ depends on your RDC. This is the URL in your browser when you are connected to your account. In this example, this is HTTPS://ie.extremecloudiq.com.
- The endpoint that you use, in this case `v1/monitor/devices`, must be prepended by `xapi/`.

When you run the Python script, you should see the following:

```
C:\Extreme API with Python> xiq.py
Found 1 EXOS switches and 0 VSP switches

EXOS switches:
        X460_G2_24p_10_G4 with IP 192.168.254.160 running EXOS version
30.6.1.11 patch1-11
        2 ports UP and 33 ports DOWN
```

6.2.2 Use POST

To send data to XIQ, you must use either a POST or a PUT, depending on the endpoint. The documentation describes which one to use.

The framework is similar to that of the GET method, but now you need to build the JSON data to send to XIQ.

When sending JSON data, you must add two headers:

- Content-Type
- Accept

They both must be set to **application/json**.

In this example, you assign a network policy to a device. You know the serial number and the policy name of this device, and the rest of the information is retrieved from the API.

```
import requests
import os
import sys
import json

baseURL = "HTTPS://ie.extremecloudiq.com/"

clientSecret = os.environ.get('clientSecret')
clientId = os.environ.get('clientId')
redirectURI = 'HTTPS://foo.com'
authToken = os.environ.get('authToken')
ownerID = os.environ.get('ownerID')

requestHeaders = {  'X-AH-API-CLIENT-SECRET': clientSecret,
                    'X-AH-API-CLIENT-ID': clientId,
                    'X-AH-API-CLIENT-REDIRECT-URI': 'HTTPS://foo.com',
                    'Authorization': authToken
                 }

params = {'ownerId': ownerID}

POLICY = "PolicyAPI"
PolicyID = 0
DeviceID = 0
SN = "1441N-41334"
assignPolicy = {"sns": ["1441N-41334"]}

def restGet(url):
    try:
        r = requests.get(baseURL + url, headers=requestHeaders, params=params)
        if r.status_code != 200:
            print("Connection failure! Unable to connect to API")
            sys.exit(1)
    except requests.exceptions.RequestException as e:
        print("There was an error accessing XIQ API")
        sys.exit(1)

    return r.json()

data = restGet('xapi/v1/configuration/networkpolicy/policies')
for policy in data.get('data'):
    if policy['name'] == POLICY:
        PolicyID = policy['id']
        break

data = restGet('xapi/v1/monitor/devices')
for device in data.get('data'):
    if device['serialId'] == SN:
        DeviceID = device['deviceId']
        break

assignPolicy['deviceIds'] = [DeviceID]

postHeaders = {  'X-AH-API-CLIENT-SECRET': clientSecret,
                    'X-AH-API-CLIENT-ID': clientId,
                    'X-AH-API-CLIENT-REDIRECT-URI': 'HTTPS://foo.com',
                    'Authorization': authToken,
                    'Accept': 'application/json',
                    'Content-Type': 'application/json'
                 }

try:
    r = requests.post(baseURL + 'xapi/v1/configuration/networkpolicy/{}/devices'.form
at(PolicyID), headers=postHeaders, params=params, json=assignPolicy)

    if r.status_code != 200:
        print("Connection failure! error code: {}".format(r.status_code))
        sys.exit(1)
except requests.exceptions.RequestException as e:
    print("There was an error accessing XIQ API")
    sys.exit(1)
```

```
print(r.json())
```

Run this script on your test XIQ account. The following information is returned:

```
C:\Extreme API with Python> xiq2.py
{'data': {'status': 200, 'successMessage': 'Network Policy 382247794480476
applied to devices successfully.'}}
```

You have now successfully assigned a network policy to your EXOS switch.

6.2.3 Use Webhooks

Webhooks allow you to work in a reverse API, meaning instead of continuously requesting information from the API, to monitor changes (for example in doing a hash of the answer to quickly locate a change), you wait for the API to notify you when an event occurs. This approach is far more efficient and saves a great deal of bandwidth and server processing time.

XIQ offers webhook services for Presence and Location.

To configure a webhook, and receive data, several steps are necessary:

- Presence Analytics must be enabled in the Network Policy for the AP
- The application, that receives the data from XIQ, must have subscribed to it

To test your webhook, you will be using the Webhook.site website. This service provides the URL you need to configure a webhook. The XIQ webhook must send the data to this URL.

To subscribe to a webhook, you need to send a POST to the endpoint specified by the documentation. The content of the body is also detailed there.

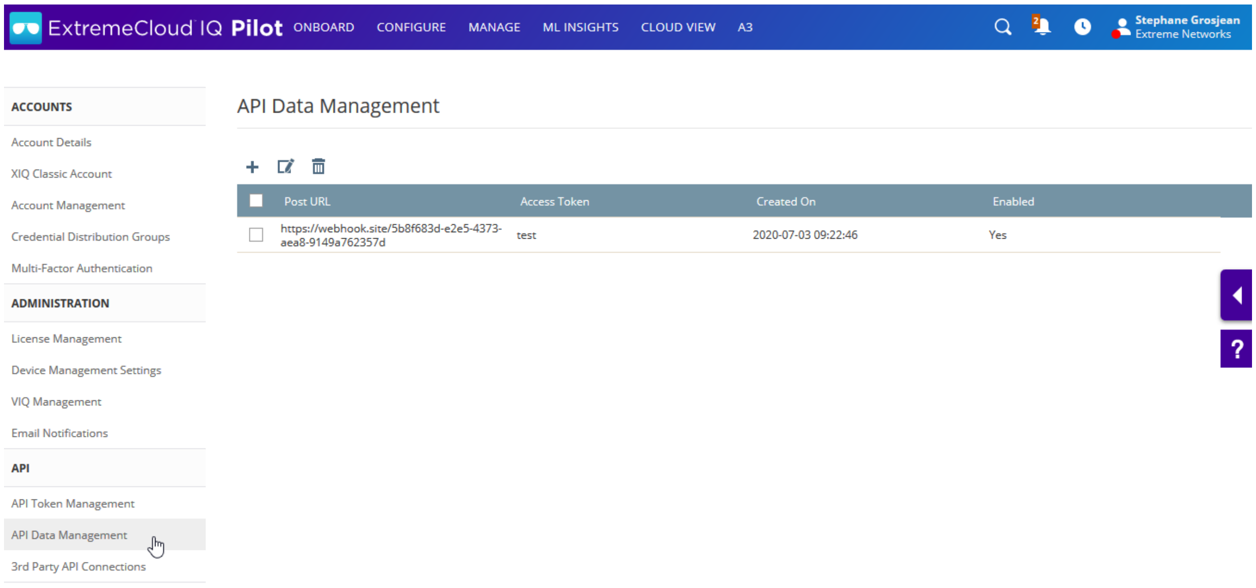
For example:

```
import requests
import os
import sys
import json

baseURL = "HTTPS://ie.extremecloudiq.com/"

clientSecret = os.environ.get('clientSecret')
clientId = os.environ.get('clientId')
redirectURI = 'HTTPS://foo.com'
authToken = os.environ.get('authToken')
ownerID = os.environ.get('ownerID')

subscriptionHeaders = {  'X-AH-API-CLIENT-SECRET': clientSecret,
                    'X-AH-API-CLIENT-ID': clientId,
                    'X-AH-API-CLIENT-REDIRECT-URI': 'HTTPS://foo.com',
                    'Authorization': authToken,
                    'Accept': 'application/json',
                    'Content-type': 'application/json'
                 }

webhookurl = 'HTTPS://webhook.site/5b8f683d-e2e5-4373-aea8-9149a762357d'

params = {'ownerId': int(ownerID), 'application': 'WebhookTest', 'url': webhookurl, '
secret': 'test', 'messageType': 'LOCATION_AP_CENTRIC', 'eventType': 'LOCATION'}
params = json.dumps(params)

try:
    r = requests.post(baseURL + 'xapi/v1/configuration/webhooks', headers=subscriptio
nHeaders, data=params)
    if r.status_code != 200:
        print("Connection failure! Unable to connect to API. error code: {}".format(r
.status_code))
        sys.exit(1)
except requests.exceptions.RequestException as e:
    print("There was an error accessing XIQ API")
    sys.exit(1)

print(r.text)
```

After you execute this code, you should see the following result:

```
C:\Extreme API with Python> webhook.py
{"data":{"ownerId":88999,"application":"WebhookTest","secret":"test","url":"H
TTPS://webhook.site/5b8f683d-e2e5-4373-aea8-
9149a762357d","messageType":"LOCATION_AP_CENTRIC","createdAt":"2020-07-
03T07:22:46.230Z","id":382247794480746}}
```

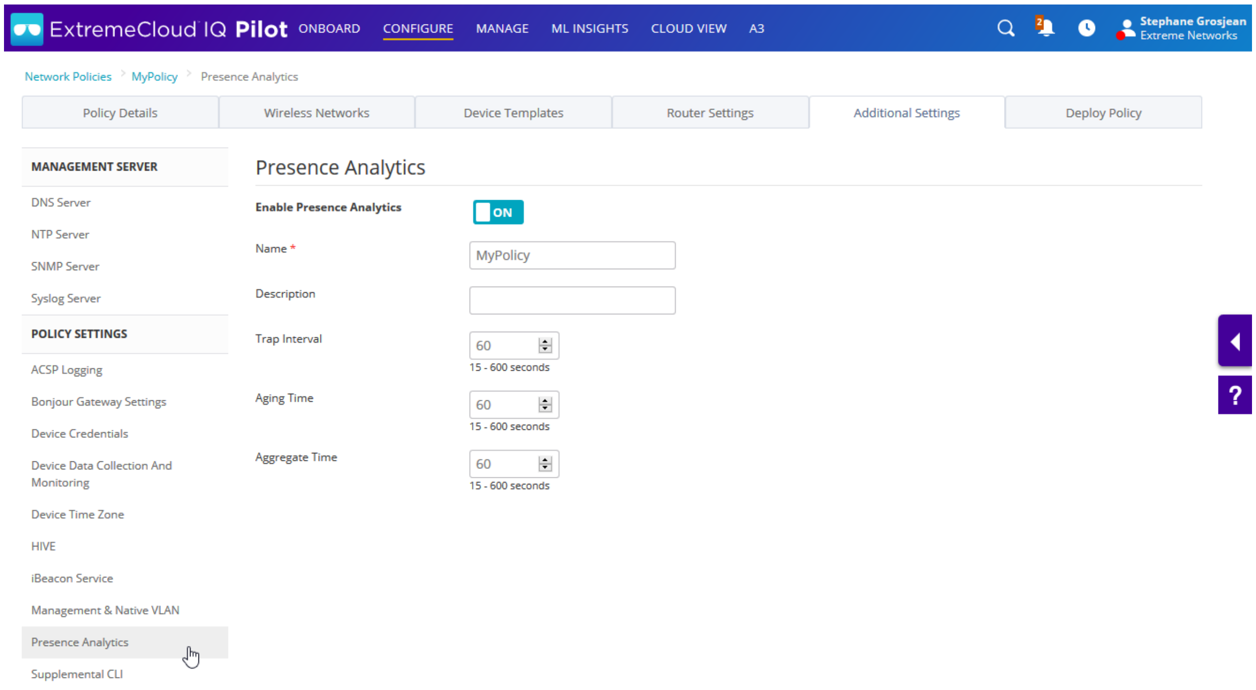
If you connect to your XIQ account, you can see the webhook is configured from the Global Settings menu, in the API Data Management panel.



API Data Management

Post URL	Access Token	Created On	Enabled
https://webhook.site/5b8f683d-e2e5-4373-aea8-9149a762357d	test	2020-07-03 09:22:46	Yes

Enable Presence Analytics in your network policy in the Additional Settings panel.

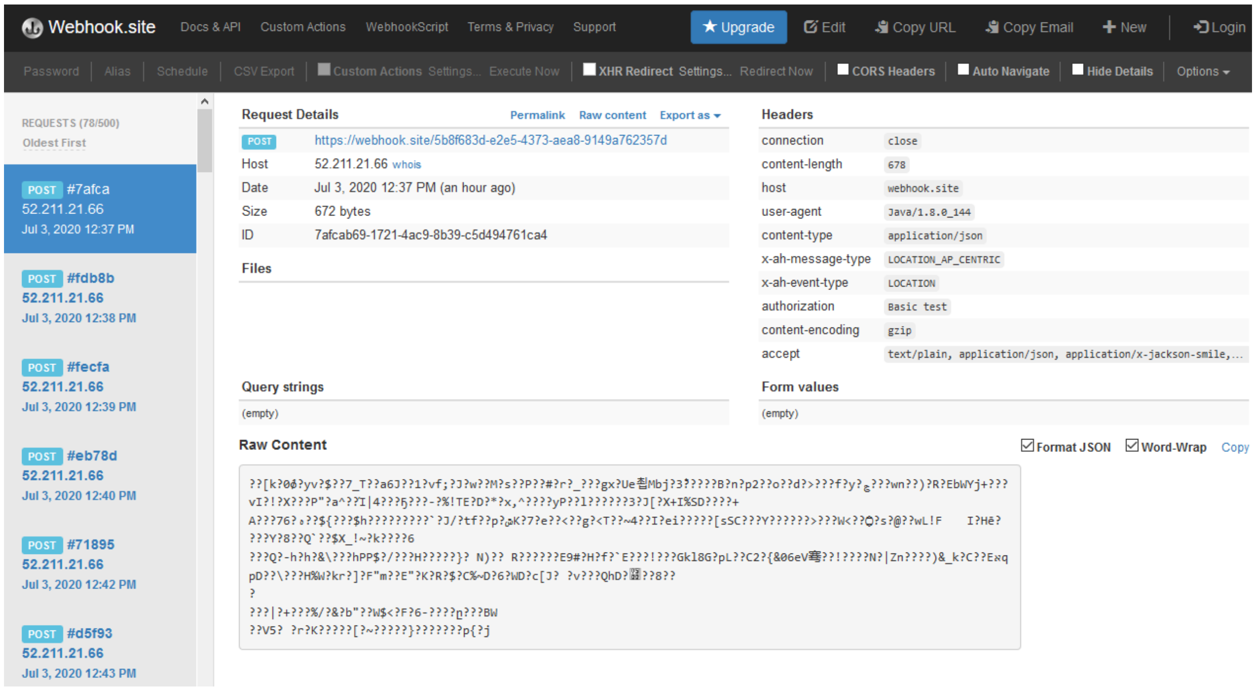


Presence Analytics

Enable Presence Analytics	ON
Name *	MyPolicy
Description	
Trap Interval	60 (15 - 600 seconds)
Aging Time	60 (15 - 600 seconds)
Aggregate Time	60 (15 - 600 seconds)

The default trap interval is set to 60 seconds.

Look at your webhook tester to see if you are receiving data every minute.



REQUESTS (78/500)
Oldest First

POST #7afca
52.211.21.66
Jul 3, 2020 12:37 PM

POST #fdb8b
52.211.21.66
Jul 3, 2020 12:38 PM

POST #fecfa
52.211.21.66
Jul 3, 2020 12:39 PM

POST #eb78d
52.211.21.66
Jul 3, 2020 12:40 PM

POST #71895
52.211.21.66
Jul 3, 2020 12:42 PM

POST #d5f93
52.211.21.66
Jul 3, 2020 12:43 PM

Request Details

POST	https://webhook.site/5b8f683d-e2e5-4373-aea8-9149a762357d
Host	52.211.21.66 whois
Date	Jul 3, 2020 12:37 PM (an hour ago)
Size	672 bytes
ID	7afcab69-1721-4ac9-8b39-c5d494761ca4

Files

Headers

connection	close
content-length	678
host	webhook.site
user-agent	Java/1.8.0_144
content-type	application/json
x-ah-message-type	LOCATION_AP_CENTRIC
x-ah-event-type	LOCATION
authorization	Basic test
content-encoding	gzip
accept	text/plain, application/json, application/x-jackson-smile,...

Query strings

(empty)

Form values

(empty)

Raw Content

```
??[k?0ɸ?yv?$??7_T??a6J??1?vf;?J?w??M?s??P??#?r?_???gx?Ue췹Mbj?3?????B?n?p2??o??d?>???f?y?ᶓ???wn??)?R?EbWYj+???
vI?!?X???P"?a^??I|4???ҕ???-?%!TE?D?*?x,^????yP??l??????3?J[?X+I%SD????+
A???76?ₔ??${???$h?????????`?J/?tf??p?ฒK?7?e??<??g?<T??~4??I?ei?????[sSC???Y??????>???W<??Ọ?s?@??wL!F    I?Hē?
???Y?8??Q`??$X_!~?k????6
???Q?-h?h?&\???hPP$?/???H?????}? N)?? R??????E9#?H?f?`E???!???Gkl8G?pL??C2?{&06eV霻??!????N?|Zn????)&_k?C??Eאq
pD??\???H%W?kr?]?F"m??E"?K?R?$?C%~D?6?WD?c[J? ?v???QhD?[illegible]??8??
?
???|?+???%/?&?b"??W$<?F?6-????ꞑ???BW
??V5? ?r?K?????[?~?????}???????p{?j
```

To stop the webhook, send a DELETE to the API endpoint, and specify the correct subscription ID. You can also list and modify all your webhooks.

This example shows how to list your webhooks:

```
import requests
import os
import sys

baseURL = "HTTPS://ie.extremecloudiq.com/"

clientSecret = os.environ.get('clientSecret')
clientId = os.environ.get('clientId')
redirectURI = 'HTTPS://foo.com'
authToken = os.environ.get('authToken')
ownerID = os.environ.get('ownerID')


requestHeaders = {  'X-AH-API-CLIENT-SECRET': clientSecret,
                    'X-AH-API-CLIENT-ID': clientId,
                    'X-AH-API-CLIENT-REDIRECT-URI': 'HTTPS://foo.com',
                    'Authorization': authToken
```

```
                  }

params = {'ownerId': ownerID}

def restGet(url):
    try:
        r = requests.get(baseURL + url, headers=requestHeaders, params=params)
        if r.status_code != 200:
            print("Connection failure! Unable to connect to API")
            print(r.content)
            sys.exit(1)
    except requests.exceptions.RequestException as e:
        print("There was an error accessing XIQ API")
        sys.exit(1)

    return r.json()

data = restGet('xapi/v1/configuration/webhooks')
print(data)
```

The result should match your current configuration:

```
C:\Extreme API with Python> webhookget.py
{'data': [{'ownerId': 88999, 'application': 'WebhookTest', 'secret': 'test',
'url': 'HTTPS://webhook.site/5b8f683d-e2e5-4373-aea8-9149a762357d',
'messageType': 'LOCATION_AP_CENTRIC', 'createdAt': '2020-07-
03T07:22:46.230Z', 'id': 382247794480746}], 'pagination': {'offset': 0,
'countInPage': 1, 'totalCount': 1}}
```

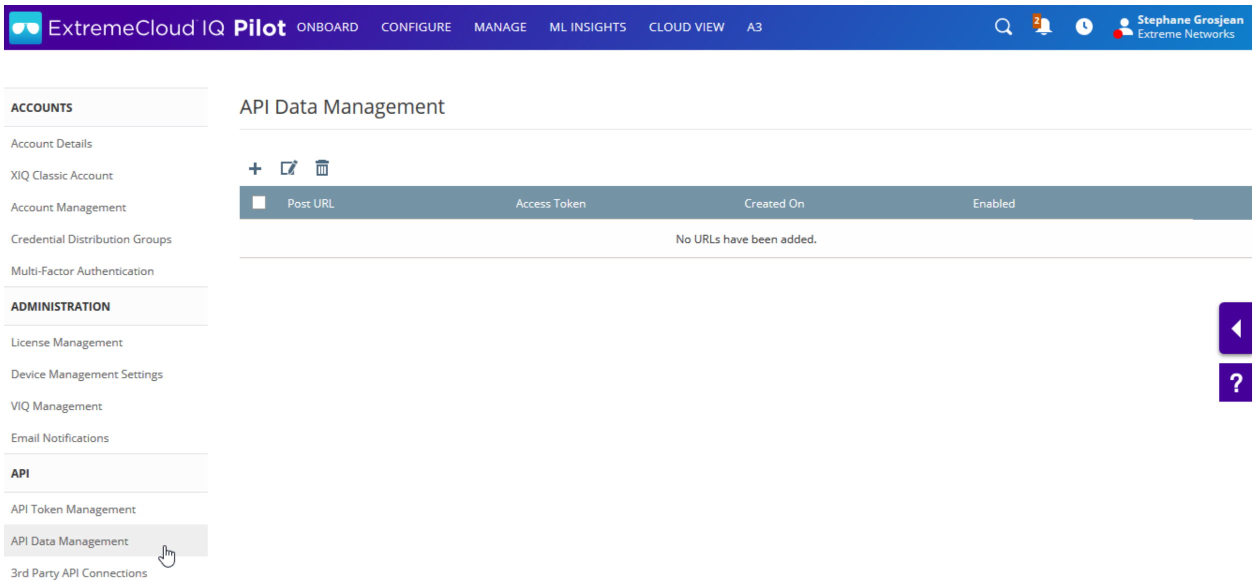
Stop the webhook by adding the following code at the end of your previous example:

```
subID = 0
for webhook in data.get('data'):
    if webhook.get('application') == "WebhookTest":
        subID = webhook.get('id')
        break

if subID:
    r = requests.delete(baseURL + 'xapi/v1/configuration/webhooks/{}'.format(subID),
                        headers=requestHeaders, params=params)
    print(r.status_code)
    print(r.text)
```

The result is shown here:

```
C:\Extreme API with Python> webhookget.py
{'data': [{'ownerId': 88999, 'application': 'WebhookTest', 'secret': 'test',
'url': 'HTTPS://webhook.site/5b8f683d-e2e5-4373-aea8-9149a762357d',
'messageType': 'LOCATION_AP_CENTRIC', 'createdAt': '2020-07-
03T07:22:46.230Z', 'id': 382247794480746}], 'pagination': {'offset': 0,
'countInPage': 1, 'totalCount': 1}}
200
```

Validate that the webhook subscription is no longer part of your XIQ configuration.



API Data Management

Post URL	Access Token	Created On	Enabled
No URLs have been added.			

7 Extreme Campus Controller API

Extreme Campus Controller is a wired and wireless management solution for campus and IoT networks. As with most Extreme Networks solutions, it also provides a REST API. It supports Bearer tokens and OAuth 2.0 for authentication and authorization.

The Extreme Campus Controller REST API is divided into three main parts:

- Application Manager API
- Platform Manager API
- Gateway API

The full documentation for each part is available at:

https://www.extremenetworks.com/support/documentation/extreme-campus-controller-latest-documentation/

The Application Manager API provides a programmatic interface to install and manage applications, create and manage containers, manage storage and images, and access system information and features.

The Platform Manager API provides a programmatic interface to access and manage backup files, flash memory, license information, controller logs, network test data and platform settings.

The Gateway API provides a single entry point between external requesting clients and the multiple internal APIs that help install, manage, and extend applications that are supported by the Extreme Campus Controller platform.

Everything that is part of the UI is accessible with the REST API.

7.1 Set Up Authorization

To obtain a token for accessing the API, you must do a POST to the endpoint shown below and include your login and password.

HTTPS://<XCC-IP-Address>:5825/management/v1/oauth2/token

The body must be in JSON format, with the following data and structure:

```
{
  'grantType': 'password',
  'userId': '<login>',
  'password': '<password>'
}
```

Add the **scope** entry in the body, to specify a given scope for this access.

As of version 5.06, the following scopes are available:

```
"scopes" : {
    "site" : "RW",
    "network" : "RW",
    "deviceAp" : "RW",
    "deviceSwitch" : "RW",
    "eGuest" : "RW",
    "adoption" : "RW",
    "troubleshoot" : "RW",
    "onboardAaa" : "RW",
    "onboardCp" : "RW",
    "onboardGroupsAndRules" : "RW",
    "onboardGuestCp" : "RW",
    "platform" : "RW",
    "account" : "RW",
    "application" : "RW",
    "license" : "RW",
    "cliSupport" : "RW"
  }
```

In return, the Extreme Campus Controller server issues a token. This is the Bearer token that you will use for REST API CALLs. The token has a finite lifetime that defaults to 7200 seconds (which is 2 hours).

Depending on the configured user privileges, the adminRole is either FULL, allowing access in read-write (RW) to everything, or read (R), granting read-only access.

This information is part of the response from the Extreme Campus Controller server. For example:

```python
import os
import requests
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

xcclogin = os.environ.get('xcclogin')
xccpassw = os.environ.get('xccpassw')

auth_url = 'HTTPS://192.168.20.90:5825/management/v1/oauth2/token'
auth_body = {'grantType': 'password', 'userId': xcclogin, 'password': xccpassw}

r = requests.post(auth_url, verify=False, json=auth_body)

print(r.text)
```

The response:

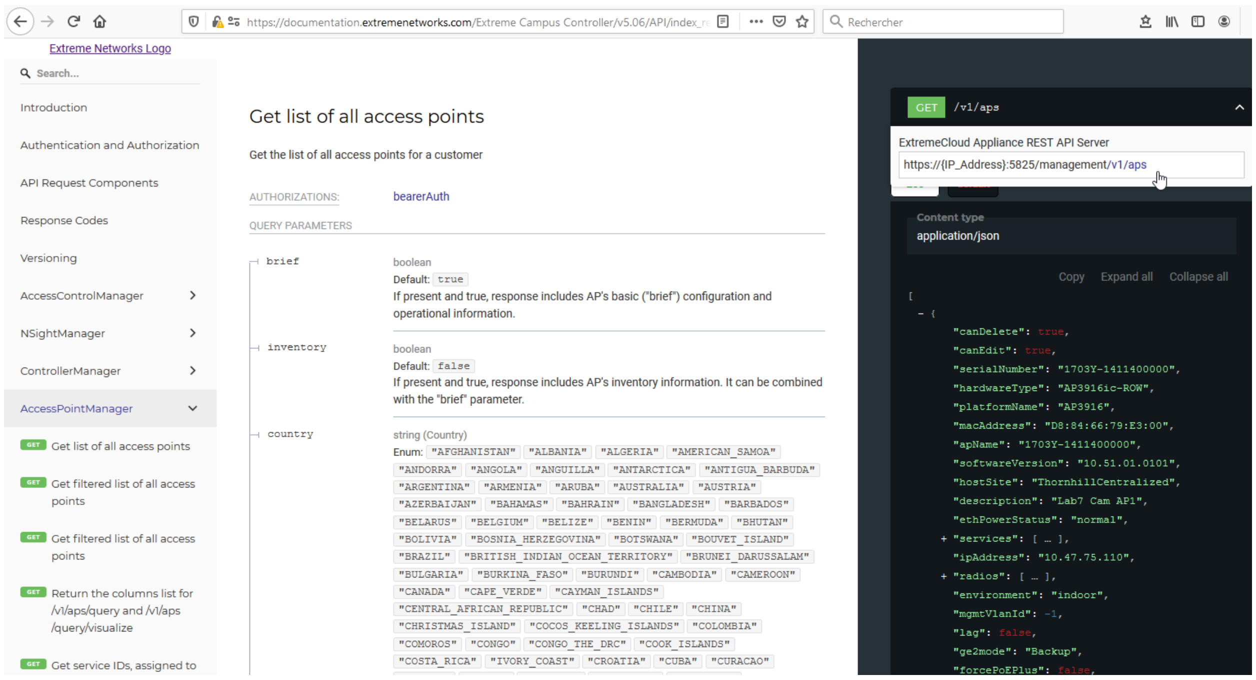
```
C:\Extreme API with Python> xcc.py
{
  "access_token" :
"eyJraWQiOiIxODI1RS1DMjYwMSIsInR5cCI6IkpXVCIsImFsZyI6IlJTMjU2In0.eyJzdWIiOiJVbmRlZmluZ
WQ6MTgyNUUtQzI2MDEiLCJsaWNlbnNlIjoib2siLCJzY29wZSI6IntcInNpdGVcIjpcIlJXXCIsXCJuZXR3b3J
rXCI6XCJSV1wiLFwiZGV2aWNlQXBcIjpcIlJXXCIsXCJkZXZpY2VTd2l0Y2hcIjpcIlJXXCIsXCJlR3Vlc3RcI
jpcIlJXXCIsXCJhZG9wdGlvblwiOlwiUldcIixcInRyb3VibGVzaG9vdFwiOlwiUldcIixcIm9uYm9hcmRBYWF
cIjpcIlJXXCIsXCJvbmJvYXJkQ3BcIjpcIlJXXCIsXCJvbmJvYXJkR3JvdXBzQW5kUnVsZXNcIjpcIlJXXCIsX
CJvbmJvYXJkR3Vlc3RDcFwiOlwiUldcIixcInBsYXRmb3JtXCI6XCJSV1wiLFwiYWNjb3VudFwiOlwiUldcIix
cImFwcGxpY2F0aW9uXCI6XCJSV1wiLFwibGljZW5zZVwiOlwiUldcIixcImNsaVN1cHBvcnRcIjpcIlJXXCJ9I
iwiaXNzIjoiYOUNDLjE4MjVFLUMyNjAxIiwiZXh0cmVtZV9yb2xlIjoiRlVMTCIsImV4cCI6MTU5Mzk1MDkxOS
wianRpIjoieW9zdHJvdnMifQ.IckhxZnvkF3JZSJd5fBaBtM_aOlGhxHKaul5vcTbZ6cklhPojKz5EEZR0zs7V
G09qhhwUrKmmMCRlb6JRAnZYZFC3ZZlTJWrZzUlLJtImn2fLsX8FcEy6Ep0j3tVBVD6yjNnzR2zLQM6btbBTie
zl_dUd2s5jT2JCckbOhMW_sUqDb2pKxqO2KN9-xa0QT7lTFpVFFAwqPvCLnHBvMu7Ab3v-cFCWmFt34fVaw-
vS5gm3dj7S612cNek2fhFDg_2CcWGWc3xGBjmZGNqfKf3yeuJ2YQm1ezIlHRQ8qHGIBwnqdtbejzmKqM11S3E6
gzI-OSGU3qrdXn5CvxGdtR0YQ",
  "token_type" : "Bearer",
  "expires_in" : 7200,
  "idle_timeout" : 356400,
  "refresh_token" : "a33d28c028959457acf46cde6385a3",
  "adminRole" : "FULL",
  "scopes" : {
    "site" : "RW",
    "network" : "RW",
    "deviceAp" : "RW",
    "deviceSwitch" : "RW",
    "eGuest" : "RW",
    "adoption" : "RW",
    "troubleshoot" : "RW",
    "onboardAaa" : "RW",
    "onboardCp" : "RW",
    "onboardGroupsAndRules" : "RW",
    "onboardGuestCp" : "RW",
    "platform" : "RW",
    "account" : "RW",
    "application" : "RW",
    "license" : "RW",
    "cliSupport" : "RW"
  }
}
```

When you have the Bearer token, you can add it to the Authorization header along with the Accept and Content-Type headers, set to application/json.

In the documentation, locate the endpoint you want to use to access, modify, create, or delete information on the Extreme Campus Controller server.

7.2 Use GET method

In this example, you can count how many APs are connected to your Extreme Campus Controller server. To do this, use the /v1/aps endpoint. This endpoint is under the management root path.



https://documentation.extremenetworks.com/Extreme Campus Controller/v5.06/API/index_re

Extreme Networks Logo

Search...

- Introduction
- Authentication and Authorization
- API Request Components
- Response Codes
- Versioning
- AccessControlManager
- NSightManager
- ControllerManager
- AccessPointManager
  - GET Get list of all access points
  - GET Get filtered list of all access points
  - GET Get filtered list of all access points
  - GET Return the columns list for /v1/aps/query and /v1/aps/query/visualize
  - GET Get service IDs, assigned to

Get list of all access points

Get the list of all access points for a customer

AUTHORIZATIONS: bearerAuth

QUERY PARAMETERS

brief	boolean Default: `true` If present and true, response includes AP's basic ("brief") configuration and operational information.
inventory	boolean Default: `false` If present and true, response includes AP's inventory information. It can be combined with the "brief" parameter.
country	string (Country) Enum: "AFGHANISTAN" "ALBANIA" "ALGERIA" "AMERICAN_SAMOA" "ANDORRA" "ANGOLA" "ANGUILLA" "ANTARCTICA" "ANTIGUA_BARBUDA" "ARGENTINA" "ARMENIA" "ARUBA" "AUSTRALIA" "AUSTRIA" "AZERBAIJAN" "BAHAMAS" "BAHRAIN" "BANGLADESH" "BARBADOS" "BELARUS" "BELGIUM" "BELIZE" "BENIN" "BERMUDA" "BHUTAN" "BOLIVIA" "BOSNIA_HERZEGOVINA" "BOTSWANA" "BOUVET_ISLAND" "BRAZIL" "BRITISH_INDIAN_OCEAN_TERRITORY" "BRUNEI_DARUSSALAM" "BULGARIA" "BURKINA_FASO" "BURUNDI" "CAMBODIA" "CAMEROON" "CANADA" "CAPE_VERDE" "CAYMAN_ISLANDS" "CENTRAL_AFRICAN_REPUBLIC" "CHAD" "CHILE" "CHINA" "CHRISTMAS_ISLAND" "COCOS_KEELING_ISLANDS" "COLOMBIA" "COMOROS" "CONGO" "CONGO_THE_DRC" "COOK_ISLANDS" "COSTA_RICA" "IVORY_COAST" "CROATIA" "CUBA" "CURACAO"

GET /v1/aps

ExtremeCloud Appliance REST API Server

https://{IP_Address}:5825/management/v1/aps

Content type

application/json

```
[
 - {
      "canDelete": true,
      "canEdit": true,
      "serialNumber": "1703Y-1411400000",
      "hardwareType": "AP3916ic-ROW",
      "platformName": "AP3916",
      "macAddress": "D8:84:66:79:E3:00",
      "apName": "1703Y-1411400000",
      "softwareVersion": "10.51.01.0101",
      "hostSite": "ThornhillCentralized",
      "description": "Lab7 Cam AP1",
      "ethPowerStatus": "normal",
    + "services": [ … ],
      "ipAddress": "10.47.75.110",
    + "radios": [ … ],
      "environment": "indoor",
      "mgmtVlanId": -1,
      "lag": false,
      "ge2mode": "Backup",
      "forcePoEPlus": false,
```

One way to obtain this information using Python is shown here:

```python
import os
import requests
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

xcclogin = os.environ.get('xcclogin')
xccpassw = os.environ.get('xccpassw')

baseURL = 'HTTPS://192.168.20.90:5825'
auth_url = baseURL + '/management/v1/oauth2/token'
auth_body = {'grantType': 'password', 'userId': xcclogin, 'password': xccpassw}

r = requests.post(auth_url, verify=False, json=auth_body)
bearerToken = r.json().get('access_token')

myHeaders = {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + bearerToken
}
```

```
def restGet(endpoint):
    try:
        r = requests.get(baseURL + endpoint, verify=False, headers=myHeaders, timeout=5)
    except requests.exceptions.Timeout:
        print("Timeout!")
        return None

    if r.status_code != 200:
        print("Cannot access XCC REST API! Error code: {}".format(r.status_code))
        print(r.content)
        return None

    return r.json()

data = restGet('/management/v1/aps')
if data:
    print(len(data))
```

Executing the script results in the following:

```
C:\Extreme API with Python> xcc.py

35
```

You can also modify your code to count the number of AP models.

```
data = restGet('/management/v1/aps')
if data:
    APList = []
    print(len(data))
    for ap in data:
        APList.append(ap.get('platformName'))

    for count, model in sorted(((APList).count(ap), ap) for ap in set(APList)):
        print("{} {}".format(model, count))
```

You should see the following:

```
C:\Extreme API with Python> xcc.py
35
AP310 1
AP3917 1
AP460 1
AP3912 3
```

```
AP3935 3
AP410 3
AP510 3
AP3915 4
AP3916 4
AP505 4
SA201 8
```

7.3 Use POST

The PUT or POST method is very similar to GET. The main difference is that you must pass the data structure you want to add or modify during the REST API CALL.

For example, create a new role (for Policy) in Extreme Campus Controller. Name it Stef. According to the documentation, there are three required types of information in the JSON you will send, and the expected response should be a 201 status code.

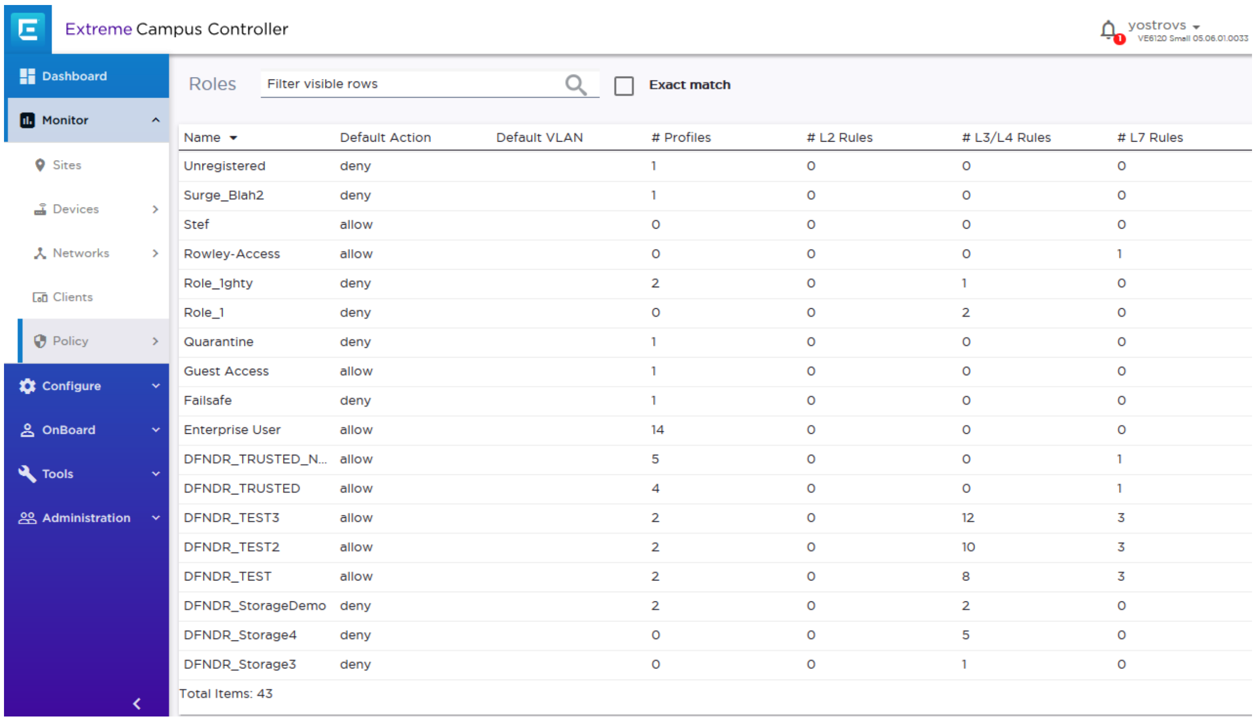
Tweak your previous CALL to minimize the output length and store the information in a more practical way. Next, issue a POST to create the new role, if it doesn't already exist.

```python
data = restGet('/management/v1/aps')
if data:
    APList = []
    print("There are {} APs".format(len(data)))
    for ap in data:
        APList.append(ap.get('platformName'))

    new_list = []
    for count, model in sorted(((APList).count(ap), ap) for ap in set(APList)):
        entry = {}
        entry['model'] = model
        entry['count'] = count
        new_list.append(entry)

    print(new_list)
data = restGet('/management/v3/roles')
if data:
    print("\nThere are {} roles".format(len(data)))
    found = False
    for name in data:
        if name.get('name') == "Stef":
            print("The role Stef already exists")
            found = True
            break

    if not found:
        role = {'name': 'Stef', 'defaultAction': 'allow', 'defaultCos': None}
        r = requests.post(baseURL + '/management/v3/roles', verify=False,
                          headers=myHeaders, json=role)
        if r.status_code != 201:
            print("Cannot access XCC REST API! Error code: {}".format(r.status_code))
            print(r.content)
            exit(0)
```

The output should look like this:

```
C:\Extreme API with Python> xcc.py
There are 35 APs
[{'model': 'AP310', 'count': 1}, {'model': 'AP3917', 'count': 1}, {'model':
'AP460', 'count': 1}, {'model': 'AP3912', 'count': 3}, {'model': 'AP3935',
'count': 3}, {'model': 'AP410', 'count': 3}, {'model': 'AP510', 'count': 3},
{'model': 'AP3915', 'count': 4}, {'model': 'AP3916', 'count': 4}, {'model':
'AP505', 'count': 4}, {'model': 'SA201', 'count': 8}]

There are 42 roles
```

Log in to your Extreme Campus Controller server to confirm that you have one more role (43) and the role you created is listed.



Name	Default Action	Default VLAN	# Profiles	# L2 Rules	# L3/L4 Rules	# L7 Rules
Unregistered	deny		1	0	0	0
Surge_Blah2	deny		1	0	0	0
Stef	allow		0	0	0	0
Rowley-Access	allow		0	0	0	1
Role_1ghty	deny		2	0	1	0
Role_1	deny		0	0	2	0
Quarantine	deny		1	0	0	0
Guest Access	allow		1	0	0	0
Failsafe	deny		1	0	0	0
Enterprise User	allow		14	0	0	0
DFNDR_TRUSTED_N...	allow		5	0	0	1
DFNDR_TRUSTED	allow		4	0	0	1
DFNDR_TEST3	allow		2	0	12	3
DFNDR_TEST2	allow		2	0	10	3
DFNDR_TEST	allow		2	0	8	3
DFNDR_StorageDemo	deny		2	0	2	0
DFNDR_Storage4	deny		0	0	5	0
DFNDR_Storage3	deny		0	0	1	0

Total Items: 43

7.4 Use PUT

In this example, rename your newly created role. You must do a PUT on the role, using its ID in the endpoint, and sending the updated JSON data structure.

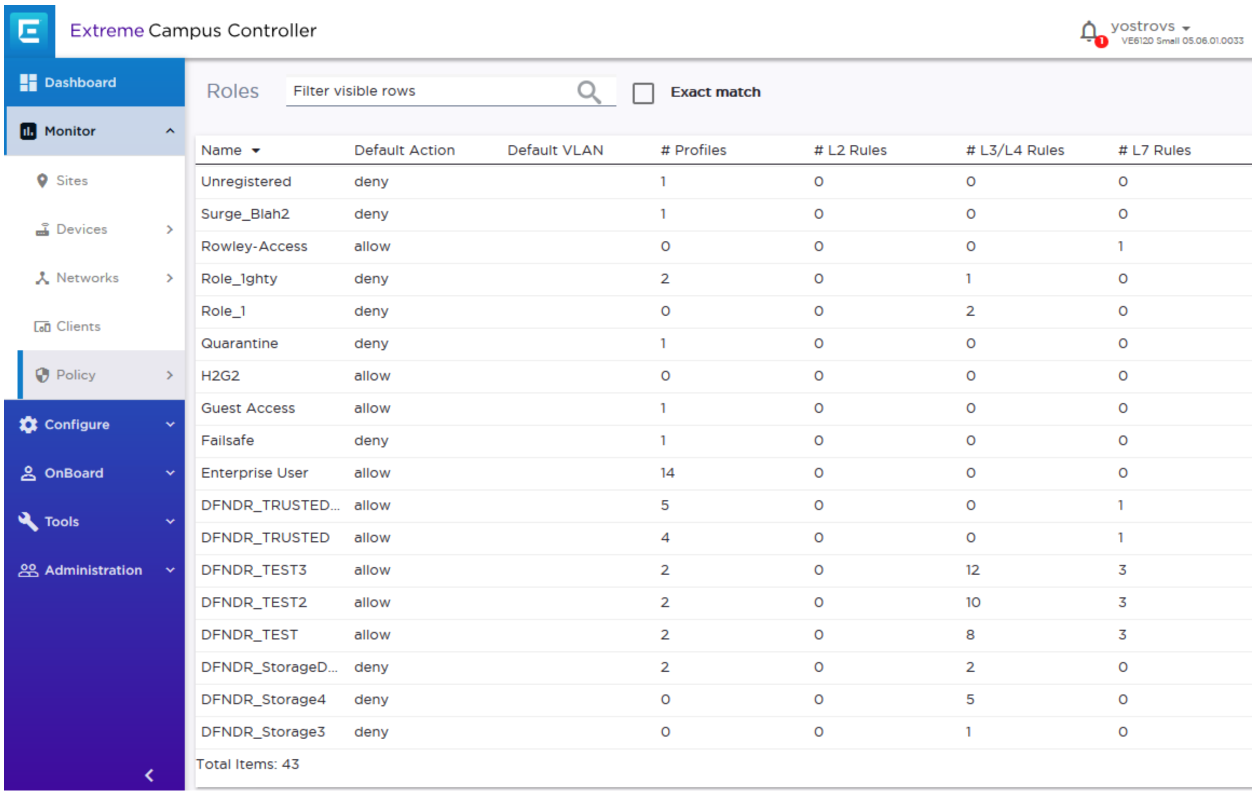
Make a slight modification to your code:

```
data = restGet('/management/v3/roles')
if data:
    print("\nThere are {} roles".format(len(data)))
    found = False
    for name in data:
        if name.get('name') == "Stef":
            print("The role Stef already exists")
            found = True
            role = name
            break
    if not found:
        role = {'name': 'Stef', 'defaultAction': 'allow', 'defaultCos': None}
        r = requests.post(baseURL + '/management/v3/roles', verify=False,
                          headers=myHeaders, json=role)
        if r.status_code != 201:
            print("Cannot access XCC REST API! Error code: {}".format(r.status_code))
            print(r.content)
            exit(0)
    else:
        role['name'] = "H2G2"

        r = requests.put(baseURL + '/management/v3/roles/{}'.format(role['id']),
                         verify=False, headers=myHeaders, json=role)
        if r.status_code != 200:
            print("Cannot access XCC REST API! Error code: {}".format(r.status_code))
            print(r.content)
        else:
            print("Role name changed")
```

The result:

```
C:\Extreme API with Python> xcc.py
There are 35 APs
[{'model': 'AP310', 'count': 1}, {'model': 'AP3917', 'count': 1}, {'model':
'AP460', 'count': 1}, {'model': 'AP3912', 'count': 3}, {'model': 'AP3935',
'count': 3}, {'model': 'AP410', 'count': 3}, {'model': 'AP510', 'count': 3},
{'model': 'AP3915', 'count': 4}, {'model': 'AP3916', 'count': 4}, {'model':
'AP505', 'count': 4}, {'model': 'SA201', 'count': 8}]
There are 43 roles
The role Stef already exists
Role name changed
```

In Extreme Campus Controller, you can confirm that the role name has been changed.



Name	Default Action	Default VLAN	# Profiles	# L2 Rules	# L3/L4 Rules	# L7 Rules
Unregistered	deny		1	0	0	0
Surge_Blah2	deny		1	0	0	0
Rowley-Access	allow		0	0	0	1
Role_1ghty	deny		2	0	1	0
Role_1	deny		0	0	2	0
Quarantine	deny		1	0	0	0
H2G2	allow		0	0	0	0
Guest Access	allow		1	0	0	0
Failsafe	deny		1	0	0	0
Enterprise User	allow		14	0	0	0
DFNDR_TRUSTED...	allow		5	0	0	1
DFNDR_TRUSTED	allow		4	0	0	1
DFNDR_TEST3	allow		2	0	12	3
DFNDR_TEST2	allow		2	0	10	3
DFNDR_TEST	allow		2	0	8	3
DFNDR_StorageD...	deny		2	0	2	0
DFNDR_Storage4	deny		0	0	5	0
DFNDR_Storage3	deny		0	0	1	0

Total Items: 43

7.5 Use DELETE

This example shows you how to delete the role you just created using DELETE. Add the instruction at the end of your code, as shown below:

```
data = restGet('/management/v3/roles')
if data:
    for name in data:
        if name.get('name') in ["Stef", "H2G2"]:
            print("Deleting role {}".format(name.get('name')))
            r = requests.delete(baseURL + '/management/v3/roles/{}'.format(name['id']),
                                verify=False, headers=myHeaders)
            print(r.content)
```

Run the script to see the following:

```
C:\Extreme API with Python> xcc.py
There are 35 APs
[{'model': 'AP310', 'count': 1}, {'model': 'AP3917', 'count': 1}, {'model':
'AP460', 'count': 1}, {'model': 'AP3912', 'count': 3}, {'model': 'AP3935',
'count': 3}, {'model': 'AP410', 'count': 3}, {'model': 'AP510', 'count': 3},
{'model': 'AP3915', 'count': 4}, {'model': 'AP3916', 'count': 4}, {'model':
'AP505', 'count': 4}, {'model': 'SA201', 'count': 8}]

There are 43 roles
Deleting role H2G2
b''
Deleting role Stef
b''
```

Because you executed the entire code, it created Stef again because Stef no longer existed after you renamed it to H2G2. The new piece of code found and deleted each new entry. Because this was a successful delete, no response was returned.