

# Getting Started

In order to follow this lab please clone the DEVWKS-2601 repository from github

```
$ git clone https://github.com/RunSi/DEVWKS-2601.git  
  
$ cd DEVWKS-2601  
  
$ source ./lab_setup.sh
```

## Lab\_script

Your development environment will be setup by 'sourcing' lab\_script.sh

The script performs the following actions:-

- Sets up a local python3 virtual environment and installs all necessary dependencies.
- Brings up a CSR1000v using Vagrant
- Configures the CSR1000v for:-
  - Netconf
  - Interface configuration
  - Loopback configuration
  - Creates BGP Configuration
  - Creaets VxLAN configuration

Please allow several minutes for the script to complete

[Beginning](#) [Back](#) [Next](#)

## The Topology

Prior to initiating any tests then the network needs to be described in a Topology File.

The topology file is written in yaml and will describe attributes of your test network, such as device type, login details, login method, links between devices etc.

The GENIE Topology object is created by issuing the following commands:-

```
from genie.conf import Genie

testbed = Genie.init('path_to_name_of_yaml_file')
```

The topology file for this lab is displayed below. The topology file describes two devices *iosxe1* and *iosxe2*.

```
testbed:
  name: IOS_Testbed
  tacacs:
    username: vagrant
  passwords:
    tacacs: vagrant
    enable: vagrant

devices:
  iosxe1:
    alias: iosxe1
    type: CSR1000v
    os: iosxe
    connections:
      ssh:
        protocol: ssh
        ip: 127.0.0.1
        port: 3122
    custom:
      abstraction:
        order: [os, type]

  iosxe2:
    alias: iosxe2
    type: CSR1000v
    os: iosxe
    connections:
      ssh:
        protocol: ssh
        ip: 127.0.0.1
        port: 3222
    custom:
      abstraction:
        order: [os, type]
```

```

topology:
  iosxe1:
    interfaces:
      GigabitEthernet1:
        ipv4: 10.0.2.15/24
        link: management_link
        type: ethernet
      GigabitEthernet2:
        ipv4: 192.168.100.20/24
        link: iosxe1_to_iosxe2
        type: ethernet

  iosxe2:
    interfaces:
      GigabitEthernet1:
        ipv4: 10.0.2.15/24
        link: management_link
        type: ethernet
      GigabitEthernet2:
        ipv4: 192.168.100.21/24
        link: iosxe1_to_iosxe2
        type: ethernet


```

Some important points to note with the topology file.

- The testbed must have a name - in this case IOS\_Testbed
- The devices described - their name must correspond exactly with the hostname of the device in the testbed. e.g. iosxe1 is the hostname of the first device

The topology file for this lab can be found at:- [Topology](#)

Once the topology file has been initiated with `Genie.init('path_to_yaml_file')` a testbed object will be created.

The testbed object will have a number of attributes, objects and methods, a set of these are described in the diagram below. 

## Load the Genie Class and instantiate the testbed file

Run an iPython interactive shell:-

```
$iPython
```

Import the Genie Class from `genie.conf` and initiate the testbed file

```

from genie.conf import Genie

testbed = Genie.init('vagrant_single_ios.yaml')

```

The topology object that has been created is called `testbed`. Now look at some of the attributes of the topology object by issuing the following commands

```
testbed.devices
```

```
testbed.name
```

```
testbed.interfaces
```

The topology file has numerous attributes, objects and methods, to view these then in the iPython session type **`testbed.`** and press tab. Alternatively you can issue the following command within iPython

```
dir(testbed)
```

To explore further attributes and methods then enter the following within iPython (or tab completion)

```
dir(testbed.devices)
```

You will see that from issuing this command that `testbed.devices` has number of attributes/objects/methods, of note are *iosxe1* and *iosxe2*

Once again type within iPython (or tab completion)

```
dir(testbed.devices.iosxe1)
```

You should see are large number of attributes/objects/methods. Of particular note is the *connect* method. We shall be using the *connect* method to establishing connectivity with our testbed devices in forthcoming exercises.

[Beginning](#) [Back](#) [Next](#)

## GENIE OPS Library

The GENIE OPS Library is used to represent a device/feature's operational state/data through a Python Object. Each feature on each device is represented via a single Ops object instance, where state/status information is stored as an object attribute.

Ops objects are snapshots of a particular feature on a particular device at a specific time.

To demonstrate the power of the GENIE OPS library then please follow the sections below.

To start make sure that your Python Virtual Environment is still running from step 2 and that you are in the scripts directory. Initiate an iPython interactive session

```
$ ipython
```

Import the Topology library and the Ops Interface Model(Class) and instantiate the topology object (referred to in this example as *testbed*)

```
from pprint import pprint
from genie.conf import Genie

from genie.libs.ops.interface.iosxe.interface import Interface

testbed = Genie.init('vagrant_single_ios.yaml')
```

The commands above will:-

- Import the pprint library so as to 'pretty print' structured data to make it more easily readable
- From genie.conf import the Genie Class
- Import the Operational Model for IOSXE Interfaces
- Initiate the testbed file in order to interact with the testbed devices

Access to the devices needs to be established prior to sending any additional GENIE API calls to the device, leveraging the topology *connect* method.

First make a reference to the topology device object

As mentioned previously the device object has a method called connect. Using the connect method will establish a connection to the device using the connection method described in the topology yaml file, in this case *ssh*. You will know that a connection is successful with the output from the device being displayed in the interactive session. Once connection is made the device will be prepared for further calls on the device.

```
uut = testbed.devices.iosxe1
```

```
uut.connect()
```

---

## Learn the state of the interfaces on the device under test (iosxe1)

First an interface Ops object needs to be instantiated. The argument for instantiating the object is the device that is being tested, defined earlier as *uut*.

The *interface* object that has been instantiated has a **learn** method. The learn method will send several relevant show commands to an IOSXE device. The output of the show commands will be parsed and collated and subsequently stored as a single structured data entity(dictionary).

```
interface = Interface(device=uut)

interface.learn()
```

The data that is parsed and collated is stored as a single entry under the *info* attribute of the interface object.

To view all the returned data:-

```
pprint(interface.info)
```

From the above output you should recognise that the data is now stored as a dictionary and thus the values can be retrieved by referencing the relevant key.

For example:-

```
pprint(interface.info['nve1'])
```

And:-

```
pprint(interface.info['nve1']['phys_address'])
```

[Beginning](#) [Back](#) [Next](#)

## Genie Ops continued

---

### Compare State

In order to determine what state has changed over time we can compare 'snapshots'. Consider that each time you initiate the learn method, you are effectively taking a snapshot of current state.

The code below will demonstrate, please enter into iPython:-

```
interface_before = Interface(device=uut, attributes=['info[(.*)][bandwidth]'])
interface_before.learn()
```

---

Now use the Genie Conf class to reconfigure the device.

```
from genie.conf.base import Interface as Intf_conf
interface_cfg = Intf_conf(device=uut, name='GigabitEthernet3')
interface_cfg.bandwidth = 5000
```

Review the configuration to check it is correct

```
print(interface_cfg.build_config(apply=False))
```

Now apply configuration to the device

```
interface_cfg.build_config()
```

---

Now enter the following code:-

```
interface_after = Interface(device=uut, attributes=['info[(.*)][bandwidth]'])
interface_after.learn()
```

And finally compare the two by entering the following code:-

```
diff = interface_after.diff(interface_before)
```

```
print(diff)
```

Disconnect from the device

```
uut.disconnect()
```

---

## Conclusion

As demonstrated the Ops library is an extremely useful set of tools for retrieving state data from your devices. The preceding exercise only explored the Ops *Model* for IOSXE Interfaces. There are hundreds of further models at your disposal that support a vast range of features across IOSXE, IOSXR and NXOS. To view the available models please go to [Model Wiki](#)

## Optional Extras

### Partial retrieval of Ops data

Rather than retrieving the entire state you can choose to only save the attributes you require for the interface.

For example we only wish to retrieve the Mac Addresses of the interfaces. To achieve this

```
interface = Interface(device=uut, attributes=['info[(.*)][mac_address]'])
```

Now 'relearn' the interface object and display the output

```
interface.learn()  
pprint(interface.info)
```

Now try and find other parameters from the interface object to learn and display (for example, 'mtu', 'bandwidth') **Verify State**

A very useful feature of the Ops object is to verify the condition of a particular state.

The code below creates a function that checks the current oper\_status of GigabitEthernet3.

If the oper\_status is up, then the verification is successful it will print that Gig3 is up and return to the main body of the code.

If the oper\_status is down it will learn the interface state 3 more times with a sleep interval of 3 seconds, if after 3 attempts the interface is still down then an Exception will be raised.



Enter the code as is below to your iPython session

```
interface = Interface(device=uut)

def verify_interface_status(obj):
    if obj.info['GigabitEthernet3'].get('oper_status', None) and\
        obj.info['GigabitEthernet3']['oper_status'] == 'up':
        print('\n\nGig 3 is up')
        return
    raise Exception('Gig 3 is currently down')

interface.learn_poll(verify=verify_interface_status, sleep=3, attempt=3)
```

---

Now use the Genie Conf class to reconfigure the device.

```
interface_cfg.shutdown = True
print(interface_cfg.build_config(apply=False))
```

Review the configuration to check it is correct

```
print(interface_cfg.build_config(apply=False))
```

Now apply configuration to the device

```
interface_cfg.build_config()
```

---

Run **Verify** code again

```
interface = Interface(device=uut)

def verify_interface_status(obj):
    if obj.info['GigabitEthernet3'].get('oper_status', None) and\
        obj.info['GigabitEthernet3']['oper_status'] == 'up':
        print('\n\nGig 3 is up')
        return
    raise Exception('Gig 3 is currently down')

interface.learn_poll(verify=verify_interface_status, sleep=3, attempt=3)
```

**Full Script**

---

```
import pprint
from genie.conf import Genie

from genie.libs.ops.interface.iosxe.interface import Interface

testbed = Genie.init('vagrant_single_ios.yaml')

uut = testbed.devices.iosxe1

uut.connect()

interface = Interface(device=uut)

interface.learn()

pprint.pprint(interface.info)
pprint.pprint(interface.info['nve1'])

interface = Interface(device=uut, attributes=['info[(.*)][mac_address]'])

interface.learn

pprint.pprint(interface.info)
```

[Beginning](#) [Back](#) [Next](#)

## GENIE Parsergen

In addition to using the Ops package to retrieve and parse operational state of a device, the Genie Parsergen Class provides a one-step parsing mechanism that is capable of parsing dynamic tabular and non-tabular device outputs in a “noticeably” less lines of code compared to standard parsing mechanisms.

The Parsergen Class is particularly useful where Genie Ops does not have a model for the particular state you are looking to parse.

As an example there is currently no Genie Ops Model for NVE/VXLAN. This gap can be overcome by creating the parser that can then be leveraged by pyATS/GENIE.

The object of the remaining exercises is to \* Parse VXLAN relevant state \* Create an Ops library \*  
Run a pyATS easypy script to test condition of VXLAN state

## Tabular Parsing

The Genie Parsergen Class can deal with both Tabular and Non Tabular device output from a networking device. We shall initially explore Tabular parsing

Consider the output from the show command 'show nve vni'

Interface	VNI	Multicast-group	VNI state	Mode	BD	cfg	vrf
nve1	6001	N/A	Up	L2DP	1	CLI	N/A

As can be seen above this is a column based/tabular output. In order to parse this output we need to instruct parsergen as to the titles of the columns. Follow the commands below to parse the command 'show nve vni'

To start make sure that your Python Virtual Environment is still running from step 3 and that you are in the scripts directory. If not already running initiate an iPython interactive session

```
$ ipython
```

As in previous sections initiate the testbed topology and import the relevant libraries for this exercise

```
from pprint import pprint
from genie.conf import Genie
from genie import parsergen

testbed = Genie.init('vagrant_single_ios.yaml')
uut = testbed.devices.iosxe1
uut.connect()
```

The testbed object 'uut.device' has a method of execute. Execute will run the command on the device and return a string as the result of the command

```
output = uut.device.execute('show nve vni')
```

A list identifying the headers of the expected column output is created

```
header = ['Interface', 'VNI', 'Multicast-group', 'VNI state', 'Mode', 'BD', 'cfg',  
'vrf']
```

We will now use the parsergen oper\_fill\_tabular method to parse the string and store as structured data

```
result = parsergen.oper_fill_tabular(device_output=output, device_os='iosxe',  
header_fields=header, index=[0])
```

Now print the structured data returned

```
pprint(result.entries)
```

Determine the type of the result object entries attribute

```
type(result.entries)
```

As you will see the returned data is now structured data in the form of a dictionary

Disconnect from the device

```
uut.disconnect()
```

---

## Full Script

```
#Import Genie libraries  
from genie.conf import Genie  
from genie import parsergen  
import re  
  
from pprint import pprint
```

```
#Create Testbed Object with Genie
testbed = Genie.init('vagrant_single_ios.yaml')

#Create Device Object
uut = testbed.devices.iosxe1

#Use connect method to initiate connection to the device under test
uut.connect()

#Execute command show nve nvi on connected device
output = uut.device.execute('show nve vni')

#Create list of Header names of the table from show nve nvi - must match exactly to
that which is output on cli
header = ['Interface', 'VNI', 'Multicast-group', 'VNI state', 'Mode', 'BD', 'cfg',
'vrf']

#Use Parsergen to parse the output and create structured output (dictionary of
operational stats)
result = parsergen.oper_fill_tabular(device_output=output, device_os='iosxe',
header_fields=header, index=[0])

#Pretty Print the Dictionary
pprint(result.entries)
```

[Beginning](#) [Back](#) [Next](#)

## GENIE Non Tabular Parsing

Not all output from the device will be in tabular form. Parsergen can deal with non tabular returned data.

Parsergen tries to match a given set of data using regular expressions that describe the values found in the show command output.

Consider the following output from the *show nve interface nve 1* .

We shall parse the data to retrieve Source\_Interface and Primary address based upon an encapsulation of Vxlan

```
Interface: nve1, State: Admin Up, Oper Up, Encapsulation: Vxlan,  
BGP host reachability: Disable, VxLAN dport: 4789  
VNI number: L3CP 0 L2CP 0 L2DP 1  
source-interface: Loopback10 (primary:172.16.10.1 vrf:0)
```

There are two methods by which we can retrieve this data - Manual regular expressions and Markup

### Using Regular Expressions manually

To start make sure that your Python Virtual Environment is still running from step 4 and that you are in the scripts directory. Initiate an iPython interactive session and initialise the testbed

```
$ ipython
```

```
from pprint import pprint  
from genie.conf import Genie  
from genie import parsergen  
  
testbed = Genie.init('vagrant_single_ios.yaml')  
uut = testbed.devices.iosxe1  
uut.connect()
```

Create a dictionary of show commands. Only one show command for IOSXE in this instance

```
show_cmds = {  
    'iosxe': {  
        'show_int' : "show nve interface {}",  
    }  
}
```

Create a dictionary of regular expressions to capture the elements required in the output. The example has regular expressions that will capture the encapsulation type, the source interface and the primary address.

As useful tool for creating and validating python *re* based regular expressions can be found here: [Pythex](#)

```
regex = {
    'iosxe': {
        'nve.intf.if_encap': r'[a-zA-Z0-9\:\,\s]+Encapsulation:\s+(\w+)',
        'nve.intf.ifname': r'^Interface:\s(\w+)',
        'nve.intf.vxdport': r'[a-zA-Z0-9\:\,\s]+\sVxLAN\sdpport:\s(\w+)',
        'nve.intf.source_intf': r'^source-interface:\s+(\w+)',
        'nve.intf.primary': r'[a-zA-Z0-9\:\-\s]+Loopback[a-zA-Z0-9\s\(\)+\:(\d+\.\d+\.\d+\.\d+)',
        'nve.intf.vrf': r'[a-zA-Z0-9\:\-\s]+Loopback[a-zA-Z0-9\s\(\)+\:(\d+\.\d+\.\d+\.\d+)\s+vrf:(\w+)\)',
    }
}

regex_tags = {
    'iosxe': ['nve.intf.if_encap', 'nve.intf.ifname', 'nve.intf.vxdport',
        'nve.intf.source_intf',
        'nve.intf.primary', 'nve.intf.vrf']
}
```

'Extend' the Parsergen Class to include the show commands and the regular expressions

```
parsergen.extend(show_cmds=show_cmds, regex_ext=regex, regex_tags=regex_tags)
```

Now determine the parameters you wish to start the regex search on. The first item in the tuple is the key name of the regex value, the second item is the value being searched in this case all interfaces with Vxlan encapsulation

```
attrValPairsToParse = [('nve.intf.if_encap', 'Vxlan')]
```

Finally we create the object pgfill by calling the *parsergen.oper\_fill* method is called. The arguments in this method will \* determine the device to be called (uut) \* determine which show command to call from the key show\_int and use nve1 as the interface name for the show command \* Provide the attribute value pairs to search on \* And use the defined regular expressions that begin with *nve.intf*

```
pgfill = parsergen.oper_fill (
    uut,
    ('show_int', ['nve1']),
    attrValPairsToParse,
    refresh_cache=True,
    regex_tag_fill_pattern='nve\ intf')
```

Now enter the parse method for pgfill to populate parsergen ext\_dictio attribute with the parsed

items

```
pgfill.parse()

pprint(parsergen.ext_dictio)
```

Disconnect from the device

```
uut.disconnect()
```

---

## Full Script

```
from genie.conf import Genie
from genie import parsergen

from pprint import pprint

testbed = Genie.init('vagrant_single_ios.yaml')
uut = testbed.devices.iosxel
uut.connect()

show_cmds = {
    'iosxe': {
        'show_int' : "show nve interface {}",
    }
}

regex = {
    'iosxe': {
        'nve.intf.if_encap': r'[a-zA-Z0-9\\:\\,\\s]+Encapsulation:\\s+(\\w+)',
        'nve.intf.ifname': r'^Interface:\\s(\\w+)',
        'nve.intf.vxdport': r'[a-zA-Z0-9\\:\\,\\s]+\\sVxLAN\\sdport:\\s(\\w+)',
        'nve.intf.source_intf': r'^source-interface:\\s+(\\w+)',
        'nve.intf.primary': r'[a-zA-Z0-9\\:\\-\\s]+Loopback[a-zA-Z0-9\\s\\(\\)+\\:\\(\\d+\\.\\d+\\.\\d+\\.\\d+)',
        'nve.intf.vrf': r'[a-zA-Z0-9\\:\\-\\s]+Loopback[a-zA-Z0-9\\s\\(\\)+\\:\\d+\\.\\d+\\.\\d+\\.\\d+\\s+vrf:(\\w+)\\)',
    }
}

regex_tags = {
    'iosxe': ['nve.intf.if_encap', 'nve.intf.ifname', 'nve.intf.vxdport',
        'nve.intf.source_intf',
        'nve.intf.primary', 'nve.intf.vrf']
}
```



```
parsergen.extend(show_cmds=show_cmds, regex_ext=regex, regex_tags=regex_tags)

attrValPairsToParse = [('nve.intf.if_encap', 'Vxlan')]
pgfill = parsergen.oper_fill (
    uut,
    ('show_int', ['nve1']),
    attrValPairsToParse,
    refresh_cache=True,
    regex_tag_fill_pattern='nve\ intf')
pgfill.parse()
pprint(parsergen.ext_dictio)
```

[Beginning](#) [Back](#) [Next](#)

## Using Markup Text to parse Non Tabular Output

Rather than explicitly defining regular expressions for each item to retrieve, as an alternative we can use a special CLI command markup format that will automatically generate the regular expressions.

If you have an iPython session running. Close and restart iPython

Initiate an iPython interactive session and initialise the testbed

```
$ ipython

from pprint import pprint
from genie.conf import Genie
from genie import parsergen

testbed = Genie.init('vagrant_single_ios.yaml')
uut = testbed.devices.iosxe1
uut.connect()
```

Enter the following to assign the *marked up* string to the variable markedupIOSX

```
markedupIOSX = '''
OS: iosxe
CMD: show_nve_interface
SHOWCMD: show nve interface {ifname}
PREFIX: nve.intf
ACTUAL:

Interface: nve1, State: Admin Up, Oper Up, Encapsulation: Vxlan,
BGP host reachability: Disable, VxLAN dport: 10000
VNI number: L3CP 0 L2CP 0 L2DP 1
source-interface: Loopback10 (primary:1.1.1.1 vrf:22)

MARKUP:
Interface: XW<ifname>Xnve1, State: Admin XW<state>XUp, Oper Up, Encapsulation:
XW<encap>XVxlan,
BGP host reachability: Disable, VxLAN dport: XN<udp_port>X1000
VNI number: L3CP 0 L2CP 0 L2DP 1
source-interface: XW<source_interface>XLoopback0
(primary:XA<primary_address>X1.1.1.1 vrf:XN<VRF>X22)'''
```

You will notice in the string that there are some key components

**OS:** Define the operating system being used

**CMD:** Used by parsergen as the dict key for the *SHOWCMD*

**SHOWCMD:** The actual show command to be issued

**PREFIX** Will be used to prefix the keys for each item parsed

**ACTUAL** Output expected from the device (optional)

**MARKUP** The Output with markup added. Will be used to identify items to parse

The Markup itself begins and ends with **X** with the key name inbetween. For example **XW\X** will assign a value to the key `nve.intf.ifname`

Full list of Markup tags are included at the bottom of this file.

The remaining commands are similar to those used for parsing with regular expressions

'Extend' the Parsergen Class to include the show commands and the regular expressions

```
parsergen.extend_markup(markedupIOSX)
```

Now determine the parameters you wish to start the regex search on. The first item in the tuple is the key name of the regex value, the second item is the value being searched. In this instance only nve interfaces that have a Vxlan encapsulation are being considered

```
attrValPairsToCheck = [('nve.intf.encap', 'Vxlan'),]
```

Create an object called pgfill from the parsergen.oper\_fill method in order to create a dictionary of the parsed output.

```
pgfill = parsergen.oper_fill(device=uut,
                             show_command=('show_nve_interface', []),
                             {'ifname': 'nve1'}),
                             attrvalpairs=attrValPairsToCheck,
                             refresh_cache=True,
                             regex_tag_fill_pattern='nve\.intf')
```

Now call the parse method for the object pgfill and print

```
pgfill.parse()

pprint(parsergen.ext_dictio)
```

Disconnect from the device

```
uut.disconnect()
```

---

## Mark Up Reference

The following are the available values for x in the XxX notation:

- A - IPv4 or IPv6 address.
  - B - Value terminated with a close brace, bracket, or parenthesis.
  - C - Value terminated with a comma.
  - F - Floating point number.
  - H - Hexidecimal number.
  - I - Interface name.
  - M - Mac address.
  - N - Decimal number.
  - R - everything else to the newline.
  - P - IPv4 or IPv6 prefix.
  - Q - Value terminated by a double quote.
  - S - Non-space value.
  - T - Time (00:00:00)
  - W - A word.
- 

## Full Script

```
from genie.conf import Genie
from genie import parsergen
from pprint import pprint

#open marked up text file and read into variable
f = open('markup.txt', 'r')
markedupIOSX = f.read()

#initiate testbed
testbed = Genie.init('vagrant_single_ios.yaml')
uut = testbed.devices.iosxel
uut.connect()

#extend parsergen markup with text file indicating elements to parse
parsergen.extend_markup(markedupIOSX)

#identify which parsed element to check as true
attrValPairsToCheck = [
    ('nve.intf.encap', 'Vxlan'),]

#parsergen will connect to device and issue show nve interface nve1
#Will check for elements to be parsed and create a dictionary
pgfill = parsergen.oper_fill(device=uut,
    show_command=\
('show_nve_interface', [], {'ifname':'nve1'}),
    attrvalpairs=attrValPairsToCheck,
    refresh_cache=True, regex_tag_fill_pattern='nve\.intf')

pgfill.parse()
pprint(parsergen.ext_dictio)
```

[Beginning](#) [Back](#) [Next](#)

## GENIE Creating an OPS object

We are now going to create a VxLAN OPS object that will collate the output of the two parsers we created earlier.

For the sake of brevity these two parsers have been defined within Classes in the file [iosxevxlan.py](#). The parsers are also inheriting from Genie Metaparser. The configuration of Metaparser is outside the scope of this workshop but further details can be found at - [Metaparser](#)

---

If you have an iPython session running. Close and restart iPython

Initiate an iPython interactive session and initialise the testbed

```
$ ipython

from genie.conf import Genie
testbed = Genie.init('vagrant_single_ios.yaml')
ut = testbed.devices.iosxel
ut.connect()
```

First we shall import from Genie ops the Base class. We will create a class that will inherit from 'Base' to leverage the 'Maker' functionality.

'Maker' simplifies the process of mapping parsers output to the ops object attributes.

Further information on the Maker class can be found at [Maker](#)

In addition we will import the parsers that were created earlier.

Enter the code below into your ipython session

```
from genie.ops.base import Base
from iosxevxlan import ShowNveVni, ShowNvePeers, ShowNveIntf
```

We now create a class that will be our Ops object, named Vxlan. This class inherits from the Base class of Genie Ops.

A method which referred to as *learn* is created. The remaining code performs the following functions

- Runs a for loop issuing the commands for the parsers and then adds data (add\_leaf) to the new Ops object structure.
- src is the dictionary item from the parsed output. For example '['(?P.\*)][VNI]' will equate to the value of VNI (6001)
- dest is where the data will be placed in the new object structure referenced as *info*. In this

case the src and dest keys are the same but this does not have to be the case

- Finally the make() is invoked to finalise the new object structure.

```
#Create the Vxlan Ops Class
class Vxlan(Base):

    def learn(self, custom=None):

        # Capture output from ShowNveVni parser
        src = '[(?P<interf>.*)]'
        dest = 'info[(?P<interf>.*)]'
        req_keys = ['[VNI]', '[Multicast-group]', '[VNIstate]', '[Mode]']
        for key in req_keys:
            self.add_leaf(cmd=ShowNveVni,
                          src=src + '[{}]' .format(key),
                          dest=dest + '[{}]' .format(key))

        # Capture output from ShowNvePeers parser
        src = '[(?P<nvename>.*)]'
        dest = 'info[(?P<nvename>.*)]'
        req_keys = ['[Peer-IP]', '[Router-RMAC]', '[Type]', '[state]']
        for key in req_keys:
            self.add_leaf(cmd=ShowNvePeers,
                          src=src + '[{}]' .format(key),
                          dest=dest + '[{}]' .format(key))

        # Capture output from ShowNveIntf parser
        src = '[(?P<nveint>.*)]'
        dest = 'info[(?P<nveint>.*)]'
        req_keys =
        ['[nve.intf.if_encap]', '[nve.intf.primary]', '[nve.intf.source_intf]']
        for key in req_keys:
            key_strip = key[10:]
            self.add_leaf(cmd=ShowNveIntf,
                          src=src + '[{}]' .format(key),
                          dest=dest + '[{}]' .format(key_strip))

        #Add ops data to the Vxlan object
        self.make()
```

Finally create a new ops object called myvxlan and learn from the device

```
myvxlan = Vxlan(device=uut)
```

```
myvxlan.learn()
```

```
myvxlan.info
```

```
Disconnect from the device
```python
ut.disconnect()
```

**You have successfully created a VxLAN Ops Model.**

[Beginning](#) [Back](#) [Next](#)



## Using EasyPy to run a Test with the new VxLAN Genie Ops model

Now that we have a newly created VxLAN Genie Ops Model we can leverage it within the pyATS framework.

The functionality of EasyPy and pyATS AETest is beyond the scope of this workshop, however running the commands below will demonstrate the power of pyATS for automating and reporting on your testing protocols.

---

Open a new terminal session and go to the scripts directory

```
cd ~/DEVWKS-2601/scripts
```

Now run the following command in your terminal session

```
easypy vxlancheckjob.py -html_logs . -no_archive -testbed_file  
vagrant_single_ios.yaml
```

Open up the Task.html file in the runinfo directory

```
open TaskLog.html
```

---

From your terminal ssh into your device

If the lab you are using is the Sandbox - ssh cisco@10.10.20.48 password cisco\_1234!

If the lab you are using is a local vagrant machine - ssh -p 3122 vagrant@127.0.0.1 vagrant

```
conf t  
interface nve 1  
shut
```

Exit the session from your device and rerun EasyPy

```
easypy vxlancheckjob.py -html_logs -no_archive -testbed_file  
vagrant_single_ios.yaml
```

And finally review the report and open the Task.html file

```
less runinfo/vxlancheckjob.xxxxxxxxxxxxxx/vxlancheckjob.report  
open runinfo/vxlancheckjob.xxxxxxxxxxxxxx/TaskLog.html
```

## **Congratulations you have now finished the DEVWKS-2601 Workshop**

---

*Restore the laptop to it's initial state*

```
$ ./lab_cleanup.sh
```

**Be sure to attend DEVWKS-2595 Stateful Network Validation using pyATS + GENIE**

[Beginning](#) [Back](#) [Next](#)