Python Scripting with XMC 8.1.3

For any NOS Products

	Version	Comments
Stéphane Grosjean	0.1	Initial Release – October 2017
Stéphane Grosjean	0.5	Creation of example scripts for Automated Campus, various fixes
Stéphane Grosjean	0.9	Update with 8.1.2 changes + NBI GraphQL
Stéphane Grosjean	0.91	Ending prompt of cli_output is now removed automatically
Stéphane Grosjean	0.92	Fixing some typos in text and code examples
Stéphane Grosjean	0.93	Adding 3rd parameter info for emc_cli.send()
Stéphane Grosjean	0.94	Fixing the cli output as 8.1.3 changes it

Stéphane Grosjean
Principal SE, EMEA Southern, France
stgrosjean@extremenetworks.com

Extreme
Connect Beyond the Network

Table of Contents

1 Disclaimer

This document is **internal only** and shouldn’t be used externally by any means. This is not an official document from Extreme Networks and cannot be used to validate any design, feature or scalability. This is an informational document only.

1.1 References

The following documents were used extensively in the preparation of this document:

Python jsonrpc.py & restconf.py class by Dave Hammers
Long list of emails with Lou ☺

2 Introduction

Starting with XMC 8.0.4.54, a new language is available for scripting: Python. Its inclusion doesn't mean the previous scripting programming language, TCL, is being replaced, but with Python new possibilities are made available for the users.

The different options for scripting in XMC are now the following:

- TCL scripting
- JSONRPC scripting
  - CLI method: sends CLI commands over HTTPS to an EXOS-capable switch (EXOS 21.1 or later)
  - Python method: execute a Python script remotely to an EXOS-capable switch (EXOS 21.1 or later)
- Python Scripting

Note: ISW products do support JSONRPC too, but they are using a different method, not compatible with EXOS. At the moment, there's no plan to add that method to XMC.

The Python Engine embedded in XMC 8.0.4 is running python scripts inside the java jvm (XMC is currently using Jython). As this is not a full blown CPython, some external python modules that you would like to add may not work. The version used is Jython 2.7.6.

Note: There's a plan to move the Python Engine external to XMC and use CPython, running the latest Python 2.7 version. This is more of a Plan of Intent at the moment, most likely for the XMC 9.x timeframe.

For the most part, from the XMC Scripting GUI you can simply create a new script, selecting Python as the programming language, and write it just like you would have normally. What XMC Python Engine brings you is a powerful set of tools to interact with XMC environment and variables, and run against devices selected from XMC topology view, for example. You have the possibility to add home-made features to XMC, in a way.

The three main global objects available to the user are the following:

- emc_vars
- emc_cli
- emc_nbi

2.1 EMC_VARS

The **emc_vars** global object is a python dictionary containing all the global variables previously accessible to TCL scripts. So this is the same variables used with TCL, and as such the data returned may not always been the most appropriate for Python (string instead of list, for example). They are accessible for convenience, but the goal with the Python engine will be to use the coming NBI. The following are all the variables available:

serverIP	server IP address
serverVersion	server version
serverName	server host name
time	current time at server (HH:mm:ss z)
date	current date at server (yyyy-MM-dd)
userName	EMC user name
userDomain	EMC user domain name
auditLogEnabled	true/false if audit log is supported
scriptTimeout	max script timeout in secs
scriptOwner	scripts owner
deviceName	DNS name of selected device
deviceIP	IP address of the selected device
deviceId	device DB ID
deviceLogin	login user for the selected device
devicePwd	logn password for the selected device
deviceSoftwareVer	software image version number on the device
deviceType	device type of the selected device
deviceSysOid	device system object id
deviceVR	device virtual router name
cliPort	telnet/ssh port
isExos	true/false. Is this device an EXOS device?
family	device family name
vendor	vendor name
deviceASN	AS number of the selected device
port	selected ports
vrName	selected port(s) VR name
ports	all device ports
accessPorts	all ports which have config role access
interSwitchPorts	all ports which have config role interswitch
managementPorts	all ports which have config role management

As this is a python dictionary, a script can simply read the value of any of them, when required, just like any typical python dictionary. Below is an example:

```
myVar = emc_vars["deviceIP"]
```

Many of these variables are self-explanatory. One variable, however, requires a bit more of attention: `port`.

When used in a script, it will prompt the user to select some ports that will be used by the script. This variable will return a string containing all the ports, comma-separated. For Python scripting, manipulating a list seems more adequate, so you may want to add a function similar to the following in your code:

```
# transforms a string into a list and returns it.
def string2list(inputString):
    return inputString.split(",")
```

It is worth to note that the variable `isExos` is a legacy variable and there's no plan to add other similar variable for other NOS. The plan, going forward, is to use device data that will come from the NBI.

All the `device<X>` variables are certainly the most useful for script development. The `deviceLogin` and `devicePwd` provides the credential to access the switch: they are known from the CLI Profile associated to the device.

Note: By default, every script run in the context of a device, so executing the same script against multiple devices with different CLI credentials shouldn't be an issue. If the script is correctly coded, it should also be able to handle different NOS and CLI syntax.

The `deviceSoftwareVer` variable will not return the patch level version of an EXOS switch, only the first four digits (ie: 22.4.1.4). Some variables may be left blank.

2.2 EMC_CLI

The **emc_cli** is a Python object used to execute CLI commands. This object uses the same internal CLI session objects than TCL.

This object is used a bit differently with Python: a Boolean allows to choose to wait for system/shell prompt, or not. Setting the Boolean value to False, no cli output is returned. The Boolean value is optional and defaults to True. A third optional parameter is a timer, in seconds, to wait for information if needed.

Several methods are available with this python object, to retrieve several information from the 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/echoed back from the device (including cli command prompt) as a string

The `isSuccess()` doesn't tell if the CLI command was successful or not, but if the `send()` has been completed correctly. Whatever is the result of that CLI command is left to the script to handle, by analyzing the CLI output.

Below is an 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 that example, EXOS is the NOS. However, this is not restricted to that NOS, and any other NOS is eligible, as long as the device is accessible from XMC with a correct CLI Profile.

The `emc_cli` object will connect to the device using either telnet or ssh, so any device from any vendor is accessible. However, the login banners and sub-prompts can vary a lot from one vendor to another. EMC has a list of many 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 than the `CLIRules.xml` file (caution, 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 than the `CLIRules.xml` file. Typically, BOSS and VOSS access uses this file as well.

Note: CLI scripting for BOSS and VOSS is very inconsistent. Those devices have all kinds of different banners during logins and subprompts that are very different. Make sure that the CLI profile for those devices is correct, emc_cli relies on the CLI profile that is set for that device. By default, it will try to use the regular expressions defined in `CLIRules.xml` *under the "Avaya" section, but not all commands and/or prompts have been added so if your CLI profile is correct and the script fails this might be the reason.*

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

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

The cliRule name normally will come from Vendor Profiles which each device (family, subfamily or device type) will or should have a property called cliRuleFileName (name is misleading, it's really the cliRuleName, not a file name).

Note: The cliRuleName can also be set dynamically from Python by invoking *`emc_cli.setCliRule`.*
For example:
```
# must be called before using emc_cli.send
emc_cli.setCliRule("ruleName")
```

Tip: If a specific command prompts the shell for an input, breaking a script to execute properly, you can add to myCLIRules.xml *that prompt, for the correct platform, and specify the desired answer to it. For example:*

```
<CommandPrompt command=".*">
  <defaultPrompt>
    <prompt>^Do you want to continue \(y/n\)</prompt>
    <reply>y</reply>
  </defaultPrompt>
</CommandPrompt>
```

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

Note: Since XMC 8.0.4 and up to XMC 8.1.1, the string returned was also including the trailing CLI prompt. XMC 8.1.2 removed it, and XMC 8.1.3 brings it back. So existing Python scripts may need to be updated.

One way to get rid of that extra information is to create a function, similar to this one:

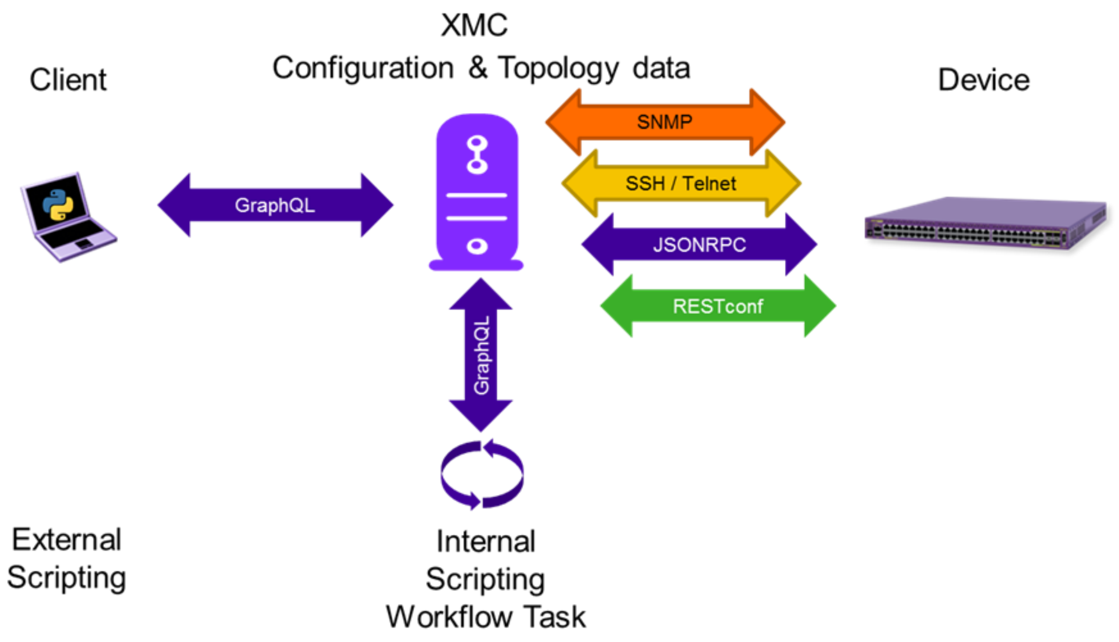
```
# transforms the string into a list, removes first and last entries
# if running with XMC 8.1.1 or before, or just first line,
# returns the result as a string keeping the carriage return
# returns None if anything goes wrong
def getOutputOnly(inputStrings):
    try:
        version = ''.join(emc_vars["serverVersion"].split('.')[:3])
        pivotVersion = ''.join("8.1.2".split('.'))
        if int(version) == int(pivotVersion):
            lines = inputStrings.splitlines()[1:]
        else:
            lines = inputStrings.splitlines()[1:-1]
        return '\n'.join(lines)
    except:
        return None
```

2.3 EMC_NBI

The **emc_nbi** global object is a client API into XMC northbound interface. With this API, Python scripts have access to virtually all the data XMC manages and it's powered by GraphQL-based queries and returns the data as json, making it a very flexible and powerful solution for advanced scripting.

Note: GraphQL is a query language developed by Facebook, before becoming public in 2015. Just like REST, it accesses data via an HTTP GET and receives the outcome in JSON format. But one fundamental difference is that using GraphQL, the client can receive only the data it needs, not all the data available. To manipulate large amount of data in databases, this can be very important and way more efficient.

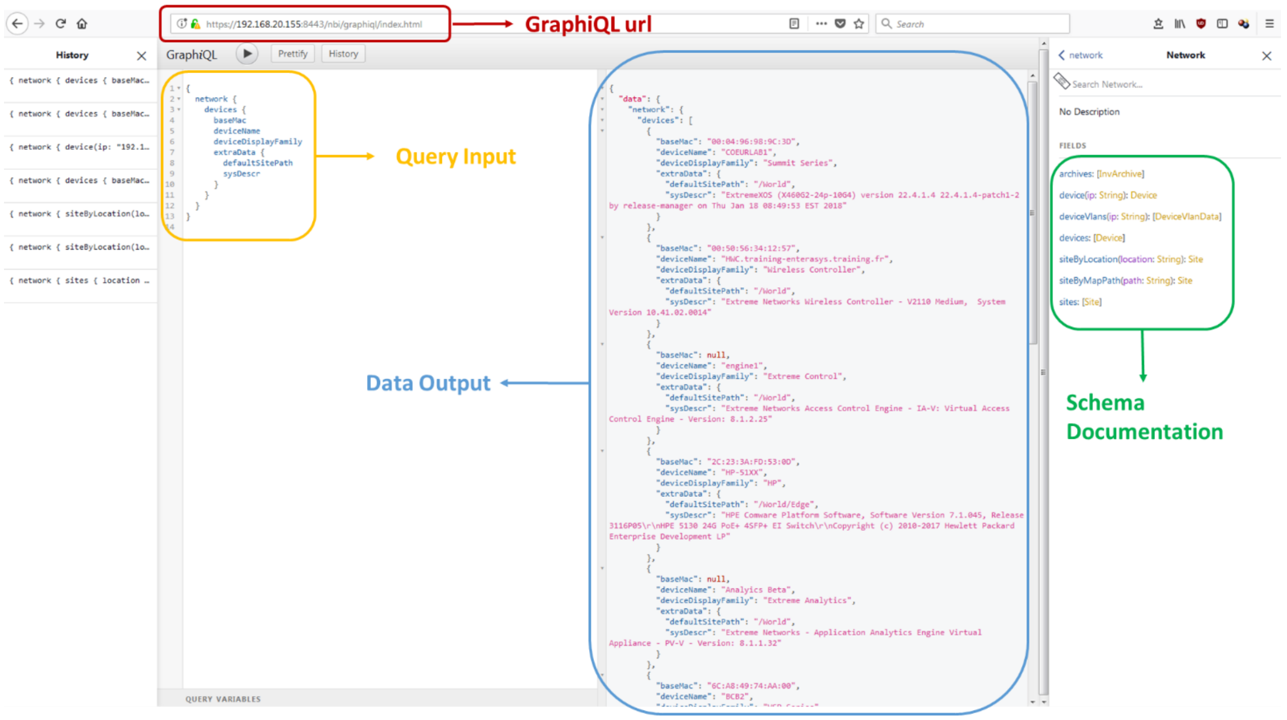
The GraphQL query language becomes central to XMC interaction with the devices, scripting and workflows. It can be accessed either internally via a Python script, or externally as well to interact with any third-party application.



2.3.1 Queries

Using the GraphQL NBI requires to create a query to retrieve an information. Just like REST, any query is well-defined and the programmer knows precisely the format of the query.

For ease of programming, XMC 8.1.2 integrates GraphiQL, an interface to allow the user to test queries and see the output. It can be accessed using the following url on a given XMC server: https://<xmc server IP>:8443/nbi/graphiql/index.html


https://192.168.20.155:8443/nbi/graphiql/index.html
GraphiQL url
Search
History
{ network { devices { baseMac…
{ network { devices { baseMac…
{ network { device(ip: "192.1…
{ network { devices { baseMac…
{ network { siteByLocation(lo…
{ network { siteByLocation(lo…
{ network { sites { location …
GraphiQL
Prettify
History
Query Input
Data Output
QUERY VARIABLES
network
Network
Search Network...
No Description
FIELDS
Schema Documentation

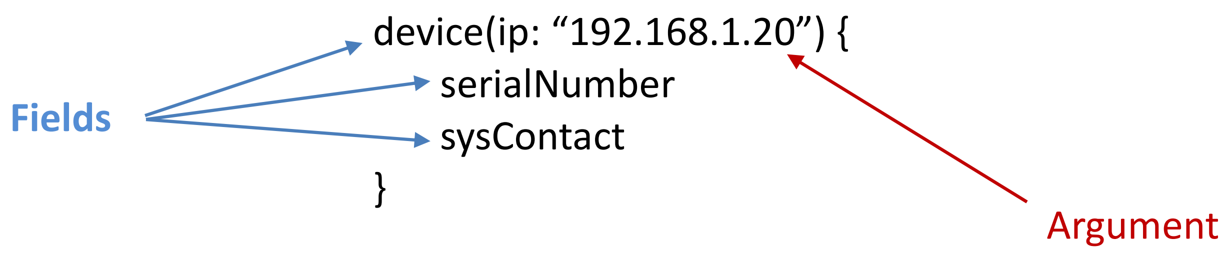
```
{
  network {
    devices {
      baseMac
      deviceName
      deviceDisplayFamily
      extraData {
        defaultSitePath
        sysDescr
      }
    }
  }
}
```

```
{
  "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, and this is the only operation supported as of XMC 8.1.2. In a future release of XMC, mutation (write) is planned to be supported as well.

A query is a string, that can be formatted as a JSON object. Central to 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 that is used for the various entry point is ExtremeApi. From there, the subsystem that can be used for queries are:

```
ExtremeApi {
    accessControl,
    administration,
    network,
    policy,
    wireless,
    workflows
}
```

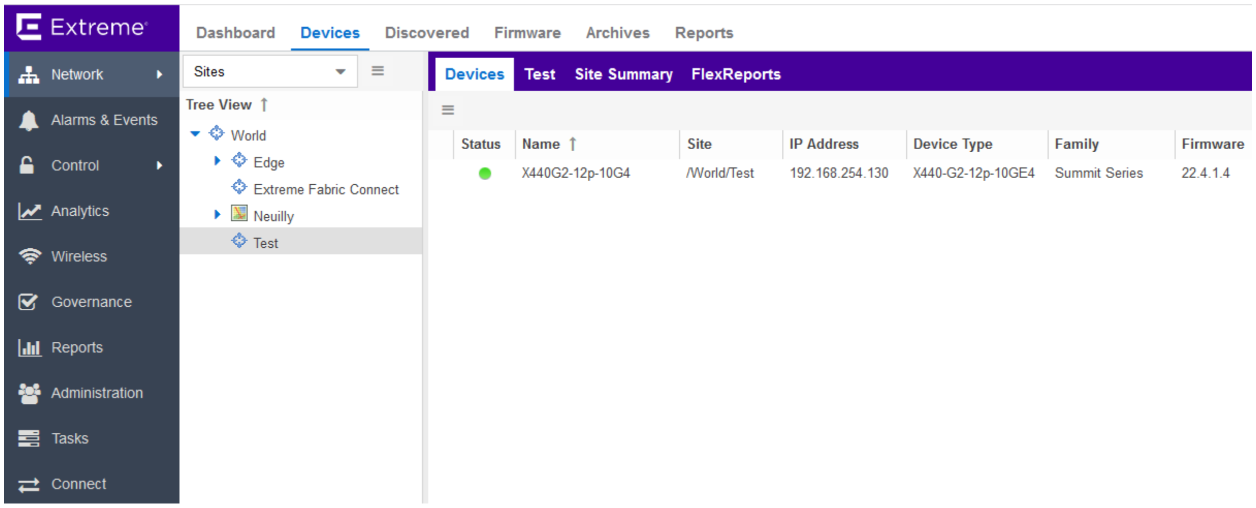
The GraphQL schema description can be accessed ever as an IDL file or a JSON file, using the following urls on an XMC server:

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

2.3.2 Query Examples

Following are some examples, using Python executed from XMC, to send a query to the NBI and use the returned information.

From XMC, if we naviguate to the Network -> Devices menu and select the Sites view, we can see the following tree view on our server:



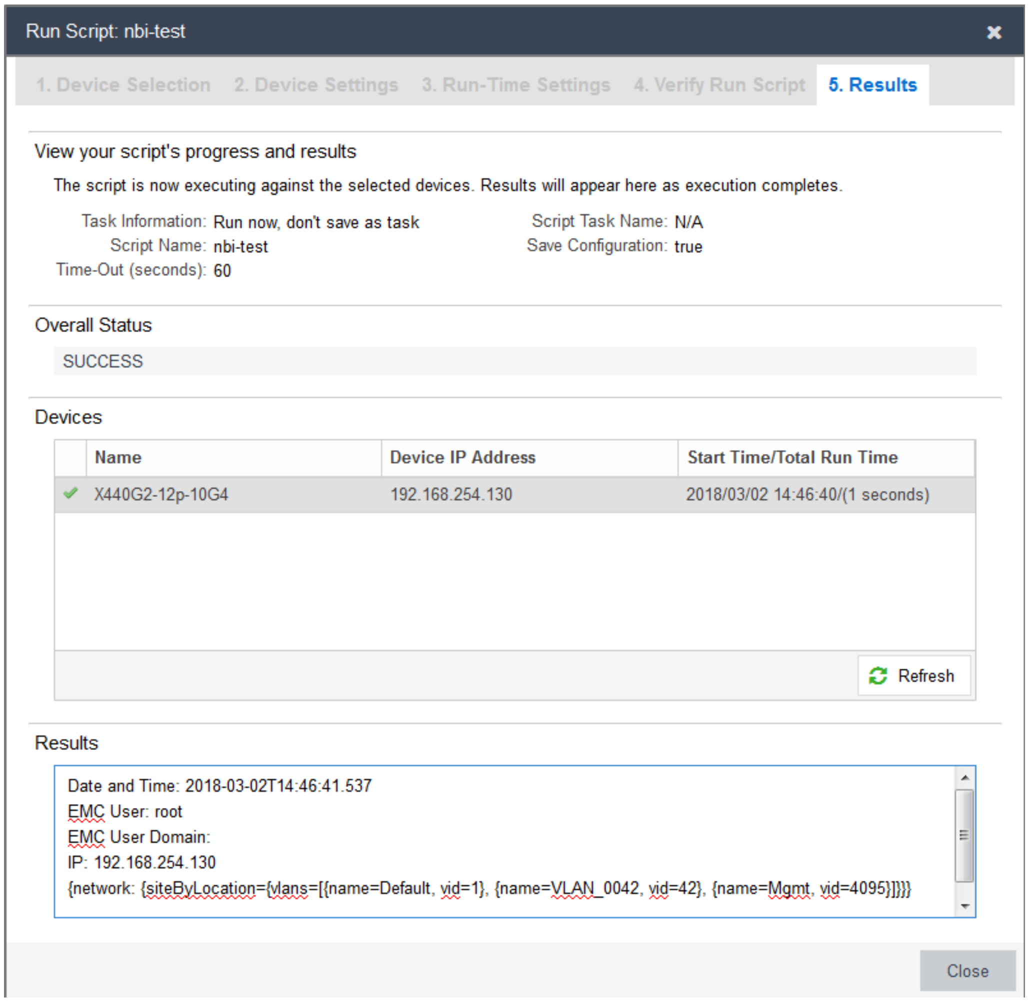
Status	Name	Site	IP Address	Device Type	Family	Firmware
●	X440G2-12p-10G4	/World/Test	192.168.254.130	X440-G2-12p-10GE4	Summit Series	22.4.1.4

Let's query XMC about that switch in the /World/Test site.

A GraphQL query is a string. Using GraphiQL, we can find and test queries. Let's query the name and vid of any VLAN on that switch. Here's a very simple Python script that we can run from XMC.

```
nbiQuery = '''
{
  network {
    siteByLocation(location: "/World/Test") {
      vlans {
        name
        vid
      }
    }
  }
}
'''

result = emc_nbi.query(nbiQuery)

print result
```

If we execute it, here's the result:



View your script's progress and results

The script is now executing against the selected devices. Results will appear here as execution completes.

Task Information: Run now, don't save as task
Script Name: nbi-test
Time-Out (seconds): 60
Script Task Name: N/A
Save Configuration: true

Overall Status

SUCCESS

Devices

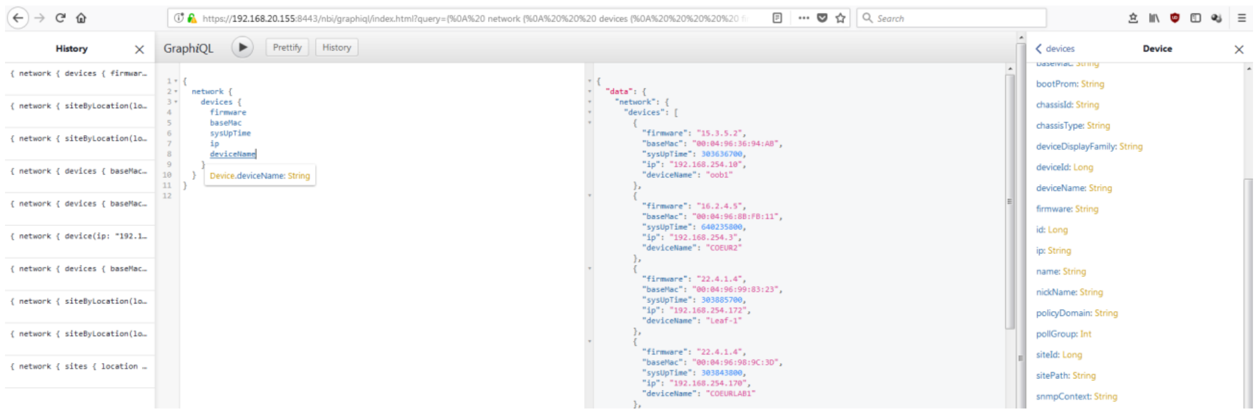
	Name	Device IP Address	Start Time/Total Run Time
✓	X440G2-12p-10G4	192.168.254.130	2018/03/02 14:46:40/(1 seconds)

Results

```
Date and Time: 2018-03-02T14:46:41.537
EMC User: root
EMC User Domain:
IP: 192.168.254.130
{network: {siteByLocation={vlans=[{name=Default, vid=1}, {name=VLAN_0042, vid=42}, {name=Mgmt, vid=4095}]}}}
```

Note: As of XMC 8.1.3, it is still necessary to run a script along with a device, even if that script is not in relation to a device. This should be fixed in a future release of XMC.

Here are a few other examples, browsing the GraphiQL url to find more possibilities.


https://192.168.20.155:8443/nbi/graphiql/index.html?query={%0A%20 network {%0A%20%20%20 devices {%0A%20%20%20%20%20
Search
History
GraphiQL
Prettify
History
devices
Device

```
{ network { devices { firmwar…
{ network { siteByLocation(lo…
{ network { siteByLocation(lo…
{ 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 {
      firmware
      baseMac
      sysUpTime
      ip
      deviceName
    }
  }
}
```

Device.deviceName: String

```
{
  "data": {
    "network": {
      "devices": [
        {
          "firmware": "15.3.5.2",
          "baseMac": "00:04:96:36:94:AB",
          "sysUpTime": 303636700,
          "ip": "192.168.254.10",
          "deviceName": "oob1"
        },
        {
          "firmware": "16.2.4.5",
          "baseMac": "00:04:96:8B:FB:11",
          "sysUpTime": 640235800,
          "ip": "192.168.254.3",
          "deviceName": "COEUR2"
        },
        {
          "firmware": "22.4.1.4",
          "baseMac": "00:04:96:99:83:23",
          "sysUpTime": 303885700,
          "ip": "192.168.254.172",
          "deviceName": "Leaf-1"
        },
        {
          "firmware": "22.4.1.4",
          "baseMac": "00:04:96:98:9C:3D",
          "sysUpTime": 303843800,
          "ip": "192.168.254.170",
          "deviceName": "COEURLAB1"
        },
```

bootProm: String
chassisId: String
chassisType: String
deviceDisplayFamily: String
deviceId: Long
deviceName: String
firmware: String
id: Long
ip: String
name: String
nickName: String
policyDomain: String
pollGroup: Int
siteId: Long
sitePath: String
snmpContext: String

```
nbiQuery = '''
{
  network {
    devices {
      firmware
      baseMac
      sysUpTime
      ip
      deviceName
    }
  }
}
'''

result = emc_nbi.query(nbiQuery)

for item in result['network']['devices']:
    print item['ip'] + ': \t' + item['deviceName'] + ' \t' + item['firmware'] + ' \t'
+ str(item['sysUpTime'])
```

We can also query multiple information from ExtremeApi at the same time.

```
nbiQuery = '''
{
  accessControl {
    appliances {
      ipAddress
      displayName
    },
    engineStatus
  },
  network {
    siteByLocation(location: "/World/Extreme Fabric Connect") {
      ospf
      vxlan
    },
    devices {
      sysName
    }
  }
}
```

```
'''

result = emc_nbi.query(nbiQuery)

for item in result['accessControl']['appliances']:
    print item['ipAddress'] + ' \t' + item['displayName']

for item in result['network']['devices']:
    print item['sysName']

print result['accessControl']['engineStatus']
```

2.3.2.1 *Using templates*

Some of the queries require arguments. An elegant way to manage them in a Python script is to use templates.

We can build a Python class, so that the client can focus only on the API and not the query.

```
from string import Template

class Device:
   def __init__(self,ip=None):
      self.ip = ip
      if ip == None:
        self.ip = emc_vars["deviceIP"]

   def get_facts(self):
      query_tpl='''query ExtremeApi {
         network {
             device(ip:"${deviceIP}") {
                 assetTag
                 baseMac
                 bootProm
                 chassisId
                 chassisType
                 deviceDisplayFamily
                 deviceName
                 firmware
                 ip
                 name
                 nickName
                 policyDomain
                 snmpContext
                 sysContact
                 sysLocation
                 sysName
                 sysObjectId
                 sysUpTime
                 maintenance
                 unknown
                 up
                 upSnmpError
            }
            }
        }'''
        query = Template(query_tpl).safe_substitute(dict(deviceIP=self.ip))
        response = emc_nbi.query(query);
        if response:
            return response["network"]["device"]
        else:
            return None

device = Device("192.168.254.170");
results = device.get_facts();
print results
```

We can leverage that by creating a separate module with our class, that we may then import in future scripts.

Let's save the class as a python script, and put it into the following location:

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

Then, we can call it from any Python script. Assuming our file is called MyDevice.py, here's an example of how to use it:

```
from MyDevice import Device
import MyDevice

MyDevice.emc_nbi = emc_nbi

device = Device("192.168.254.170");
results = device.get_facts();
print results
```

Because the **emc_** variables are scoped to the module and are not global variables, we have to export these to the module that need/use it. This is the reason why we have to add the line: `MyDevice.emc_nbi = emc_nbi`.

2.3.2.2 Using External Scripts

Let's present a last example of accessing XMC via GraphQL, but this time from an external program.

This python script is written under a Windows 7 system, with Python 3.5.2.

```
#!/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'])
```

Here’s the output of such script, running from a laptop:

```
C:\05 - Trainings\_Guides\Python with XMC>python external-nbi.py
Enter remote system username: root
Remote system password:
Enter IP of the XMC server: 192.168.20.155
192.168.254.10  oob1
192.168.254.3   COEUR2
192.168.254.172         Leaf-1
192.168.254.170         COEURLAB1
192.168.254.173         SPINE1
192.168.20.250  HWC.training-enterasys.training.fr
192.168.254.2   COEUR1
192.168.20.154  engine1
192.168.254.108         HP-51XX
192.168.20.153  Analyics Beta
192.168.254.111         BCB2
192.168.254.109         BEB3
192.168.254.110         BCB1
192.168.254.113         BEB2
192.168.254.112         BEB1
192.168.254.114
192.168.30.114  VX9-NSIGHT
192.168.254.130         X440G2-12p-10G4
```

3 Enhancing the Default Python Engine

Starting with XMC 8.0.4, the python engine runs python in java jvm, using Jython 2.7.6. Most, if not all, of the standard Python 2.7 library should be available, but if you need to add a new module, this can be accomplished.

Since XMC 8.1.2, the directory structure has changed for the python modules location, both for the default ones and users-based. Also, the requests module and pip utility are installed by default.

3.1 Default Location for Scripts

When a user creates or modify a script under the XMC UI, the script is saved in the following location:

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

3.2 Adding a User Script

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

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

From the embedded python scripts, simply 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.

3.3 Legacy Python/TCL Scripts

Legacy Python/TCL scripts are shipped with XMC under the following location:

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

3.4 Python Modules Shipped with XMC

With XMC 8.1.2, all the modules shipped with XMC are located into the following location:

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

This is where default jsonrpc.py and restconf.py are located.

3.5 System Path and Precedence

All 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` would be used first.

3.6 Installing a Library

To install a library, the easiest way is to use pip. Starting with XMC 8.1.2, pip utility is part of the default XMC server installation. Using the pip utility should be done that way:

```
cd /usr/local/Extreme_Networks/NetSight/jython/bin
export JAVA_HOME=/usr/local/Extreme_Networks/NetSight/java
./pip install <module>
```

3.7 Using JSONRPC Capability for EXOS

From the default XMC Python Engine, you can only access switches from telnet or ssh. If you are planning to use scripts with switches running EXOS 21.1 (or later), a great alternative is to use, instead, JSONRPC.

Starting with XMC 8.1.2, jsonrpc.py (version 2.0.0.3) is installed by default. You can simply use it from any Python script. You can check for a newer version on Extreme Networks' GitHub:

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

As an example, a generic python script using jsonrpc would look like:

```
from jsonrpc import JsonRPC

def main():
    # open a session with the switch
    jsonrpc = JsonRPC(emc_vars["deviceIP"], username=emc_vars["deviceLogin"],
password=emc_vars["devicePwd"])

    # send a CLI command and save the result in a variable
    response = jsonrpc.cli('show vlan')

    print response


main()
```

A major benefit of using JSONRPC is that for a common CLI command, both the typical CLI Output will be sent back, but also the JSON equivalent of that command. This part is the same output than running in command line the cli2json.py python script for that CLI command.

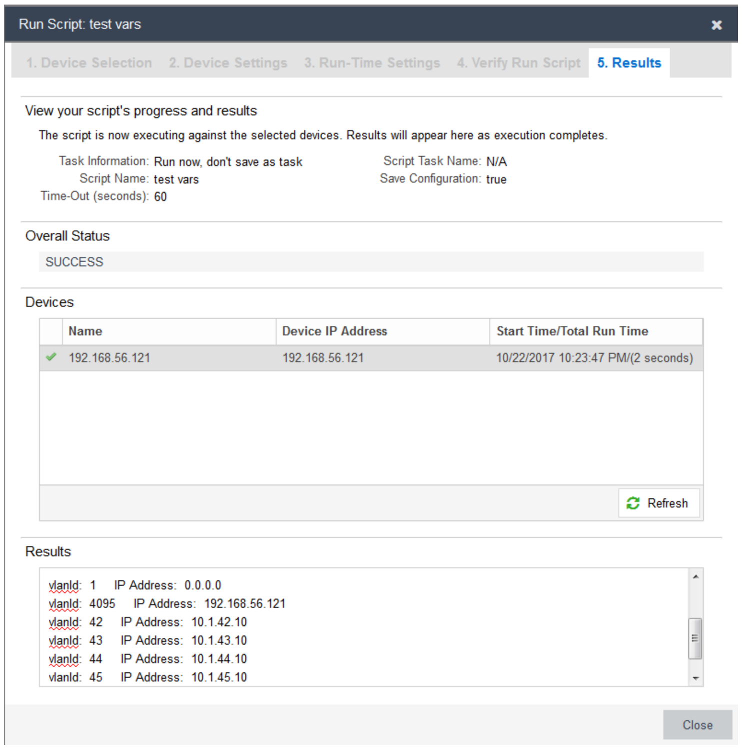
Note: The cli2json.py script is part of EXOS since version 15.6.

To use the json output, you need to know what key to use. But for a same command, this will always be the same key, so a script is easy to write. No more screen scraping, for most of the commands.

Below is an example, to easily retrieve VLAN tag and IP address of the VLANs on a switch.

```
from jsonrpc import JsonRPC

jsonrpc = JsonRPC(emc_vars["deviceIP"], username=emc_vars["deviceLogin"],
password=emc_vars["devicePwd"])

response = jsonrpc.cli('show vlan')
data = response.get('result')

for row in data:
    vlaninfo = row.get('vlanProc')
    if vlaninfo:
        print "vlanId: ", vlaninfo.get('tag'), "\tIP Address: ",
vlaninfo.get('ipAddress')
```

The important key to know is "vlanProc". If you look at the various information in it, looking at a global print of the response, you can find what you need and just get it. The result of this script, when we run the script on a switch (or VM), is shown below.



View your script's progress and results

The script is now executing against the selected devices. Results will appear here as execution completes.

Task Information: Run now, don't save as task
Script Name: test vars
Time-Out (seconds): 60
Script Task Name: N/A
Save Configuration: true

Overall Status

SUCCESS

Devices

Name	Device IP Address	Start Time/Total Run Time
✓ 192.168.56.121	192.168.56.121	10/22/2017 10:23:47 PM/(2 seconds)

Results

```
vlanId:  1    IP Address:  0.0.0.0
vlanId:  4095    IP Address:  192.168.56.121
vlanId:  42    IP Address:  10.1.42.10
vlanId:  43    IP Address:  10.1.43.10
vlanId:  44    IP Address:  10.1.44.10
vlanId:  45    IP Address:  10.1.45.10
```

Note: This example only illustrates the cli method. The jsonrpc.py class also provides way to use python method and runscript method. The runscript method is about executing a Python script on a remote switch, when the python script is local to the server. The python method is similar but runs under the expy context, just like Python Apps.

3.8 Using RESTConf Capability for EXOS

With XMC 8.1.2, a RESTConf python class is provided by default to create python scripts using that API. The version included with XMC 8.1.2 is version 1.1.0.3. You can look for a newer version on Extreme Networks' Github:

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

EXOS supports RESTCONF natively starting with release 22.4, following the Openconfig model. It can be backported down to EXOS 22.1, by installing the restconf.pyz module. At the time of writing of this document, the latest version is restconf_xos_1.0.1.30.pyz, available here:

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

Note: How to install the restconf module? Several ways are possible. Either you directly reference the url, or you download it to your tftp server.

```
# download url
https://github.com/extremenetworks/EXOS_Apps/blob/master/REST/download
s/restconf_xos_1.0.1.30.lst
```

or

```
# download url tftp://<ip>/restconf_xos_1.0.1.30.lst
```

or using Chalet.

Make sure DNS is correctly configured on the switch to make the url work.

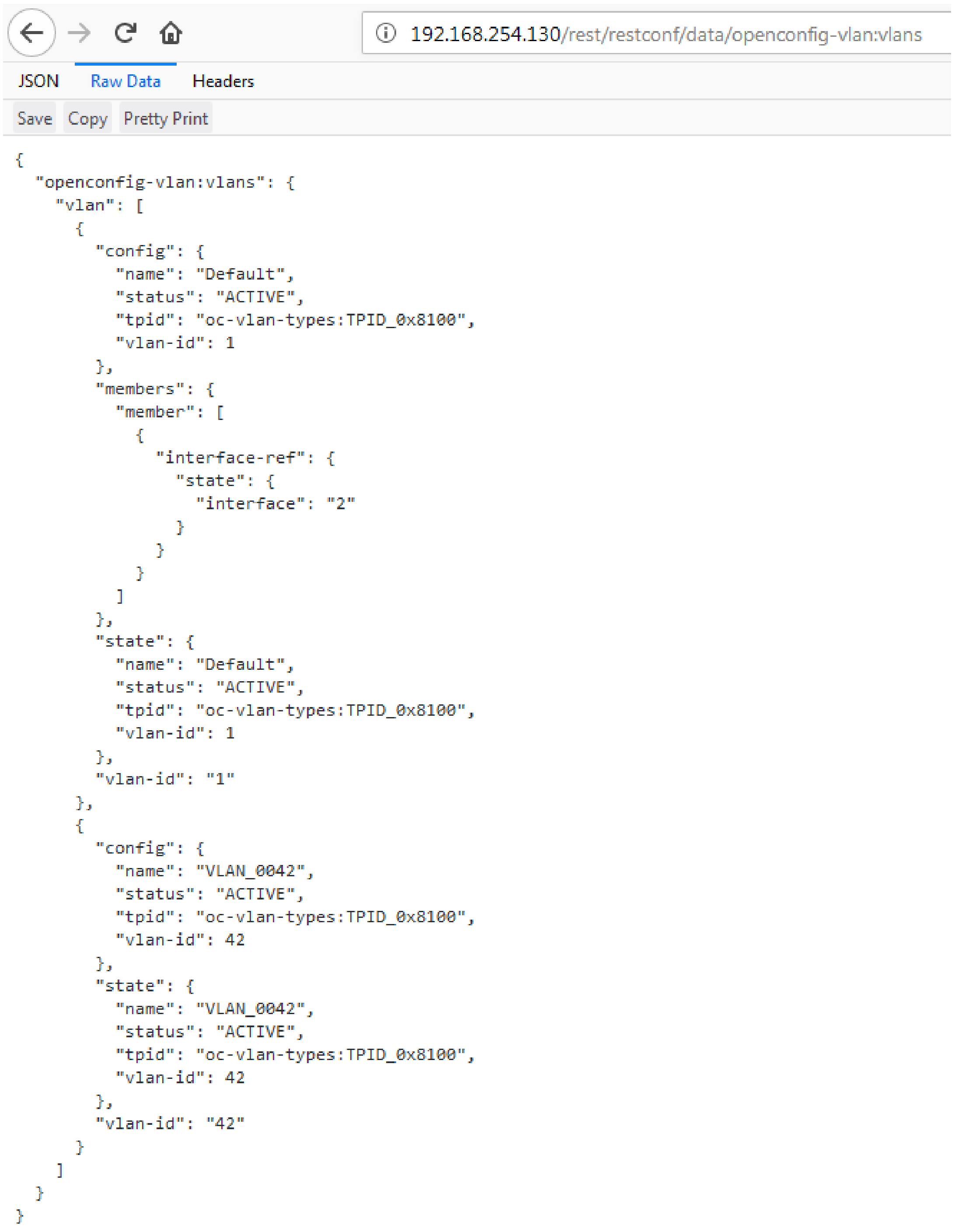
```
# config dns-client add name-server 192.168.20.83 vr VR-Mgmt
```

Note: With EXOS 22.5, .lst file will be supported with the download image CLI command.

```
# download image <ip> restconf_xos_1.0.1.30.lst
```

Below is an example of how to retrieve the VLAN list using RESTConf. The information we are looking for is related to the VLANs. This is the data model we want to browse: `openconfig-vlan:vlans`.

We can read it directly from the switch using any browser, by browsing the following url: http://<switch IP>/rest/restconf/data/openconfig-vlan:vlans



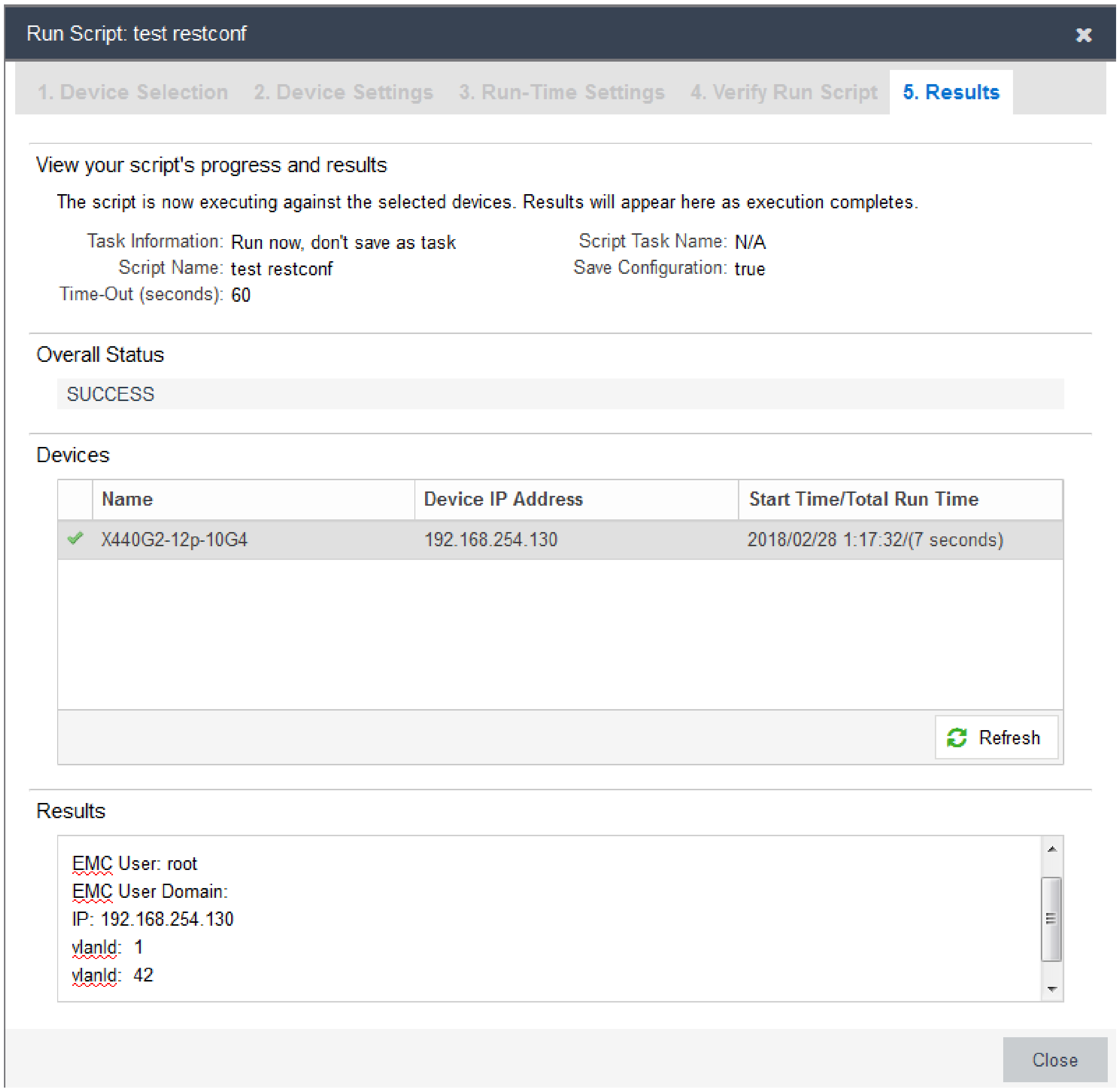
```
{
  "openconfig-vlan:vlans": {
    "vlan": [
      {
        "config": {
          "name": "Default",
          "status": "ACTIVE",
          "tpid": "oc-vlan-types:TPID_0x8100",
          "vlan-id": 1
        },
        "members": {
          "member": [
            {
              "interface-ref": {
                "state": {
                  "interface": "2"
                }
              }
            }
          ]
        },
        "state": {
          "name": "Default",
          "status": "ACTIVE",
          "tpid": "oc-vlan-types:TPID_0x8100",
          "vlan-id": 1
        },
        "vlan-id": "1"
      },
      {
        "config": {
          "name": "VLAN_0042",
          "status": "ACTIVE",
          "tpid": "oc-vlan-types:TPID_0x8100",
          "vlan-id": 42
        },
        "state": {
          "name": "VLAN_0042",
          "status": "ACTIVE",
          "tpid": "oc-vlan-types:TPID_0x8100",
          "vlan-id": 42
        },
        "vlan-id": "42"
      }
    ]
  }
}
```

Note: You need to know what url to use. To have a list of all the yang data models supported, you can browse the following url, as defined by RFC 8040:
http://<switch IP>/rest/restconf/data/ietf-yang-library:modules-state

Let’s use a Python script to list all the VLANs.

```
import json
from restconf import Restconf

def get_data(restobj, url):
    result = restobj.get(url)
    return result.json()

if emc_vars["isExos"]:
    restconf = Restconf(emc_vars["deviceIP"], emc_vars["deviceLogin"],
emc_vars["devicePwd"])

    obj = get_data(restconf, '/data/openconfig-vlan:vlans')
    data = obj.get('openconfig-vlan:vlans')
    vlans = data.get('vlan')

    if vlans:
        for row in vlans:
            vlan = row.get("vlan-id")
            if vlan:
                print "vlanId: ", vlan

else:
    print "Need an EXOS switch running 22.4 or later"
```

We can see the result on a switch running EXOS 22.4.1.4-patch1-2:



View your script's progress and results

The script is now executing against the selected devices. Results will appear here as execution completes.

Task Information:	Run now, don't save as task	Script Task Name:	N/A
Script Name:	test restconf	Save Configuration:	true
Time-Out (seconds):	60		

Overall Status

SUCCESS

Devices

	Name	Device IP Address	Start Time/Total Run Time
✓	X440G2-12p-10G4	192.168.254.130	2018/02/28 1:17:32/(7 seconds)

Results

```
EMC User: root
EMC User Domain:
IP: 192.168.254.130
vlanId:  1
vlanId:  42
```

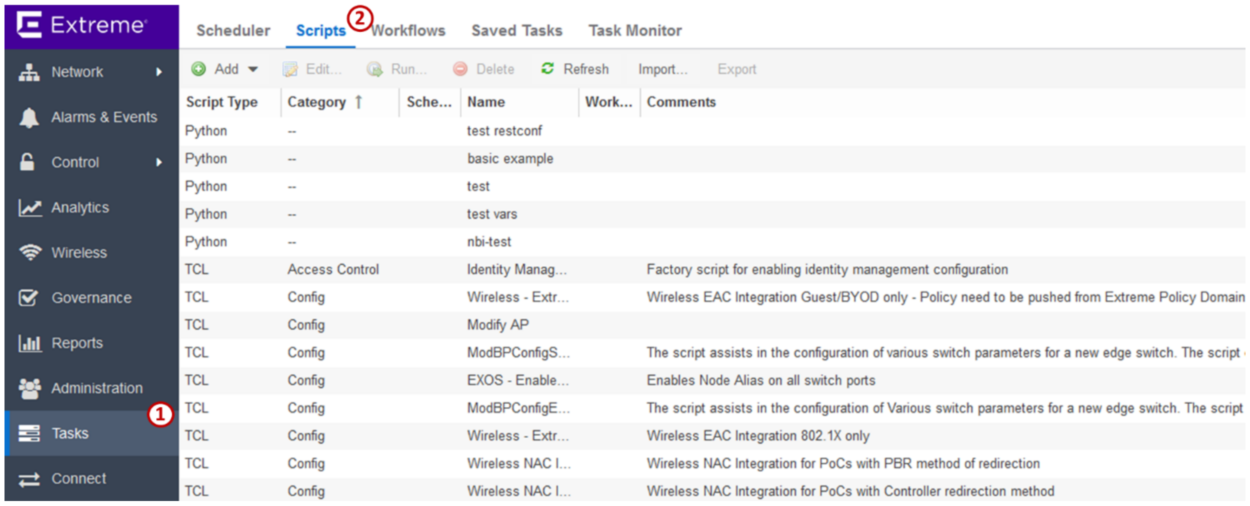
Note: All these examples are using HTTP, for simplicity. They work also with HTTPS, as long as SSL has been configured and enabled on the switch.

4 Examples

Starting with XMC 8.0.4, Python scripting is available for scripting. With XMC 8.1.2, some enhancements have been made, such as the metadata fields.

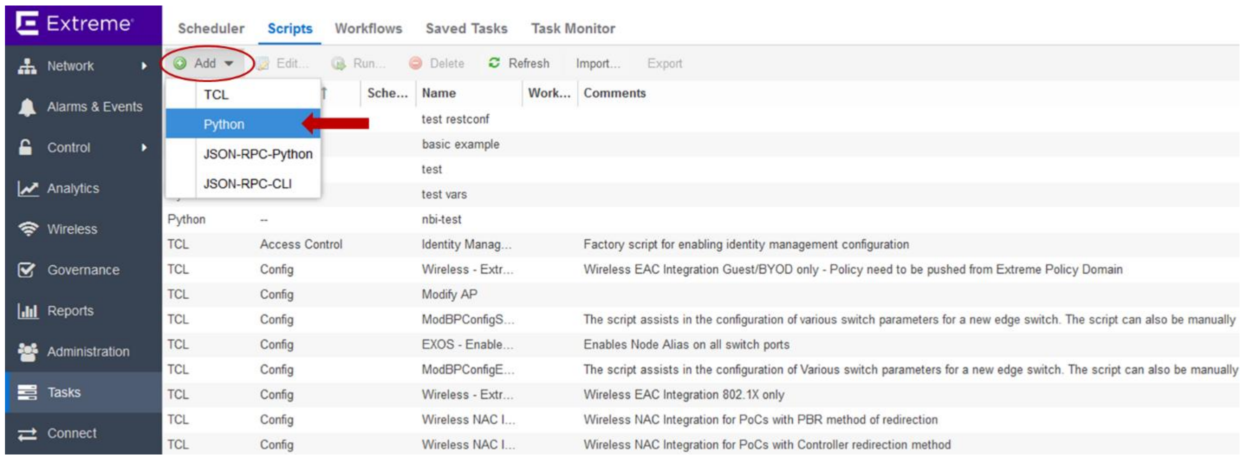
4.1 Getting Started

From XMC GUI, you access the Scripting Engine via the following menus: Tasks -> Scripts.



Script Type	Category ↑	Sche...	Name	Work...	Comments
Python	--		test restconf		
Python	--		basic example		
Python	--		test		
Python	--		test vars		
Python	--		nbi-test		
TCL	Access Control		Identity Manag...		Factory script for enabling identity management configuration
TCL	Config		Wireless - Extr...		Wireless EAC Integration Guest/BYOD only - Policy need to be pushed from Extreme Policy Domain
TCL	Config		Modify AP		
TCL	Config		ModBPConfigS...		The script assists in the configuration of various switch parameters for a new edge switch. The script
TCL	Config		EXOS - Enable...		Enables Node Alias on all switch ports
TCL	Config		ModBPConfigE...		The script assists in the configuration of Various switch parameters for a new edge switch. The script
TCL	Config		Wireless - Extr...		Wireless EAC Integration 802.1X only
TCL	Config		Wireless NAC I...		Wireless NAC Integration for PoCs with PBR method of redirection
TCL	Config		Wireless NAC I...		Wireless NAC Integration for PoCs with Controller redirection method

A list of existing scripts is displayed, and you simply create a new one by clicking on the Add button. You have a choice for the type of script you want to create: simply select Python.

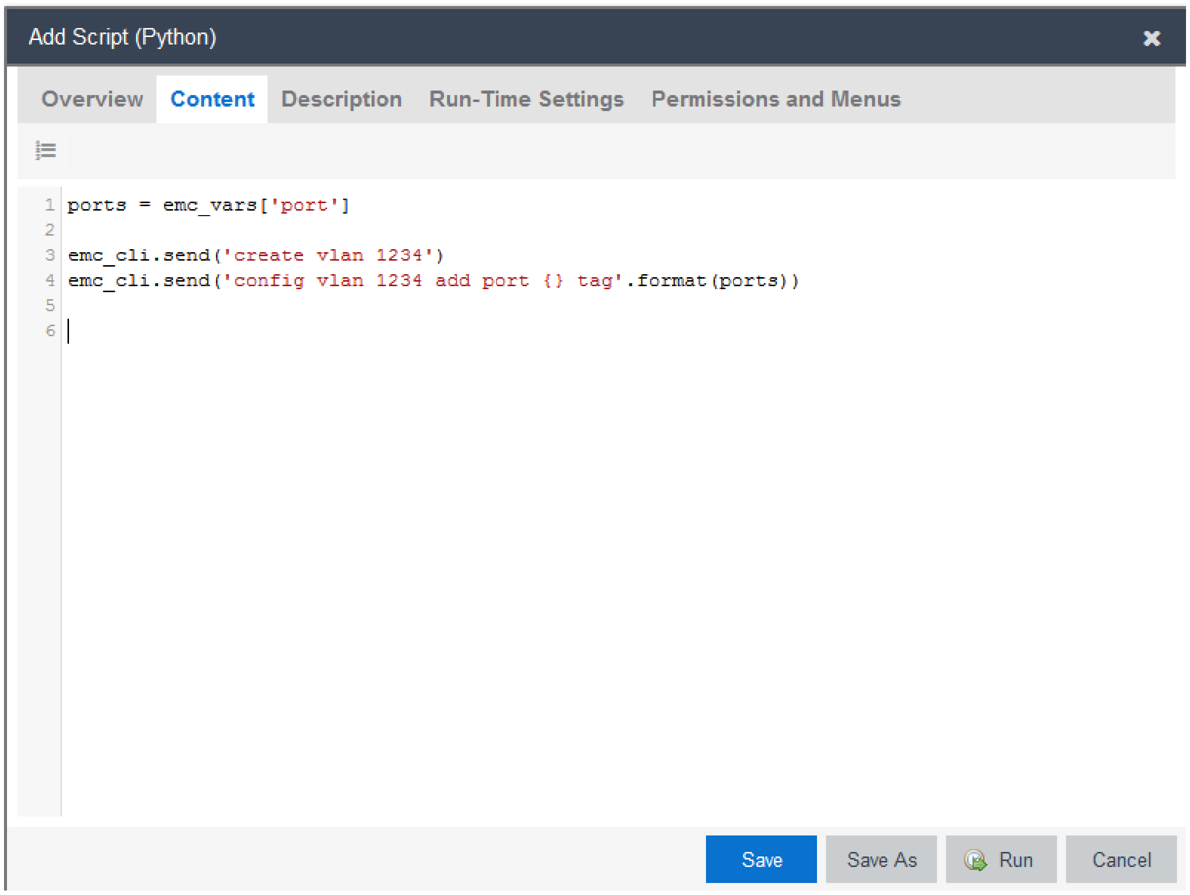


Script Type	Category	Sche...	Name	Work...	Comments
			test restconf		
			basic example		
			test		
			test vars		
Python	--		nbi-test		
TCL	Access Control		Identity Manag...		Factory script for enabling identity management configuration
TCL	Config		Wireless - Extr...		Wireless EAC Integration Guest/BYOD only - Policy need to be pushed from Extreme Policy Domain
TCL	Config		Modify AP		
TCL	Config		ModBPConfigS...		The script assists in the configuration of various switch parameters for a new edge switch. The script can also be manually
TCL	Config		EXOS - Enable...		Enables Node Alias on all switch ports
TCL	Config		ModBPConfigE...		The script assists in the configuration of Various switch parameters for a new edge switch. The script can also be manually
TCL	Config		Wireless - Extr...		Wireless EAC Integration 802.1X only
TCL	Config		Wireless NAC I...		Wireless NAC Integration for PoCs with PBR method of redirection
TCL	Config		Wireless NAC I...		Wireless NAC Integration for PoCs with Controller redirection method

After selecting Python, the editor is launched and you can start writing your script.

Note: You can write your Python script outside of the XMC embedded editor, as long as you place the resulting script into the following location:
`/usr/local/Extreme_Networks/NetSight/appdata/scripting/extensions`
This directory does not exist by default and must be created.

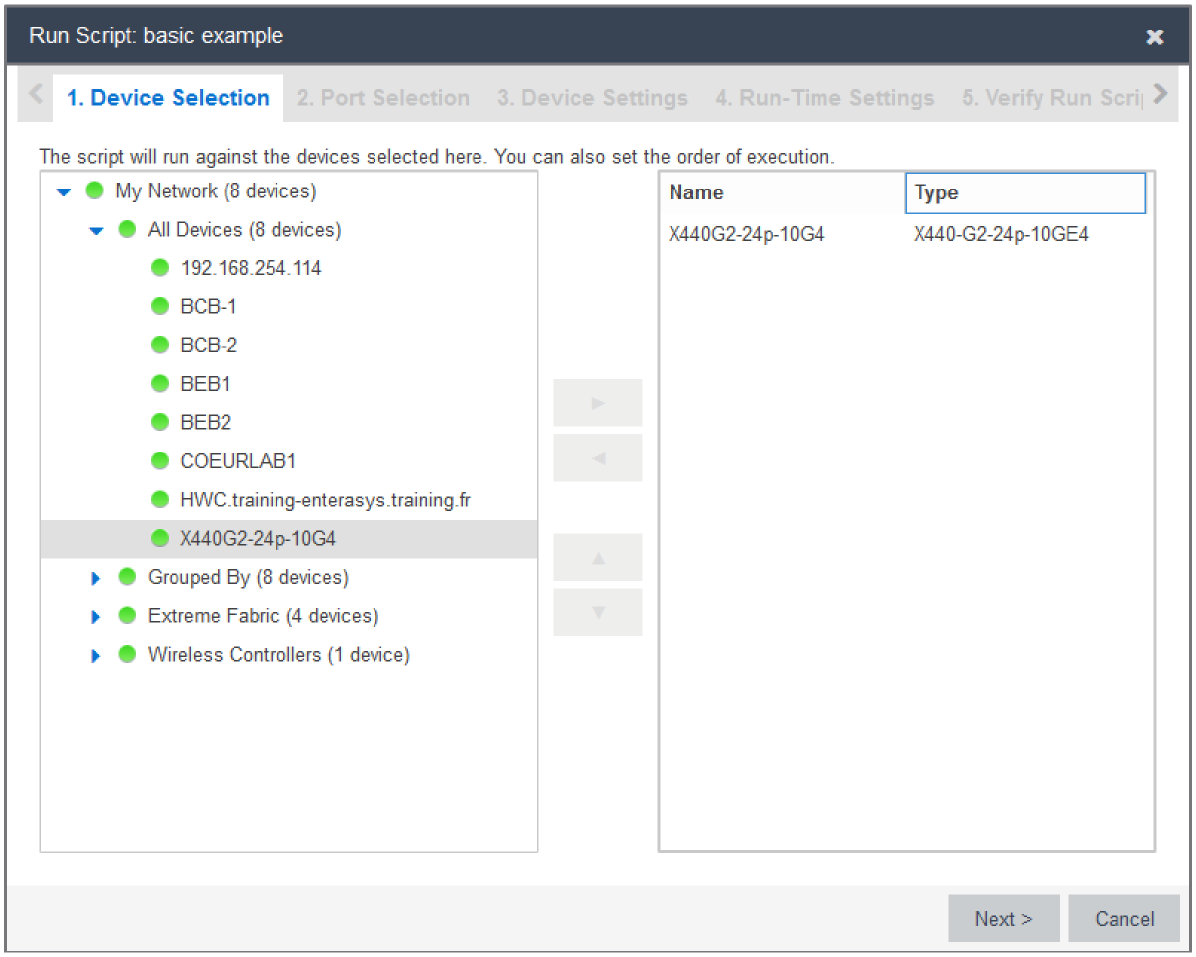
As an example, we'll create a very basic script. As you can see, we can simply write down python code if we don't need to have different functions.



```
ports = emc_vars['port']

emc_cli.send('create vlan 1234')
emc_cli.send('config vlan 1234 add port {} tag'.format(ports))

```

We need to Save it before running it. Once it is saved, clicking on the "Run" button will guide us through the different steps.

To start with, we need to select the list of devices we want to run the script against:


Run Script: basic example

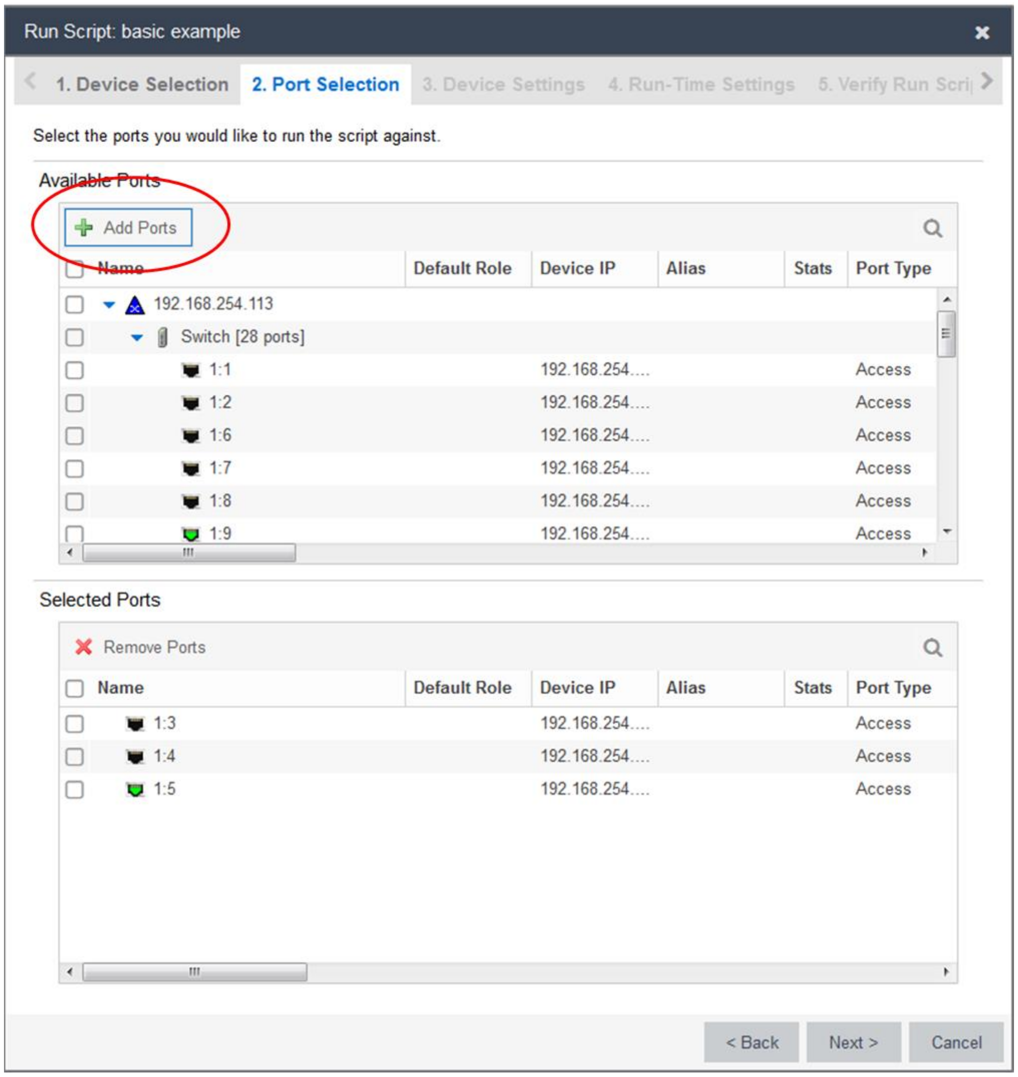
The script will run against the devices selected here. You can also set the order of execution.

- My Network (8 devices)
  - All Devices (8 devices)
    - 192.168.254.114
    - BCB-1
    - BCB-2
    - BEB1
    - BEB2
    - COEURLAB1
    - HWC.training-enterasys.training.fr
    - X440G2-24p-10G4
  - Grouped By (8 devices)
  - Extreme Fabric (4 devices)
  - Wireless Controllers (1 device)

Name	Type
X440G2-24p-10G4	X440-G2-24p-10GE4

Because we have used the `emc_vars["port"]` variable, we are asked to select a list of ports from the switch.

We need to select the ports and click the “Add Ports” button for the selection to be effective. Forgetting to click on “Add Ports” would run the script without any port selected and would raise an exception.



Select the ports you would like to run the script against.

Available Ports

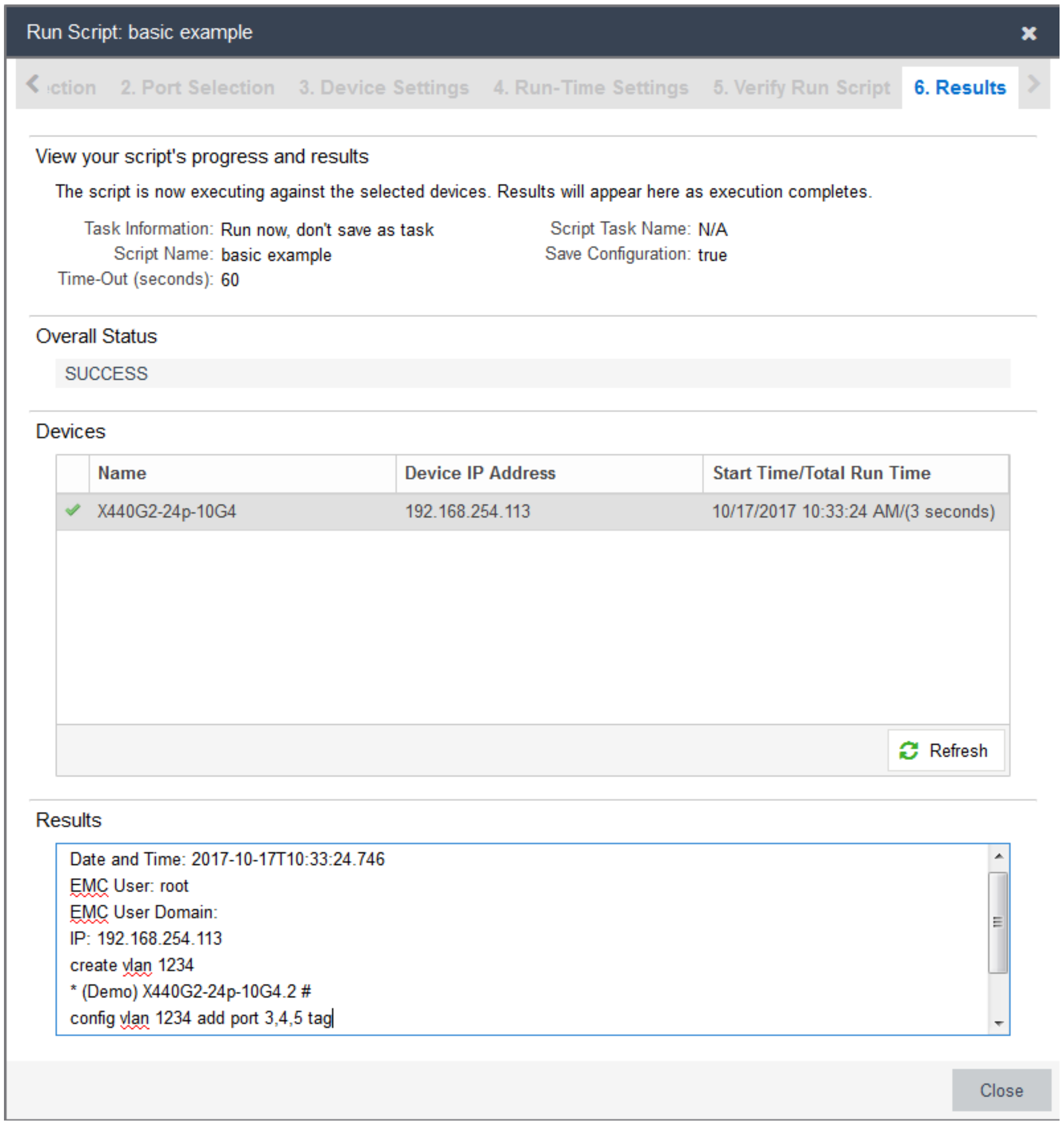
Name	Default Role	Device IP	Alias	Stats	Port Type
192.168.254.113					
Switch [28 ports]					
1:1		192.168.254....			Access
1:2		192.168.254....			Access
1:6		192.168.254....			Access
1:7		192.168.254....			Access
1:8		192.168.254....			Access
1:9		192.168.254....			Access

Selected Ports

Name	Default Role	Device IP	Alias	Stats	Port Type
1:3		192.168.254....			Access
1:4		192.168.254....			Access
1:5		192.168.254....			Access

After a few other steps to choose how and when to run the script (in our case we just run it now and don’t save it as a task), we can actually execute the script and see the result.

After completion of the script, we see some global information in the Results window. Any `print` command that we do is also displayed into that window. It’s worth to note that even without a print, the CLI output is returned in that window as well.



View your script's progress and results

The script is now executing against the selected devices. Results will appear here as execution completes.

Task Information: Run now, don't save as task
Script Name: basic example
Time-Out (seconds): 60
Script Task Name: N/A
Save Configuration: true

Overall Status

SUCCESS

Devices

	Name	Device IP Address	Start Time/Total Run Time
✓	X440G2-24p-10G4	192.168.254.113	10/17/2017 10:33:24 AM/(3 seconds)

Results

```
Date and Time: 2017-10-17T10:33:24.746
EMC User: root
EMC User Domain:
IP: 192.168.254.113
create vlan 1234
* (Demo) X440G2-24p-10G4.2 #
config vlan 1234 add port 3,4,5 tag
```

If we connect to the switch, we have confirmation of the success of the script.

```
* (Demo) X440G2-24p-10G4.1 # sh vlan
Untagged ports auto-move: On
-----------------------------------------------------------------------------------------------
Name            VID  Protocol Addr       Flags                         Proto  Ports  Virtual
                                                                              Active router
                                                                              /Total
-----------------------------------------------------------------------------------------------
Default         1    ------------------------------------------------  ANY    0 /0   VR-Default
Mgmt            4095 192.168.254.113/24  ----------------------------  ANY    1 /1   VR-Mgmt
VLAN_0042       42   10.42.0.10     /24  ----------------------------  ANY    1 /2   VR-Default
VLAN_0202       202  10.1.202.2     /24  ----------------------------  ANY    1 /1   VR-Default
VLAN_1234       1234 ------------------------------------------------  ANY    1 /3   VR-Default
VLAN_1337       1337 ------------------------------------------------  ANY    2 /3   VR-Default
-----------------------------------------------------------------------------------------------
```

4.2 Adding User-Input Variables to a Script

As of XMC 8.0.4, the metadata used with TCL are still usable “as is”, even if the syntax is more TCL-centric than really compliant with Python. But starting with XMC 8.1.2, the MetaData fields with Python scripting has evolved so that the name field and the value field can be referenced directly. This is the most appropriate way to use the MetaData starting with XMC 8.1.2.

Note: the legacy “set var name value” syntax is still supported for backward compatibility.

Any interaction with a script has to be defined in-between the MetaData tags.

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

Description can be added, but the most important part is the user-input variable definition. You need to use the specific following meta data to define a variable that the user will be prompted to set at execution time of the script.

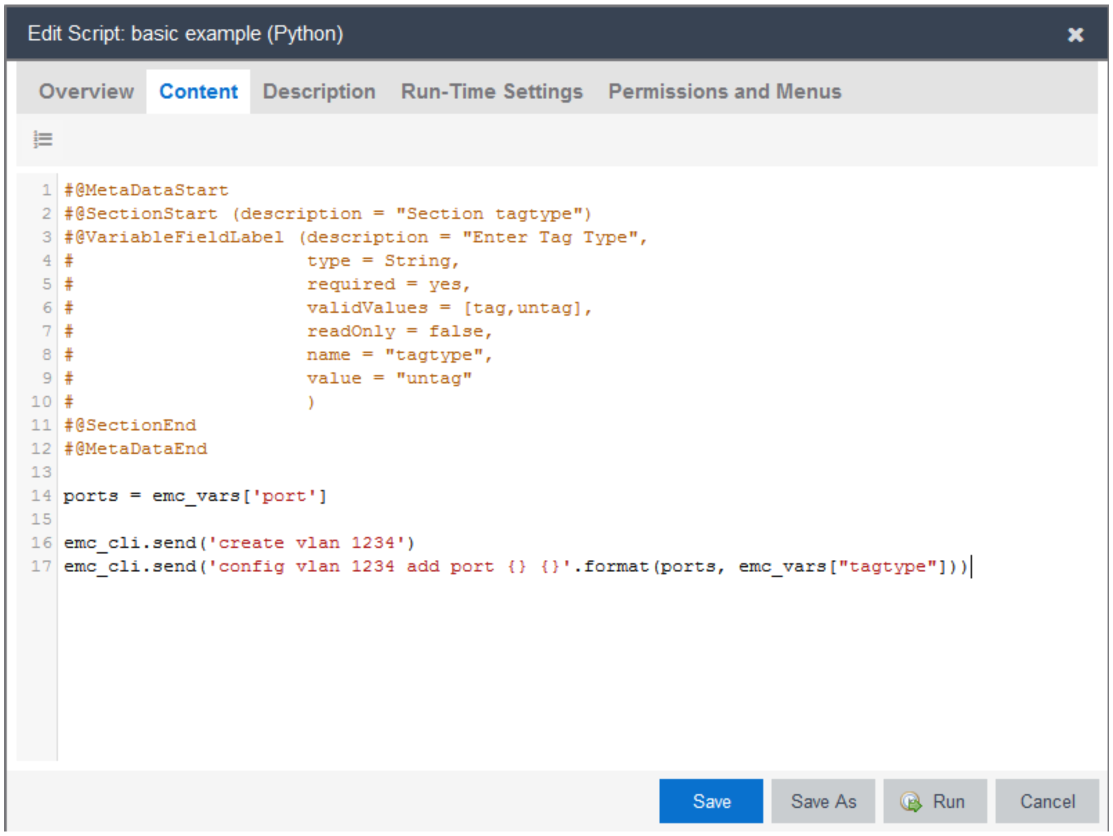
```
#@VariableFieldLabel (description = "Enter Tag Type",
#                      type = String,
#                      required = yes,
#                      validValues = [tag,untag],
#                      readOnly = no,
#                      name = "myVar",
#                      value = "42"
# 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

You can, of course, specify multiple variables if needed by repeating the above definition.

You can specify multiple values in the `VariableFieldLabel` meta data.

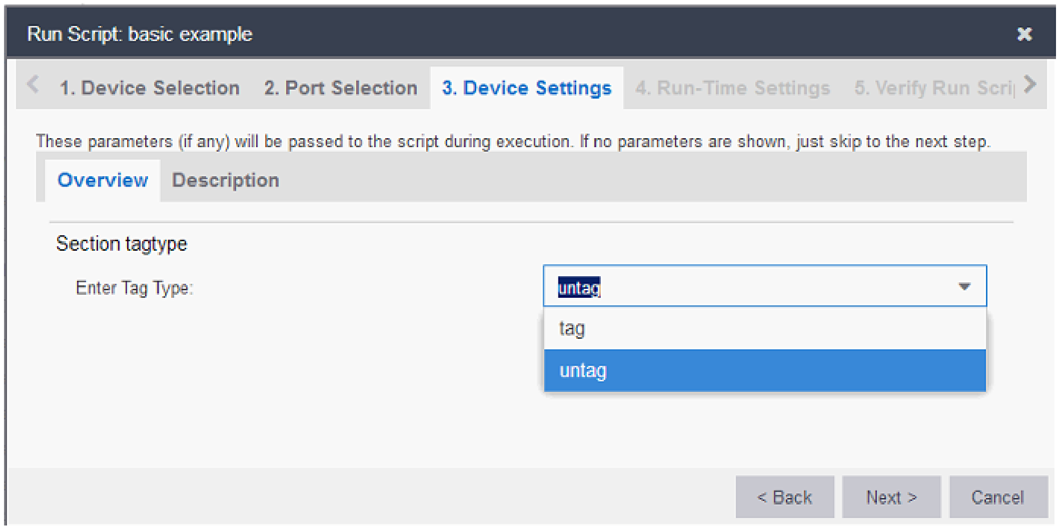
- **description**: this will be display before the value field
- **type**: what format of data is expected
- **scope**: global or device specific. Values can be device or global (default)
- **required**: yes or no
- **validValues**: a list of possible values, given inside square brackets and comma-separated
- **readOnly**: is it allowed to change the variable data? yes or no
- **name**: the name of the variable to be used in the code
- **value**: the default value of the variable, that can be overridden by the user

As of XMC 8.1.2, the type of data is string only. Below is a script example, where we ask for the user to specify if the ports are tagged or not, following our previous example.



```
#@MetaDataStart
#@SectionStart (description = "Section tagtype")
#@VariableFieldLabel (description = "Enter Tag Type",
#                     type = String,
#                     required = yes,
#                     validValues = [tag,untag],
#                     readOnly = false,
#                     name = "tagtype",
#                     value = "untag"
#                     )
#@SectionEnd
#@MetaDataEnd

ports = emc_vars['port']

emc_cli.send('create vlan 1234')
emc_cli.send('config vlan 1234 add port {} {}'.format(ports, emc_vars["tagtype"]))
```

When executing the script, we are now being prompt to fill the Tag value, with a default value already present.



These parameters (if any) will be passed to the script during execution. If no parameters are shown, just skip to the next step.

Section tagtype

Enter Tag Type: untag

As displayed in the above code, we reference our variable using the `emc_vars` object, pointing to the name of the variable: `emc_vars["tagtype"]`.

4.3 Creating a L2VSN Provisioning Script

As Fabric Provisioning for the Automated Campus solution is targeted for XMC 8.2, we can make use of the scripting capability to provide an elegant temporary integration in XMC.

4.3.1 Fabric Attach L2VSN Script

Let’s write a first script leveraging Fabric attach (FA) on both EXOS and BOSS. We’ll assume FA is already correctly configured on the FA Server.

Our script will need to connect to either EXOS or BOSS, and make sure we are running the minimum version necessary on EXOS (22.4) to use FA. Then, we configure the VLAN and the associated Service ID (I-SID).

This example tries to make a lot of verification to avoid errors, misconfigurations and other problems, but some may still happen, as it has not been tested a lot. This example is just that: an example. You can adapt it to better meet your needs.

We first declare some user-input variable for the script. We’ll need to have the VLAN ID, the Service ID (I-SID), optionally the tag configuration for optional access ports to add to that VLAN.

```
#@MetaDataStart
#@VariableFieldLabel (description = "VLAN Id <1 - 4094>",
#                      type = string,
#                      required = yes,
#                      readOnly = no,
#                      name = "vid",
#                      value = "1"
#                      )

#@VariableFieldLabel (description = "SERVICE Id <1 - 16 000 000>",
#                      type = string,
#                      required = yes,
#                      readOnly = no
#                      name = "isid",
#                      value = "1000"
#                      )

#@VariableFieldLabel (description = "802.1Q Tagging for the access (UNI) ports",
#                      type = string,
#                      required = no,
#                      validValues = [tag,untag],
#                      readOnly = no,
#                      name = "tagtype",
#                      value = "untag"
#                      )
#@MetaDataEnd
```

We’ll need to work with json data, so we are going to import that library. We also need some function to clean the various output from the CLI. We also want to create a generic function to send CLI commands.

```
import json

def getOutputOnly(inputStrings):
    try:
        version = ''.join(emc_vars["serverVersion"].split('.')[:3])
        pivotVersion = ''.join("8.1.2".split('.'))
        if int(version) == int(pivotVersion):
            lines = inputStrings.splitlines()[1:]
        else:
            lines = inputStrings.splitlines()[1:-1]
        return '\n'.join(lines)
    except:
        return None

def sendConfigCmds(cmds):
    for cmd in cmds:
        cli_results = emc_cli.send(cmd)
        if cli_results.isSuccess() is False:
            print cli_results.getError()
            return None
    return True
```

On VOSS and BOSS, a single CLI command gives us the visibility for all the VLANs configured and the corresponding I-SID if one is present. We are going to use that output to build a dictionary with both VLAN Id and I-SID as keys. This will be precious information to have to make some checks.

```
def getVidVsn(inputString):
    myList = []
    lines = inputString.splitlines()[6:-2]

    for line in lines:
        vid_dict = {}
        parts = line.split()
        if len(parts) > 1:
            vid_dict["vid"] = parts[0]
            vid_dict["isid"] = parts[-1]
        else:
            vid_dict["vid"] = parts[0]

        myList.append(vid_dict)

    return myList
```

Unfortunately, on EXOS we need two different CLI commands to have the same amount of information. We can have the list of VLANs with a NSI (I-SID in that context) configured, but it doesn’t give us all the other VLANs that may exist on the switch. However, a nice thing with EXOS is that we have access to CLI commands that output json formatted data. These commands are debug commands and not documented, but freely accessible to anyone. This is not the point of this document to explain how to find them, so we’ll just use them as is.

```
def getVlanList(reply):
    reply_json = json.loads(str(reply))
    data = reply_json.get('data')
    vlanList = []
    if data:
        for row in data:
            vlanList.append(row.get('tag'))
        return vlanList
    return None

def exosCheckNSI(vid, isid):
    cli_results = emc_cli.send('debug cfgmgr show next lldp.faMapping')
    reply = getOutputOnly(cli_results.getOutput())
    reply_json = json.loads(str(reply))
    data = reply_json.get("data")
    if data:
        for row in data:
            if isid == row.get("nsi"):
                if vid == row.get("vlanId"):
                    return 1,vid
                return 0,row.get("vlanId")
    return 2,isid
```

The first function creates a list of all the VLANs existing on the switch. We are listing the VLAN with their VID, not their names.

The second function checks if the Service ID already exists, or not, and if yes if it’s already associated to a VLAN. In that latter case, we also want to know if that VLAN is the VLAN we want to use or another one.

Now we are ready to dive into the main part of the script, where we are using all of the information to actually do something with it. The first part is basic checks to be sure we are running with correct input data and correct EXOS version, if we are on EXOS. Of course, using FA we need EXOS 22.4 as a minimum version.

```
def main():
    createVlan = True

    try:
        ports = emc_vars['port']
    except:
        ports = None

    if int(emc_vars["vid"]) > 4094 or int(emc_vars["vid"]) < 1:
        print "Error: The VLAN Id is out of range"
        return
    if int(emc_vars["isid"]) > 16000000 or int(emc_vars["isid"]) < 1:
        print "Error: The Service Id is out of range"
        return

    familyType = emc_vars['family']

    if emc_vars["isExos"] == "true":
        minExos = ''.join("22.4".split('.'))
        version = ''.join(emc_vars["deviceSoftwareVer"].split('.')[:2])
        if int(minExos) > int(version):
            print "Error: EXOS version must be 22.4 or greater"
            return
```

The last part determines if we are on EXOS or not. If this is EXOS, we need to be sure to run the correct minimum version. Everything after means we are on EXOS, and we are using our EXOS functions to do more checks.

```
        status,id = exosCheckNSI(emc_vars["vid"], emc_vars["isid"])
        if status == 1:
            createVlan = False
        elif status == 0:
            print "Error: The Service Id {} is already used for VLAN {} on device 
{}".format(emc_vars["isid"], id, emc_vars["deviceIP"])
            return
        else:
            cli_results = emc_cli.send('debug cfgmgr show next vlan.vlan')
            if cli_results.isSuccess() is False:
                print cli_results.getError()
                return
            cli_output = cli_results.getOutput()
            cli_output = getOutputOnly(cli_output)

            if cli_output:
                vlanList = getVlanList(cli_output)
                if vlanList:
                    if emc_vars["vid"] in vlanList:
                        createVlan = False
                else:
                    print "Error: No VLAN found on device {}".format(emc_vars["deviceIP"])
                    return
            else:
                print "Error: Cannot access VLAN database on device 
{}".format(emc_vars["deviceIP"])
                return
```

Once we have made all the necessary checks, we can start the configuration.

```
        if createVlan:
            cmds = ["create vlan {}".format(emc_vars["vid"])]
        else:
            cmds = []

        cmds.append("config vlan {} add nsi {}".format(emc_vars["vid"], emc_vars["isid"]))
        if ports is None:
            print "Warning: No access ports have been selected to be part of the new VLAN 
{}".format(emc_vars["vid"])
        else:
            cmds.append("config vlan {} add ports {} {}".format(emc_vars["vid"], ports, 
emc_vars["tagtype"]))

        result = sendConfigCmds(cmds)
        if result is None:
            return
```

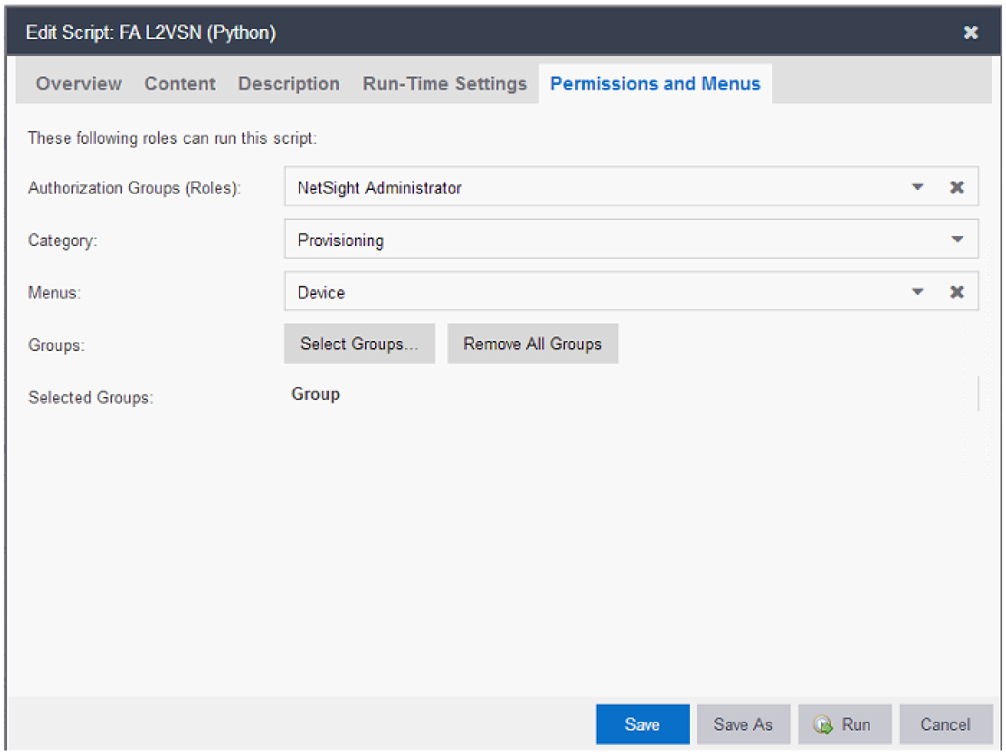
We have to do similar logic with a BOSS device.

```
    elif familyType == 'ERS Series':
        emc_cli.send("enable")
        cli_results = emc_cli.send('show vlan i-sid')
        if cli_results.isSuccess() is False:
            print cli_results.getError()
            return
        cli_output = cli_results.getOutput()
        cli_output = getOutputOnly(cli_output)

        if cli_output:
            myList = getVidVsn(cli_output)

            for row in myList:
                if row.get("vid") == emc_vars["vid"]:
                    createVlan = False
                    if row.get("isid"):
                        print "Error: The VLAN {} is already associated to the Service
{}".format(emc_vars["vid"], row.get("isid"))
                        return
                if row.get("isid"):
                    if row.get("isid") == emc_vars["isid"]:
                        print "Error: The Service Id {} is already associated to VLAN
{}".format(emc_vars["isid"], row.get("vid"))
                        return
        else:
            print "Error: Cannot access VLAN database on device {}".format(emc_vars["deviceIP"])
            return

        cmds = ["enable", "configure terminal"]
        if createVlan:
            cmds.append("vlan create {} type port".format(emc_vars["vid"]))

        if ports is None:
            print "Warning: No ports have been selected to be part of the new VLAN
{}".format(emc_vars["vid"])
        else:
            cmds.append("vlan configcontrol automatic")
            cmds.append("vlan port {} tagging {}".format(ports, emc_vars["tagtype"]+"All"))
            cmds.append("vlan members add {} {}".format(emc_vars["vid"], ports))

        cmds.append("i-sid {} vlan {}".format(emc_vars["isid"], emc_vars["vid"]))
        result = sendConfigCmds(cmds)
        if result is None:
            return

    else:
        print "You need to run this script either on an EXOS or BOSS switch"
```

And we need to not forget to call for the main() to start the Python script.

```
main()
```

We can now save our script. We can assign that script to an existing Category, such as “Provisioning”, and also add it to a Menu. We’ll choose “Device”.



Edit Script: FA L2VSN (Python)

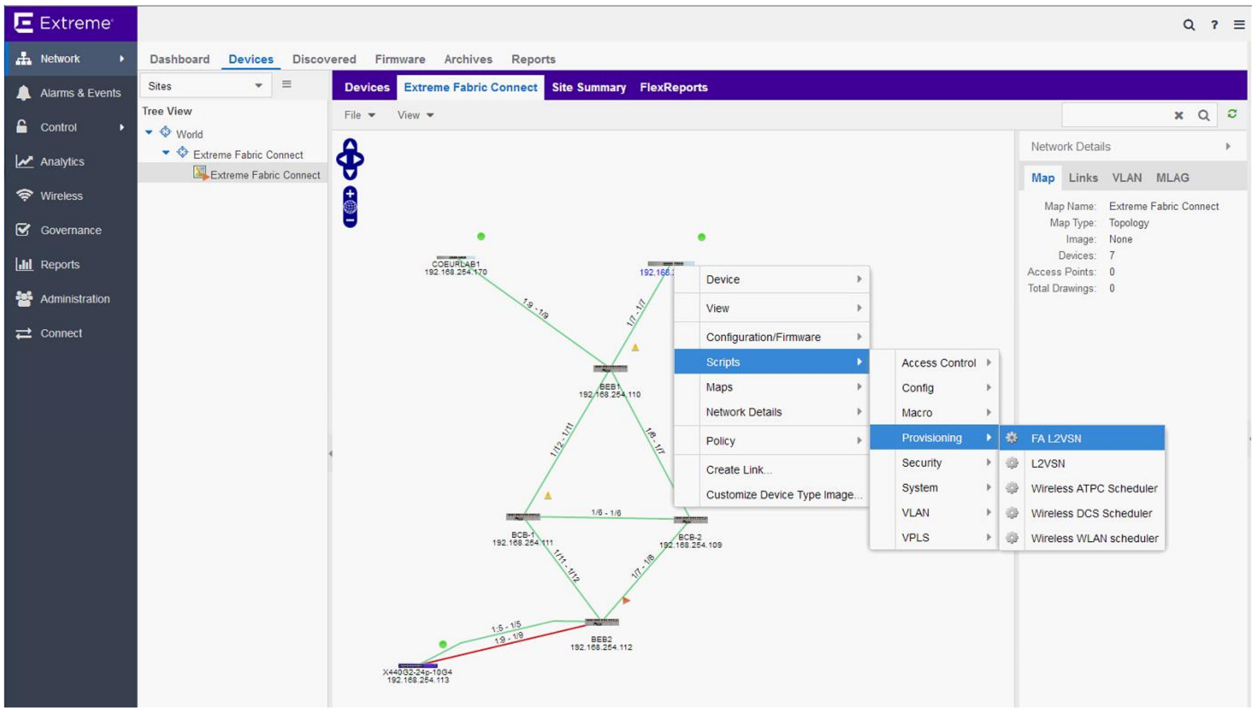
Overview | Content | Description | Run-Time Settings | **Permissions and Menus**

These following roles can run this script:

Authorization Groups (Roles):	NetSight Administrator
Category:	Provisioning
Menus:	Device
Groups:	Select Groups... / Remove All Groups
Selected Groups:	Group

Save | Save As | Run | Cancel

That way, we can launch our script from the topology view, by selecting a device on the map, and right-click on it.



4.3.2 Fabric Connect L2VSN Script

We are going to create a script to provision a L2VSN on several BEBs, asking for some user input such as VLAN Id, Service Id (I-SID) and optionally the ports to add to the VLAN. This script does not take into account a VSP Cluster.

The script will connect to a VOSS switch and configure it. Extreme Fabric Connect is assumed to be already configured and running. Once again, this script is just an example and has not been tested a lot.

```
#@MetaDataStart
#@SectionStart (description = "Service Definition")
#@VariableFieldLabel (description = "VLAN Id <1 - 4094>",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "vid",
#                     value = "1"
#                     )

#@VariableFieldLabel (description = "SERVICE Id <1 - 16 000 000>",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "isid",
#                     value = "1000"
#                     )
#@SectionEnd

#@SectionStart (description = "Single BEB Port Assignment")
#@VariableFieldLabel (description = "UNI Port",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "portlist",
#                     value = "1/1"
#                     )
#@SectionEnd

#@SectionStart (description = "Cluster BEBs MLT Assignment")
#@VariableFieldLabel (description = "MLT",
#                     type = string,
#                     required = yes,
#                     validValues = [yes,no],
#                     readOnly = no,
#                     name = "mlt",
#                     value = "no"
#                     )

#@VariableFieldLabel (description = "MLT Id <1 - 256>",
#                     type = string,
#                     required = yes,
#                     readOnly = no,
#                     name = "mltid",
#                     value = "1"
#                     )
#@SectionEnd

#@SectionStart (description = "UNI Port/MLT 802.1Q Tagging")
#@VariableFieldLabel (description = "802.1Q Tagging",
#                     type = string,
#                      required = yes,
#                      validValues = [yes,no],
#                      readOnly = no,
#                      name = "tag",
#                      value = "yes"
#                      )
#@SectionEnd
#@MetaDataEnd

def getOutputOnly(inputStrings):
    try:
        version = ''.join(emc_vars["serverVersion"].split('.')[:3])
        pivotVersion = ''.join("8.1.2".split('.'))
        if int(version) == int(pivotVersion):
            lines = inputStrings.splitlines()[1:]
        else:
            lines = inputStrings.splitlines()[1:-1]
        return '\n'.join(lines)
    except:
        return None

def getVidVsn(inputString):
    myList = []
    lines = inputString.splitlines()[6:-2]

    for line in lines:
        vid_dict = {}
        parts = line.split()
        if len(parts) > 1:
            vid_dict["vid"] = parts[0]
            vid_dict["isid"] = parts[-1]
        else:
            vid_dict["vid"] = parts[0]

        myList.append(vid_dict)

    return myList

def sendConfigCmds(cmds):
    for cmd in cmds:
        cli_results = emc_cli.send(cmd)
        if cli_results.isSuccess() is False:
            print cli_results.getError()
            return None
    return True

def CreateFAList():
    myList = []

    cli_results = emc_cli.send('show fa interface')
    if cli_results.isSuccess() is False:
        print cli_results.getError()
        return None
    cli_output = cli_results.getOutput()
    cli_output = getOutputOnly(cli_output)

    lines = cli_output.splitlines()[7:-4]

    for line in lines:
        fa_dict = {}
        parts = line.split()
        if len(parts) > 1:
            if parts[0].startswith("Port"):
                fa_dict["intf"] = ''.join( c for c in parts[0] if c not in 'Port' )
            else:
                fa_dict["intf"] = parts[0].replace("Mlt","mlt ")
            fa_dict["status"] = parts[1]
            fa_dict["auth"] = parts[4]
        myList.append(fa_dict)

    return myList

def CheckFAonPort(interface):
    fa = CreateFAList()
    if fa is None:
        return None

    for entry in fa:
        if entry["intf"] == interface:
            if entry["status"] == "enabled":
                return None
    return True

def main():
    createVlan = True

    if int(emc_vars["vid"]) > 4094 or int(emc_vars["vid"]) < 1:
        print "Error: The VLAN Id is out of range"
        return
    if int(emc_vars["isid"]) > 16000000 or int(emc_vars["isid"]) < 1:
        print "Error: The Service Id is out of range"
        return

    family = emc_vars["family"]
    if family != "VSP Series":
        print "Error: This script needs to be run on a VSP switch"
        return

    if CheckFAonPort((emc_vars["portlist"], "mlt "+emc_vars["mltid"])[emc_vars["mlt"] == "yes"])
is None:
        print "Error: Cannot create a VLAN on interface {} as Fabric Attach is configured on it
already!".format((emc_vars["portlist"], "mlt "+emc_vars["mltid"])[emc_vars["mlt"] == "yes"])
        return

    if emc_vars["mlt"] == "yes":
        ports = "MLT"
    else:
        ports = emc_vars["portlist"]

    cli_results = emc_cli.send('show vlan i-sid')
    if cli_results.isSuccess() is False:
        print cli_results.getError()
        return
    cli_output = cli_results.getOutput()
    cli_output = getOutputOnly(cli_output)

    if cli_output:
        myList = getVidVsn(cli_output)

        for row in myList:
            if row.get("vid") == emc_vars["vid"]:
                createVlan = False
                if row.get("isid"):
                    print "Error: The VLAN {} is already associated to the Service
{}".format(emc_vars["vid"], row.get("isid"))
                    return
            if row.get("isid"):
                if row.get("isid") == emc_vars["isid"]:
                    print "Error: The Service Id {} is already associated to VLAN
{}".format(emc_vars["isid"], row.get("vid"))
                    return

        cmds = ["enable", "configure terminal"]
        result = sendConfigCmds(cmds)
        if result is None:
            return

        if createVlan:
            cmds = ["vlan create {} type port-mstprstp 0".format(emc_vars["vid"])]
            if ports == "MLT":
                if emc_vars["tag"] == "yes":
                    cmds.append("mlt {} encapsulation dot1q".format(emc_vars["mltid"]))
                cmds.append("vlan mlt {} {}".format(emc_vars["vid"], emc_vars["mltid"]))
            else:
                if emc_vars["tag"] == "yes":
                    cmds.append("interface GigabitEthernet {}".format(ports))
                    cmds.append("encapsulation dot1q")
                    cmds.append("exit")
                cmds.append("vlan members add {} {} portmember".format(emc_vars["vid"], ports))

            result = sendConfigCmds(cmds)
            if result is None:
                return

        cmds = ["vlan i-sid {} {}".format(emc_vars["vid"], emc_vars["isid"])]
        result = sendConfigCmds(cmds)
        if result is None:
            return

    else:
        print "Ooops"


main()
```