



Cisco live!

**Cisco live!**

June 10-14, 2018 • Orlando, FL

IMAGINE  
INTUITIVE

## Programming for Network Engineers

### LABRST-2678



Powered by Cisco dCloud!  
<https://dcloud.cisco.com>

#### Speakers:

Ambili Sasidharan – AS Consulting Engineer  
Sivarajan Thiruvadi – AS Consulting Engineer

## Table of Contents

PREREQUISITES .....	3
DISCLAIMER .....	3
PRE-CONFIGURATION.....	3
LAB COMPLETION.....	3
INTRODUCTION .....	4
LAB TOPOLOGY .....	7
LAB 0: CONNECT TO DCLOUD LAB NETWORK.....	8
LAB 1: EXPLORE THE BASH-SHELL FEATURES ON THE CISCO NEXUS 9000 SERIES SWITCHES .....	11
LAB 2: MANAGE CISCO NEXUS 9000 SWITCH USING THE CISCO NX-API.....	18
LAB:3 CONFIGURE NX-OS VIA PYTHON PROGRAMMING.....	23

## Prerequisites

- Basic Python programming, Linux and REST API knowledge
- It is strongly recommended to go through the lab guide. This will give you a good foundation of interacting with NX-OS through various methods other than CLI like Bash, NX-API and Python scripts.

## Disclaimer

This training document is to familiarize with Cisco NX-OS programming capabilities. Although the lab design and configuration and python script examples could be used as a reference, it's not a real design, thus not all recommended features are used, or enabled optimally to use in production networks. For the design related questions please contact your representative at Cisco, or a Cisco partner. Else feel free to reach out to us. We will be glad to answer.

## Pre-Configuration

Due to limited duration of WISP lab, we have some pre-configuration, already configured to save your time.

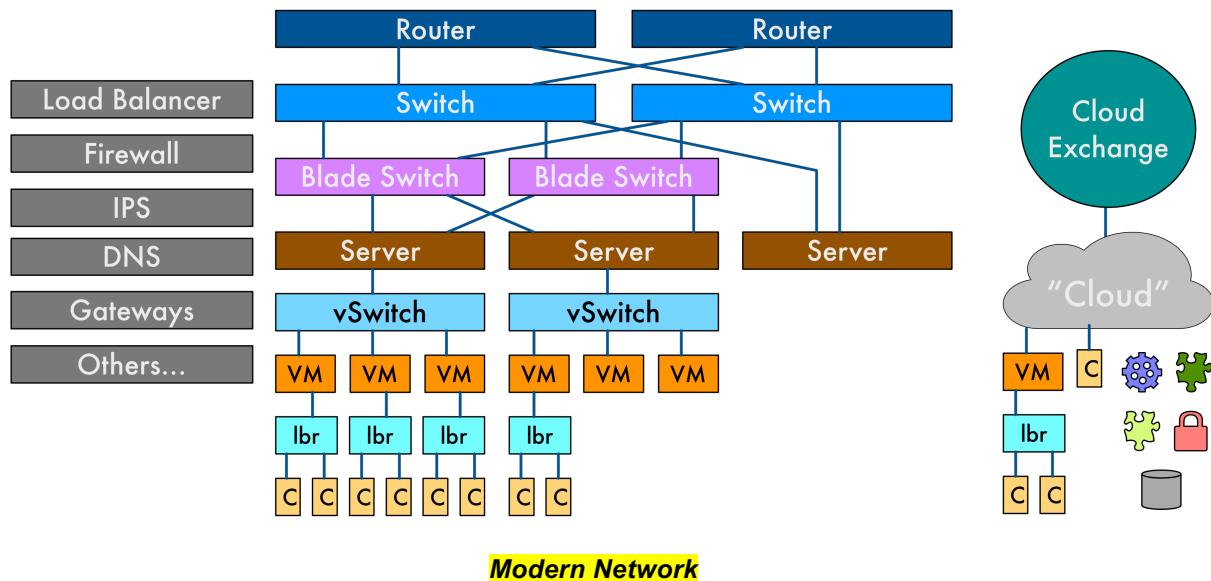
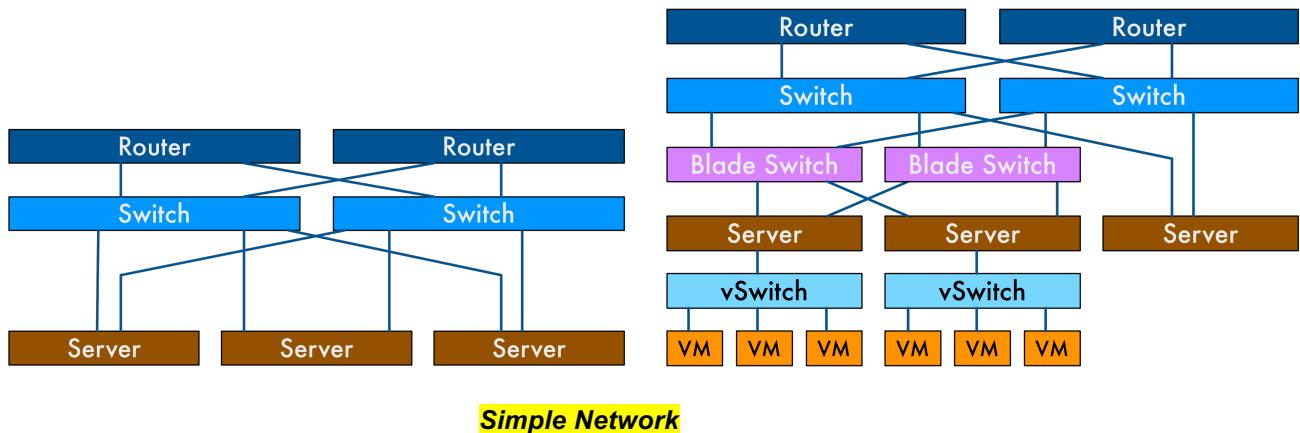
## Lab completion

Upon completion of this lab, you will be able to:

- Describe various use cases and examples of NX-OS network programmability.
- Interact with Cisco NX-OS using various programmable interfaces.
- Use NX-API to communicate with Nexus devices.
- Leverage python scripts to manage network devices.

## Introduction

In the SDN Era, the roles and responsibilities of the Network Engineers are drastically changing. Reducing Opex and Faster Time to market are the key business drivers for most of the IT organizations today. As the networks grow in size, it has increased the management and operations complexity to a very high level. State of the Network has changed a lot over the recent years from a simple three tier architecture to multi-level modern network architecture.



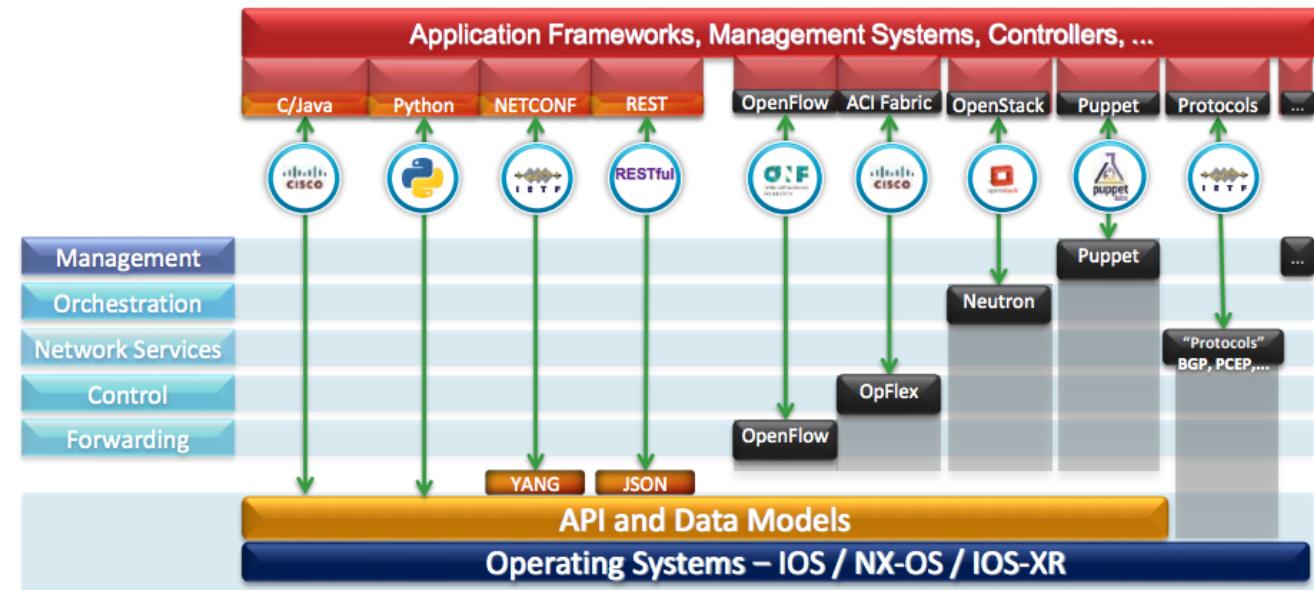
Network Programmability is the new way to of communicating with the network devices via API's. In today's world network administrators want programming capabilities in their networking devices for several reasons. The main reason is to automate provisioning, push configuration changes and troubleshoot complex network issues. Yet business and application teams demands the network devices to be provisioned faster, with less resources and no human-error.

CLI is still a critical tool in the network engineers tool kit, it is readily available, easy to use and grants valuable insights about the state of network. Unfortunately, CLI is not without its flaws, especially when it comes to network programmability and automation. CLI is inherently not scalable and it is designed to have 1:1 human interaction which makes it slow. Another issue is that the CLI outputs or data passed over CLI are unstructured raw data which are traditionally designed in a human readable fashion.

The main goal of this lab session is to educate Network engineers about the different access methods available to interact with the Networking devices other than CLI. This lab is designed for Network Engineers who are getting started with programming. The lab demonstrates Cisco NX-OS network programming capabilities by using various access methods like Bash, NX-API and Python. You will learn how to interact with the network devices using SSH and API by leveraging Python libraries and NX-API Sandbox. You will also learn how easy it is to get started with programming and automate basic day to day networking tasks.

So the general rule here is “Use Programmability and Automation whenever possible and fallback to using CLI for the tasks that cannot be automated.

## Device Programmability Options – No Single Answer!



Network Programmability in Cisco can be implemented with these industry standards and free protocols:

- **RESTful Interface:** In a simple sense, Representational State Transfer (REST) is the standard for common web browsers to interact with a website. Cisco has opened up RESTful interfaces on many routers and switches, and also in other SDN controller and orchestration platforms for Network Programmability.
- **Python:** This free programming language has grown considerably in popularity in the Linux community for years. Cisco has now integrated the interpreted scripting language of python into the NX-OS operating system, ACI and other Cisco platforms.
- **XML and JSON:** Extensible Markup Language (XML) and JavaScript Object Notation (JSON) are standard data encoding formats that are both human-readable and machine-readable. For instance, an entire router or switch configuration can be displayed in XML or JSON, edited with a text editor, then used

with Python or REST to automate changes to the network.

- **Data Models:** A data model is a standard way to define how data relates to other data and how all data is processed and stored. All of the configurations of most networking devices can be represented in a large data model. Data models can then be automated with Network Programmability. Another emerging data model that can be applied to Cisco routers and switches is the YANG model. YANG is a data model from the expression “Yet Another Next Generation,” which is an open standard designed to overcome the weaknesses of SNMP.

The days of having to make changes to network devices one at a time are coming to an end. In this lab we will explore the Network Programmability options available on NX-OS. Other Cisco Operating Systems and products also supports many more programmability options, please check the Devnet zone, breakout sessions and reference section to learn more on this topic.

## Lab Outline:

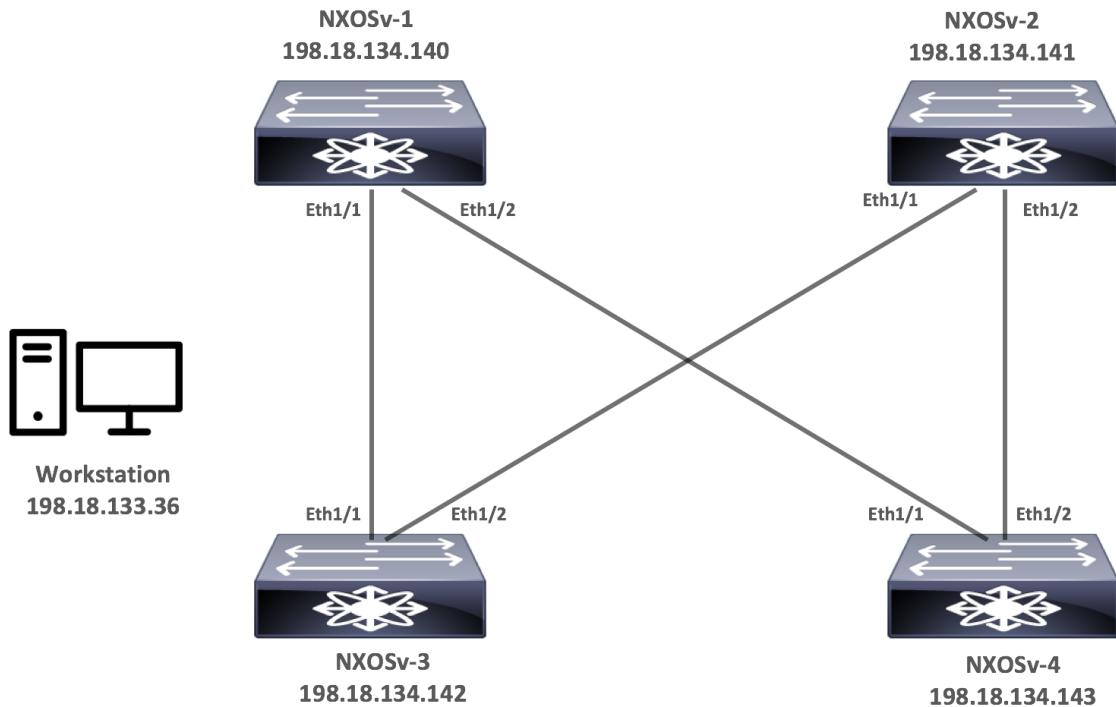
Lab 0: Connect to dcloud lab network

Lab 1: Explore the bash-shell features on the Cisco Nexus 9000 Switch

Lab 2: Manage Cisco Nexus 9000 Switch using the Cisco NX-API

Lab 3: Build and Execute Python Scripts to automate device administration

## Lab Topology



Note: The workstation (wkst1) provides all the required access to the nxos devices

## Lab 0: Connect to dCloud lab network

This lab leverages dCloud, which uses virtual servers and switches that runs on a simulator. These devices have most of the functions of actual hardware based devices, but there is no data plane traffic.

You need to VPN into the dCloud environment in order to access devices to complete this lab.

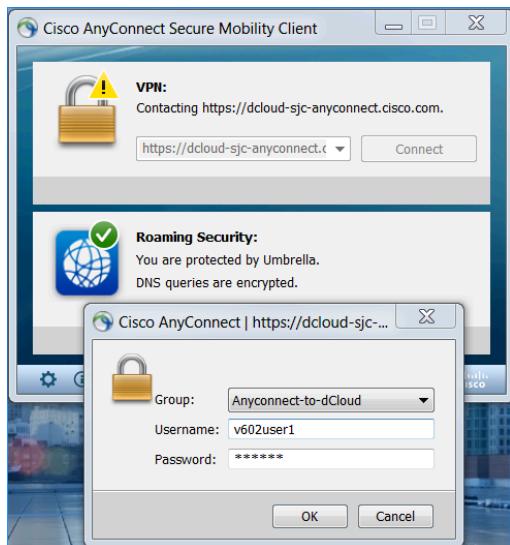
Click on the Cisco AnyConnect client and check to see if the client is connected to another VPN session. If yes, go ahead and disconnect.

To reach dCloud, enter the following server url to connect:

<https://dcloud-sjc-anyconnect.cisco.com>

For the vpn credentials, input the username and password that is provided by the lab instructor.

Input the username and password. Click OK.



Once VPN connection is successful, you should be able to access the lab environment.

Now open a Remote Desktop Connection (RDP) to the lab workstation.

### Workstation (Windows 7)

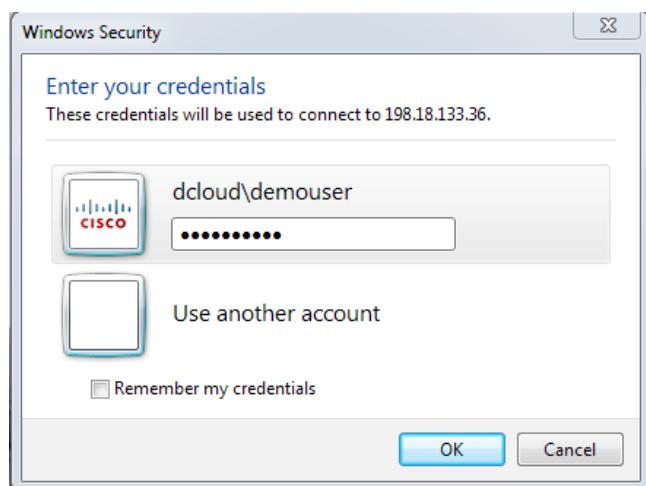
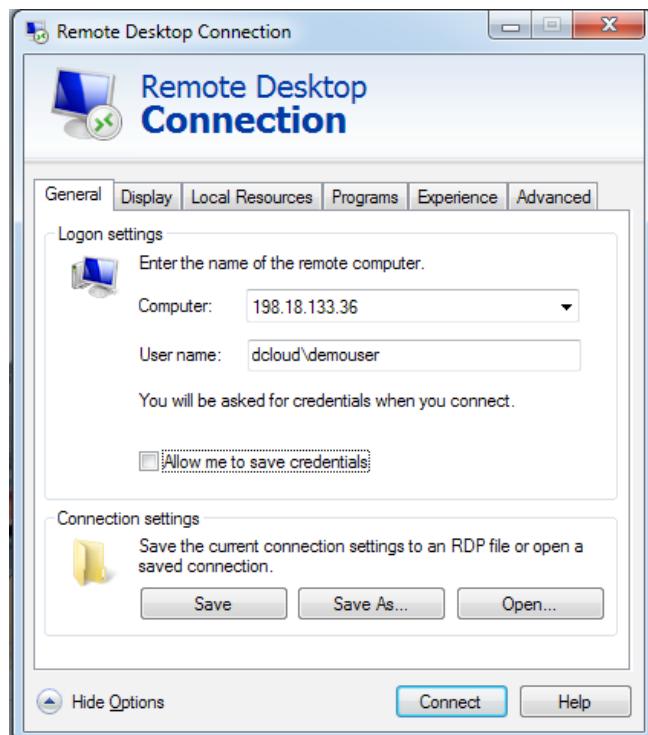
#### IP Address:

198.18.133.36

#### Credentials:

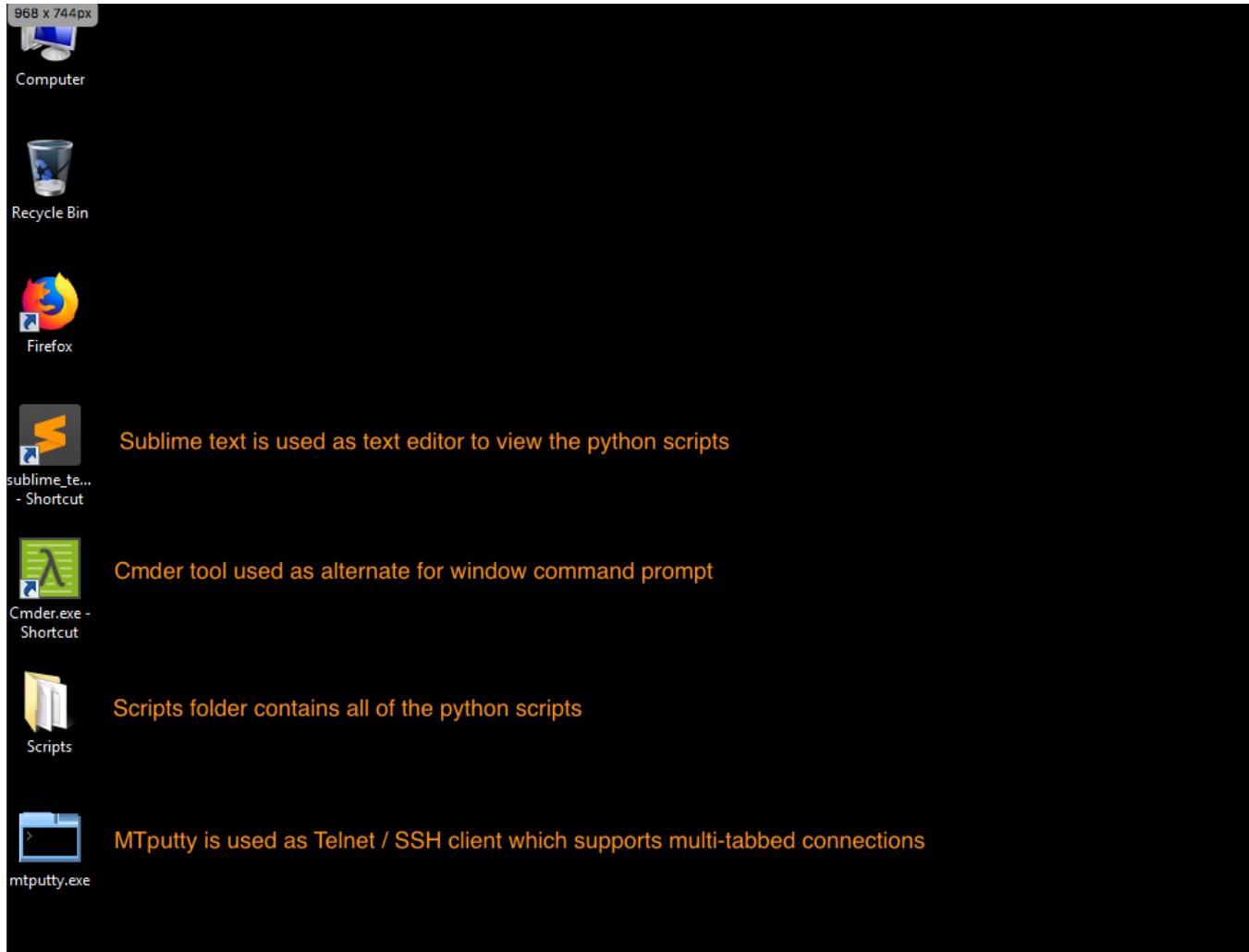
username: dcloud\demouser

password: C1sco12345



Enter the password for the RDP connection – C1sco12345

After establishing the RDP connection, you will see a similar desktop screen setup as shown below. Use the tools as needed to complete the lab tasks.



## Lab 1: Explore the bash-shell features on the Cisco Nexus 9000 Series Switches

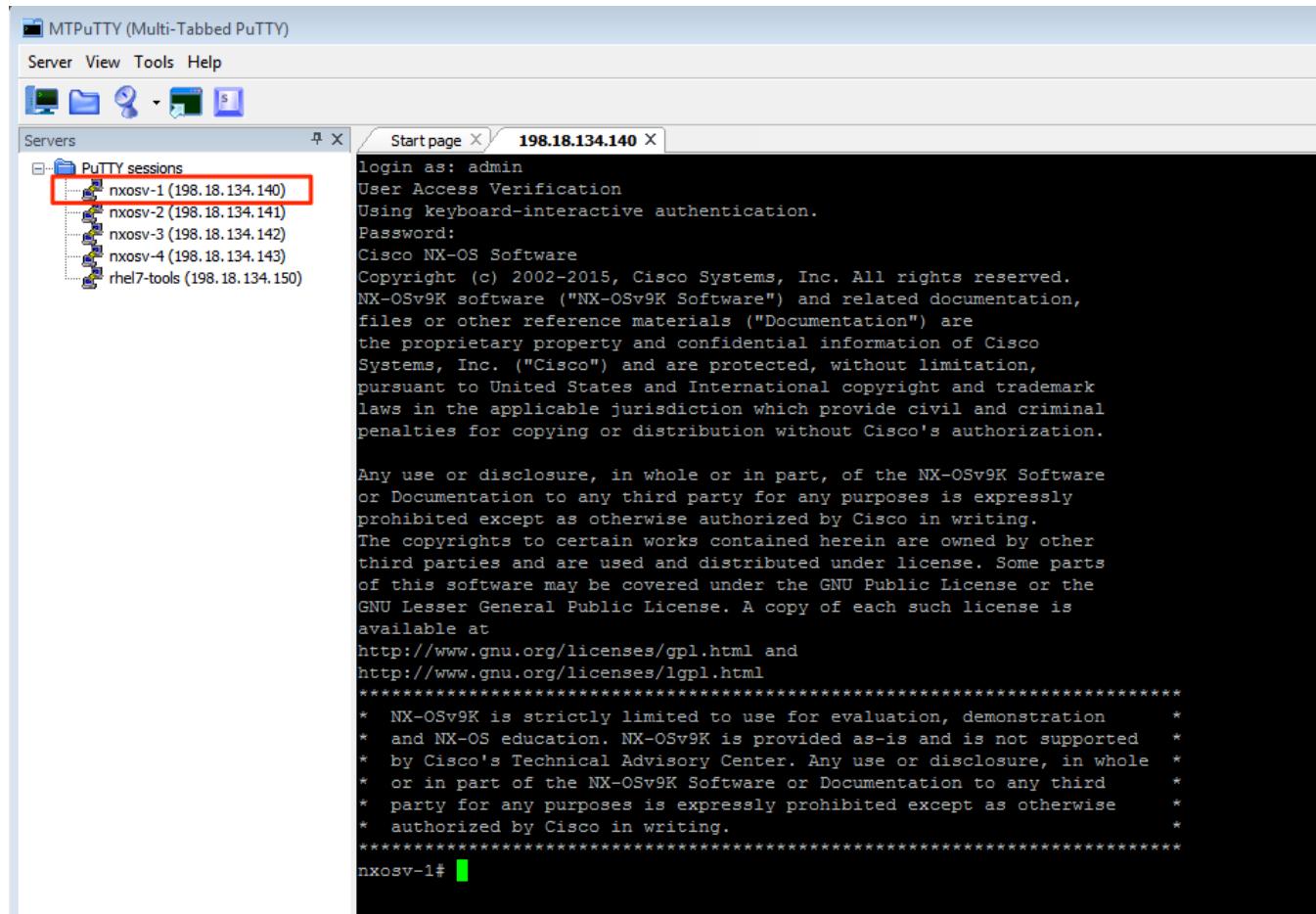
Establish a SSH connection to "nxosv-1" switch using the MTPutty client.

IP Address: 198.18.134.140

Port: 22

Username: admin

Password: C1sco12345



### Step 1.1:

In this exercise we will enable the bash shell feature, copy the running config to a temp file, edit the hostname on the temp file and then copy the temp file to the running config.

*On the switch CLI enter:*

```

nxos-v1#
nxos-v1# conf t
Enter configuration commands, one per line. End with CNTL/Z.
nxos-v1(config)# feature bash-shell
nxos-v1(config)# exit

```

```

nxos-v1# copy running-config bootflash:temp
Copy complete, now saving to disk (please wait)...
nxos-v1#
nxos-v1#

```

*switch to bash shell by typing the following command:*

```

nxos-v1# run bash
bash-4.2$
bash-4.2$ cd /bootflash/
bash-4.2$
bash-4.2$ pwd
/bootflash
bash-4.2$
bash-4.2$ ls
20151214_185114_poap_8293_init.log      nxos.7.0.3.I2.1.bin   virt_strg_pool_bf_vdc_1
cisco_node_utils-1.1.0.gem                 scripts            virtual-instance
home                           staert           virtual-instance.conf
install_puppet.sh                         temp
n9000_sample-1.0.0-7.0.3.x86_64.rpm
bash-4.2$
bash-4.2$ vi temp

```

*Change the hostname to NXOSv-1*

```

bash-4.2$
bash-4.2$ more temp
!Command: show running-config
!Time: Sun Jan 28 10:49:46 2018
version 7.0(3)I2(1)
hostname NXOSv-1
vdc nxosv-1 id 1
bash-4.2$
bash-4.2$ exit
exit
nxos-v1#
nxos-v1# copy bootflash:temp running-config
Copy complete.
NXOSv-1#

```

*NOTE: Ignore some of the syntax errors but you should see the Hostname changed*

```
N9K-04# conf t
Enter configuration commands, one per line. End with CNTL/Z.
N9K-04(config)# feature bash-shell
N9K-04(config)# exit
N9K-04# copy running-config bootflash:temp
Copy complete, now saving to disk (please wait)...
N9K-04# run bash
bash-4.2$ cd /bootflash/
bash-4.2$ pwd
/bootflash
bash-4.2$ ls
20150511_160838_poap_8048_init.log  ResetSwitchConfig
20150511_214829_poap_8086_init.log  scripts
GoldConfigDoNotDelete                temp
home                                virt_strg_pool_bf_vdc_1
n9000-dk9.7.0.3.I2.0.268.bin        virtual-instance
n9000-dplug.7.0.3.I2.0.198.gbin     virtual-instance.conf
bash-4.2$ vi temp
bash-4.2$ exit
exit
N9K-04# copy bootflash:temp running-config
```

**Figure.** Sample screen output for reference

## Step 1.2: Monitoring Processes

*In the Bash shell enter:*

```
NXOSv-1#
NXOSv-1# run bash
bash-4.2$ 
bash-4.2$ ps -aux
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        1  0.0  0.0  4224   688 ?        Ss  03:48   0:03 init [3]
root        2  0.0  0.0     0     0 ?        S  03:48   0:00 [kthreadd]
root        3  0.0  0.0     0     0 ?        S  03:48   0:00 [ksoftirqd/0]
root        4  0.0  0.0     0     0 ?        S  03:48   0:14 [kworker/0:0]
```

**Figure.** Sample screen output for reference

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	4216	700	?	Ss	Jun02	0:03	init [3]
root	2	0.0	0.0	0	0	?	S	Jun02	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	Jun02	0:00	[ksoftirqd/0]
root	6	0.0	0.0	0	0	?	S	Jun02	0:00	[migration/0]
root	7	0.0	0.0	0	0	?	S	Jun02	0:00	[watchdog/0]
root	8	0.0	0.0	0	0	?	S	Jun02	0:00	[migration/1]
root	9	0.0	0.0	0	0	?	S	Jun02	0:00	[kworker/1:0]
root	10	0.0	0.0	0	0	?	S	Jun02	0:02	[ksoftirqd/1]
root	11	0.0	0.0	0	0	?	S	Jun02	0:15	[kworker/0:1]
root	12	0.0	0.0	0	0	?	S	Jun02	0:00	[watchdog/1]
root	13	0.0	0.0	0	0	?	S	Jun02	0:00	[migration/2]
root	15	0.0	0.0	0	0	?	S	Jun02	0:02	[ksoftirqd/2]
root	16	0.0	0.0	0	0	?	S	Jun02	0:00	[watchdog/2]
root	17	0.0	0.0	0	0	?	S	Jun02	0:00	[migration/3]
root	18	0.0	0.0	0	0	?	S	Jun02	0:00	[kworker/3:0]
root	19	0.0	0.0	0	0	?	S	Jun02	0:02	[ksoftirqd/3]
root	20	0.0	0.0	0	0	?	S	Jun02	0:00	[watchdog/3]
root	21	0.0	0.0	0	0	?	S<	Jun02	0:00	[cpuset]
root	22	0.0	0.0	0	0	?	S<	Jun02	0:00	[khelper]

In the bash prompt run the top command.

```
bash-4.2$ top
```

type 'q' to quit the command

**Figure.** results of the top command

```
top - 03:02:58 up 1 day, 20:18, 2 users, load average: 1.19, 1.12, 1.14
Tasks: 261 total, 2 running, 257 sleeping, 0 stopped, 2 zombie
Cpu(s): 17.4%us, 15.5%sy, 0.0%ni, 65.1%id, 0.3%wa, 1.4%hi, 0.2%si, 0.0%st
Mem: 12282516k total, 4373996k used, 7908520k free, 178208k buffers
Swap: 0k total, 0k used, 0k free, 1838888k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
6377	root	20	0	238m	14m	7348	S	99	0.1	2644:44	l2fwder
31592	admin	20	0	323m	52m	40m	S	15	0.4	0:00.44	vsh
7421	root	20	0	233m	9640	6184	S	10	0.1	201:40.13	ecp
31591	root	20	0	225m	9412	7016	S	2	0.1	0:00.07	dcos_sshd
6211	root	20	0	511m	113m	75m	S	1	0.9	30:03.96	svc_ifc_policye
6243	root	20	0	577m	99m	62m	S	1	0.8	28:22.94	svc_ifc_eventmg
6309	root	20	0	522m	86m	47m	S	1	0.7	27:45.47	svc_ifc_confele
6315	root	20	0	327m	53m	39m	S	1	0.4	0:05.05	aaad
6452	root	20	0	807m	86m	43m	S	1	0.7	1:22.74	netstack
6202	root	20	0	335m	61m	39m	S	0	0.5	0:39.78	syslogd
31324	root	20	0	225m	9348	6944	S	0	0.1	0:00.17	dcos_sshd
31587	admin	20	0	19512	1436	976	R	0	0.0	0:00.09	top
1	root	20	0	4216	700	604	S	0	0.0	0:03.27	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0	0.0	0:00.31	ksoftirqd/0
6	root	RT	0	0	0	0	S	0	0.0	0:00.24	migration/0
7	root	RT	0	0	0	0	S	0	0.0	0:00.07	watchdog/0
8	root	RT	0	0	0	0	S	0	0.0	0:00.44	migration/1
9	root	20	0	0	0	0	S	0	0.0	0:00.00	kworker/1:0
10	root	20	0	0	0	0	S	0	0.0	0:02.55	ksoftirqd/1
11	root	20	0	0	0	0	S	0	0.0	0:15.25	kworker/0:1
12	root	RT	0	0	0	0	S	0	0.0	0:00.12	watchdog/1
13	root	RT	0	0	0	0	S	0	0.0	0:00.42	migration/2
15	root	20	0	0	0	0	S	0	0.0	0:02.71	ksoftirqd/2
16	root	RT	0	0	0	0	S	0	0.0	0:00.12	watchdog/2
17	root	RT	0	0	0	0	S	0	0.0	0:00.37	migration/3
18	root	20	0	0	0	0	S	0	0.0	0:00.00	kworker/3:0
19	root	20	0	0	0	0	S	0	0.0	0:02.80	ksoftirqd/3
20	root	RT	0	0	0	0	S	0	0.0	0:00.18	watchdog/3
21	root	0	-20	0	0	0	S	0	0.0	0:00.00	cpuset
22	root	0	-20	0	0	0	S	0	0.0	0:00.00	khelper
23	root	20	0	0	0	0	S	0	0.0	0:00.00	kdevtmpfs
24	root	0	-20	0	0	0	S	0	0.0	0:00.00	netns
25	root	20	0	0	0	0	S	0	0.0	0:00.34	sync_supers
26	root	20	0	0	0	0	S	0	0.0	0:00.00	bdi-default
27	root	0	-20	0	0	0	S	0	0.0	0:00.00	kblockd

### Step 1.3: Controlling interfaces with Linux commands

The version of code on our lab Nexus 9000 supports the iproute2(ip link show) package as well as the older net-tools(ifconfig). Spend a few minutes trying each of the commands below:

*In the Bash shell enter:*

```
bash-4.2$ip address show
```

```
bash-4.2$ip link show
```

```
bash-4.2$ifconfig
```

**Figure.** Output screen of previous commands

```
bash-4.2$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/16 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UNKNOWN qlen 1
000
    link/ether 00:50:56:91:40:80 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::250:56ff:fe91:4080/64 scope link
        valid_lft forever preferred_lft forever
14: ps-inb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9400 qdisc mq state UNKNOWN qlen 1000
    link/ether 00:00:00:01:01:01 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::200:ff:fe01:101/64 scope link
        valid_lft forever preferred_lft forever
15: veobc: <BROADCAST,UP,LOWER_UP> mtu 1494 qdisc noqueue state UNKNOWN
    link/ether 00:00:00:01:01 brd ff:ff:ff:ff:ff:ff
    inet 127.1.2.1/24 brd 127.1.2.255 scope host veobc
    inet 127.1.1.1/16 brd 127.1.255.255 scope global veobc
    inet6 fe80::200:ff:fe00:101/64 scope link
        valid_lft forever preferred_lft forever
21: ps-diag: <BROADCAST,UP,LOWER_UP> mtu 9212 qdisc pfifo_fast state UP qlen 100
    link/ether 00:00:00:00:1b:02 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::200:ff:fe00:1b02/64 scope link
        valid_lft forever preferred_lft forever
22: ps-sup-eth1: <BROADCAST,UP,LOWER_UP> mtu 9212 qdisc pfifo_fast state UP qlen 100
    link/ether 00:00:00:00:1b:03 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::200:ff:fe00:1b03/64 scope link
        valid_lft forever preferred_lft forever
23: sflow: <BROADCAST,UP,LOWER_UP> mtu 9212 qdisc pfifo_fast state UP qlen 100
    link/ether 00:00:00:00:1b:04 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::200:ff:fe00:1b04/64 scope link
        valid_lft forever preferred_lft forever
40: Eth1-1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 100
    link/ether 00:50:56:91:40:88 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::250:56ff:fe91:4088/64 scope link
        valid_lft forever preferred_lft forever
41: Eth1-2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 100
    link/ether 00:50:56:91:40:89 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::250:56ff:fe91:4089/64 scope link
```

In the Bash shell enter:

```
bash-4.2$ip link show Eth1-1  
bash-4.2$sudo ifconfig Eth1-1 down  
bash-4.2$ip link show Eth1-1  
bash-4.2$sudo ifconfig Eth1-1 up  
bash-4.2$ip link show Eth1-1
```

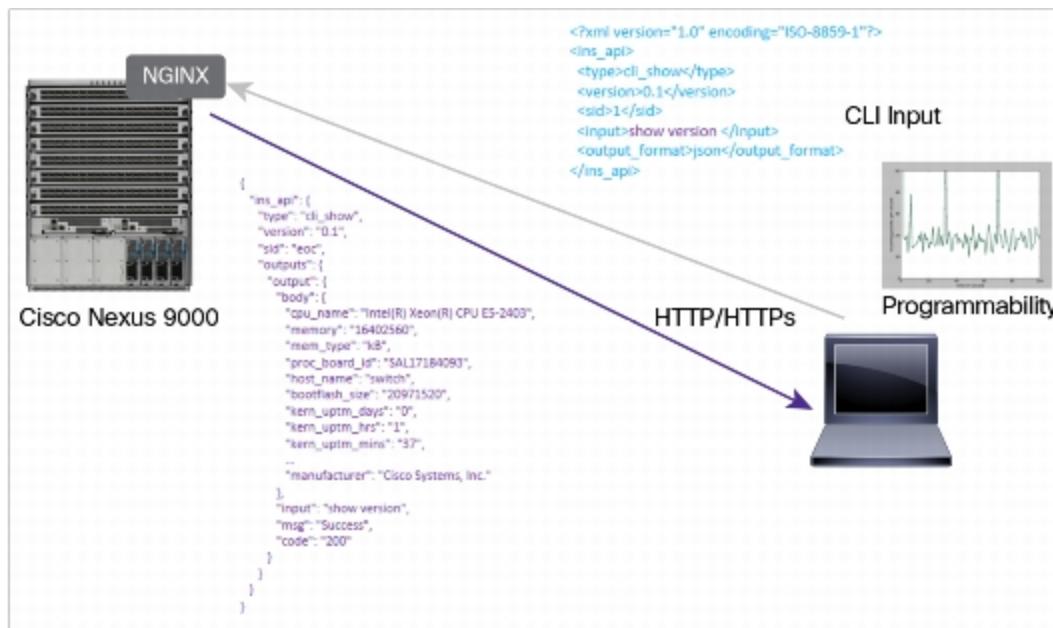
**Figure.** Output screen of previous commands

```
bash-4.2$ ip link show Eth1-1  
40: Eth1-1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEF  
    AULT qlen 100  
        link/ether 00:50:56:91:40:88 brd ff:ff:ff:ff:ff:ff  
bash-4.2$ sudo ifconfig Eth1-1 down  
bash-4.2$ ip link show Eth1-1  
40: Eth1-1: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN mode DEFAULT qlen  
    100  
        link/ether 00:50:56:91:40:88 brd ff:ff:ff:ff:ff:ff  
bash-4.2$ sudo ifconfig Eth1-1 up  
bash-4.2$ ip link show Eth1-1  
40: Eth1-1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEF  
    AULT qlen 100  
        link/ether 00:50:56:91:40:88 brd ff:ff:ff:ff:ff:ff  
bash-4.2$ █
```

## Lab 2: Manage Cisco Nexus 9000 Switch using the Cisco NX-API

Cisco NX-OS API (NX-API) allows web-based programmatic access to the Cisco Nexus 9000 Series Switches. This support is delivered through an open source web server, NGINX. NX-API exposes the complete configuration and management capabilities of the CLI through web-based APIs. The Cisco Nexus 9000 Series Switches can be instructed to publish the output of the API calls in either XML or JSON format. This comprehensive, easy-to-use API enables rapid deployment on the Cisco Nexus 9000 Series Switches.

**Figure.** Programmatic Access to Cisco Nexus 9000 Series through Cisco NX-API



### Step 2.1: Enabling NX-API

Type the following commands through CLI:

```

NXOSv-1# conf t
Enter configuration commands, one per line. End with CNTL/Z.
NXOSv-1(config)#
NXOSv-1(config)# feature nxapi
NXOSv-1(config)#
NXOSv-1(config)# exit
NXOSv-1#
NXOSv-1# show feature | inc nxapi
nxapi                  1          enabled
NXOSv-1#

```

**Figure.** Output screen of previous commands

```

N9K-02(config)# feature nxapi
N9K-02(config)# exit
N9K-02# show feature | inc nxapi
nxapi                  1          enabled
N9K-02#

```

## Step 2.2: Accessing NX-API

To Access the NX-API interface, Open the web browser and point it to switch's MGMT 0 interface ip address in the browser address bar

<https://198.18.134.140/>

Username: admin

Password: C1sco12345

in the top left box enter "show version", change the message format to json-rpc, command type to cli and click on the POST button

show version

Message format: ?

json-rpc    xml    json

Command type: ?

cli    cli\_ascii

POST    Reset

REQUEST:

```
[ { "jsonrpc": "2.0", "method": "cli", "params": { "cmd": "show version", "version": 1 }, "id": 1 } ]
```

Copy    Python

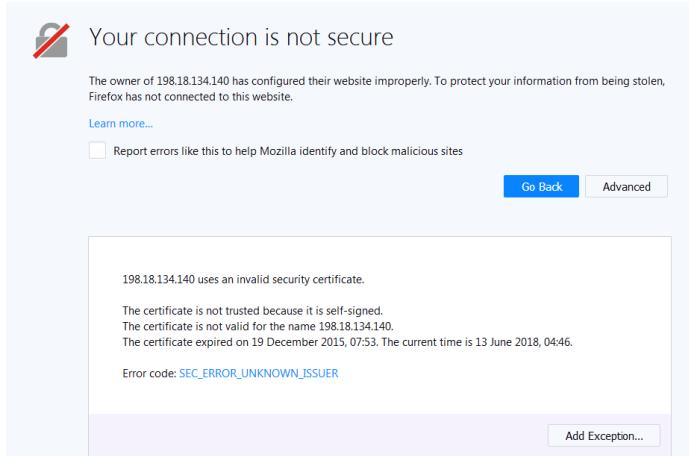
RESPONSE:

Copy

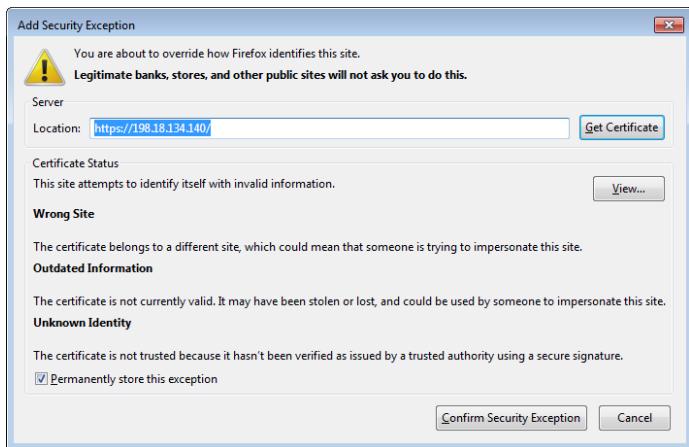
Copyright © 2014 Cisco Systems, Inc. All rights reserved.

NX-API version 1.0

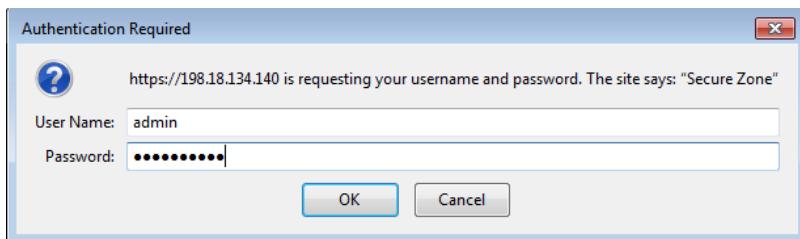
**Note: Please follow the below mentioned steps if you experience any certificate warnings.**



**Click Add Exception**



**Click on Confirm Security Exception**



**Type the login credentials**

Now you have successfully posted an API call to the switch. The API call output should look like this

The screenshot shows the Cisco NX-API Developer Sandbox interface. At the top, there's a header bar with the Cisco logo, the title "Cisco NX-API Sandbox", and a URL "https://198.18.134.140". Below the header is a navigation bar with icons for back, forward, search, and other functions. The main content area has a title "NX-API Developer Sandbox" and a "Logout" button. On the left, a command input field contains "show version". To the right of the input field are two sections: "Message format:" with options "json-rpc" (selected), "xml", and "json"; and "Command type:" with options "cli" (selected) and "cli\_ascii". Below the input field are two buttons: "POST" (blue) and "Reset" (orange). In the bottom half of the screen, there are two side-by-side code blocks. The left block is labeled "REQUEST:" and contains the JSON request message:

```
[{"jsonrpc": "2.0", "method": "cli", "params": {"cmd": "show version", "version": 1}, "id": 1}]
```

Next to the request are two buttons: "Copy" and "Python". The right block is labeled "RESPONSE:" and contains the JSON response message:

```
{
  "jsonrpc": "2.0",
  "result": {
    "body": {
      "header_str": "Cisco Nexus Operating System (NX-OS) Software\nT",
      "bios_ver_str": "",
      "kickstart_ver_str": "7.0(3)I2(1)",
      "bios_cmpl_time": "",
      "kick_file_name": "bootflash:///nxos.7.0.3.I2.1.bin",
      "kick_cmpl_time": " 9/3/2015 22:00:00",
      "kick_tmstamp": "09/04/2015 05:23:48",
      "chassis_id": "NX-OSv Chassis",
      "cpu_name": "Intel(R) Xeon(R) CPU E7- 2830 @ 2.13GHz",
      "memory": 8165884,
      "mem_type": "kB",
      "host_name": "NXOSv-1",
      "bootflash_size": 1747849,
      "kern_uptm_days": 0,
      "kern_uptm_hrs": 7,
      "kern_uptm_mins": 51
    }
  }
}
```

Next to the response are two buttons: "Copy" and "Python".

Notice the output in the response field. Next click the python button in the request window. This will generate the python code for "show version" command. Now we can use the NX-API generated python code to interact with the device via API calls.

### Step 2.3: Basic API call to Nexus device using python

The nxapi\_shver script is created from the NX-API output. Since we are getting structured data back from the device, we can retrieve any data easily by parsing and extracting the specific key:value pairs or list from the json output.

In this example we are extracting the NX-OS version running on the switch using an API call.

Open the cmder tool and type the following commands:

```
C:\Users\demouser\Desktop\Scripts
python nxapi_shver.py
```

Now you are launching the below mentioned code via CLI. The script will make an API call to the Nexus 9000 switch. Collect the show version output in the json format and print out NX-OS version.

```
import requests
import json

"""
Modify these please
"""

url='http://198.18.134.140/ins'
switchuser='admin'
switchpassword='C1sco12345'

myheaders={'content-type':'application/json-rpc'}
payload=[
{
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
        "cmd": "show version",
        "version": 1
    },
    "id": 1
}
]
response = requests.post(url,data=json.dumps(payload),
headers=myheaders,auth=(switchuser,switchpassword)).json()
print "Version: %s" % response['result']['body']['kickstart_ver_str']
```

**Figure.** Output screen off the script

```
C:\Users\demouser\Desktop\Scripts
λ python nxapi_shver.py
Version: 7.0(3)I2(1)
```

This is a very basic example of how you can interact with NX-OS via NX-API. But you can do lot more with NX-API, please refer the link for more information : <https://developer.cisco.com/site/nx-api/>

## Lab:3 Configure NX-OS via Python programming

Python is a very powerful and easy to learn programming language. Python Programming skills are mandatory for today's Network engineers, one needs to know how to write, edit, modify, and expand python scripts to utilize SSH, APIs and data models to effectively automate day to day networking tasks.

In this lab section we will look into how to leverage Python programming language and python libraries to manage the network devices efficiently. The main benefit of using python for network programmability is that, we don't need to reinvent the wheel. There are lot of open-source and community driven libraries that we can leverage to quickly get started with automating our day to day networking tasks.

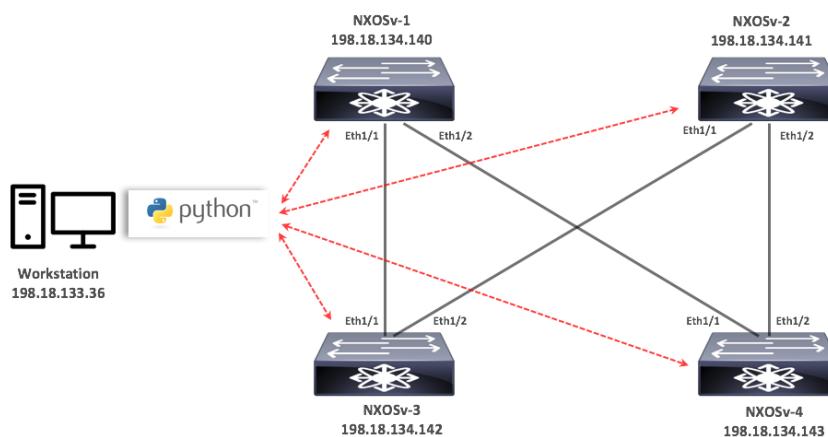
Using python programming you can automate most of the repeated network admin tasks any no. of times without any typo or human errors. So we can save lot of time wasted in doing manual way of configuring devices. But you should be very careful, "With great power comes great responsibility". Error prone scripts can break the network very fast and can cause major business impact. So the advice here is to build and test the python scripts in a safe lab environment first, not once but multiple times, before trying it out in production.

For this lab we will be using a very popular python library named Netmiko. Netmiko is an open-source python library based on Paramiko SSH library. It provides a uniform programming interface across a broad set of network devices. It also handles many of the low-level SSH details that can be time consuming and problematic when you are a beginner and getting started with programming.

Since Netmiko is based on SSH, it is still a legacy screen-scraping interface. In other words, Netmiko uses an interface primarily intended for humans (SSH) and returns text strings that have to be processed (as contrasted to using an API that returns structured data). This is where NX-API will be helpful to extract structured data from Nexus devices.

Now are moving away from manual methods of configuring and collecting information the devices to python scripts, that will handle the network administration tasks.

**Figure:** Python based network automation



**Note:** All python scripts are available in GitHub for your reference. GitHub link : <https://github.com/jagadnag/Scripts>

### Step 3.1: Building blocks of netmiko python script

First we have to import the Connection handler from the netmiko library

```
from netmiko import ConnectHandler
```

The connection parameters are collected in a Python dictionary. The connection parameters provide Netmiko with everything it needs to create the SSH connection.

```
nxos1 = {  
    'device_type': 'cisco_nxos',  
    'ip': '172.21.56.125',  
    'username': 'admin',  
    'password': 'cisco',  
}
```

*ConnectHandler is a Netmiko function that calls the necessary connection parameters and device type (cisco\_ios, cisco\_xr, cisco\_nxos, etc.) This ConnectHandler call will launch the SSH connection and login into the device.*

```
net_connect = ConnectHandler(**nxos1)
```

*.send\_command() method is used to send show commands over the channel and receive the output back. Here, we have used 'show version' command.*

```
output = net_connect.send_command('show version')
```

*Using the send\_config\_set() method, we program the network device to enter into configuration mode and execute the config\_commands and then exit the configuration mode.*

```
output = net_connect.send_config_set(config_commands)
```

*We can send only one command or more lines of commands by converting it into a simple list. The session output are stored in an output variable and then printed out for our reference. Next step you will launch a simple python script to collect show version command output and print it.*



Cisco live!

The screenshot shows a Windows desktop with two windows open. On the left is a Sublime Text window titled 'C:\Users\demouser\Desktop\Scripts\netmiko\_nxos1.py (Scripts) - Sublime Text (UNREGISTERED)'. It contains Python code for connecting to a Cisco NXOS device via SSH. On the right is a 'cmd.exe' window titled 'C:\Users\demouser\Desktop\Scripts'. It displays the output of running the script, which includes the Cisco NX-OS license, system hardware details, BIOS information, kernel uptime, last reset reason, active packages, and a command prompt at the end.

```
#!/usr/bin/env python
from netmiko import ConnectHandler

# SSH Connection Details
nxos1 = {
    'device_type': 'cisco_nxos',
    'ip': '198.18.134.140',
    'username': 'admin',
    'password': 'Cisco12345',
}

# Establish SSH to device and run show command
net_connect = ConnectHandler(**nxos1)
output = net_connect.send_command('show version')
print output
```

```
C:\Users\demouser\Desktop\Scripts
A python netmiko/nxos1.py
Cisco Nexus Operating System (NX-OS) Software
TAC support: http://www.cisco.com/tac
Documents: http://www.cisco.com/en/US/products/ns9372/tsd_products_support_series_home.html
Copyright (c) 2002-2015, Cisco Systems, Inc. All rights reserved.
The copyrights to certain works contained herein are owned by
other third parties and are used and distributed under license.
Some parts of this software are covered under the GNU Public
License. A copy of the license is available at
http://www.gnu.org/licenses/gpl.html.

NX-OSv9K is a demo version of the Nexus Operating System

Software
  BIOS: version
  NXOS: version 7.0(3)I2(1)
  BIOS compile time:
  NXOS image file is: bootflash:///nxos.7.0.3.I2.1.bin
  NXOS compile time: 9/3/2015 22:00:00 [09/04/2015 05:23:48]

Hardware
  cisco NX-OSv Chassis
  Intel(R) Xeon(R) CPU E7- 2830 @ 2.13GHz with 8165920 kB of memory.

  Device name: nxosv-1
  bootflash: 1747849 kB
  Kernel uptime is 8 day(s), 9 hour(s), 57 minute(s), 18 second(s)

Last reset
  Reason: Unknown
  System version:
  Service:

plugin
  Core Plugin, Ethernet Plugin

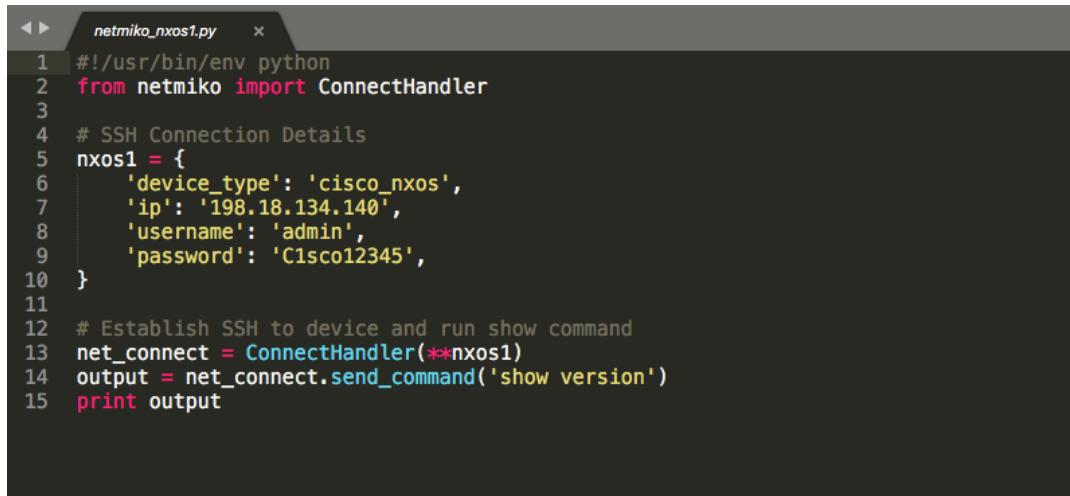
Active Package(s):
C:\Users\demouser\Desktop\Scripts
λ |
```

We recommend to have a similar desktop window setup with Sublime and cmder tools side by side, so you can view the script and output at the sametime.

### Step 3.2: Execute a show command using netmiko on one device

Let's execute our first netmiko python script. Use the cmdr to launch the python script

```
C:\Users\demouser\Desktop\Scripts  
python netmiko_nxos1.py
```



```
netmiko_nxos1.py  x  
1 #!/usr/bin/env python  
2 from netmiko import ConnectHandler  
3  
4 # SSH Connection Details  
5 nxos1 = {  
6     'device_type': 'cisco_nxos',  
7     'ip': '198.18.134.140',  
8     'username': 'admin',  
9     'password': 'Cisco12345',  
10 }  
11  
12 # Establish SSH to device and run show command  
13 net_connect = ConnectHandler(**nxos1)  
14 output = net_connect.send_command('show version')  
15 print output
```

Now you can see how easy it is to get started with python programming. Though it is a very simple python program with only few lines of code, in the following steps we will quickly iterate through it to send multiple lines of config commands or retrieve multiple show command outputs from one device or multiple devices.

The idea here is to understand the basic programmability logic first, build a base skeleton for the code and quickly iterate through it to build more complex tasks on top of it to automate the day to day network admin tasks.

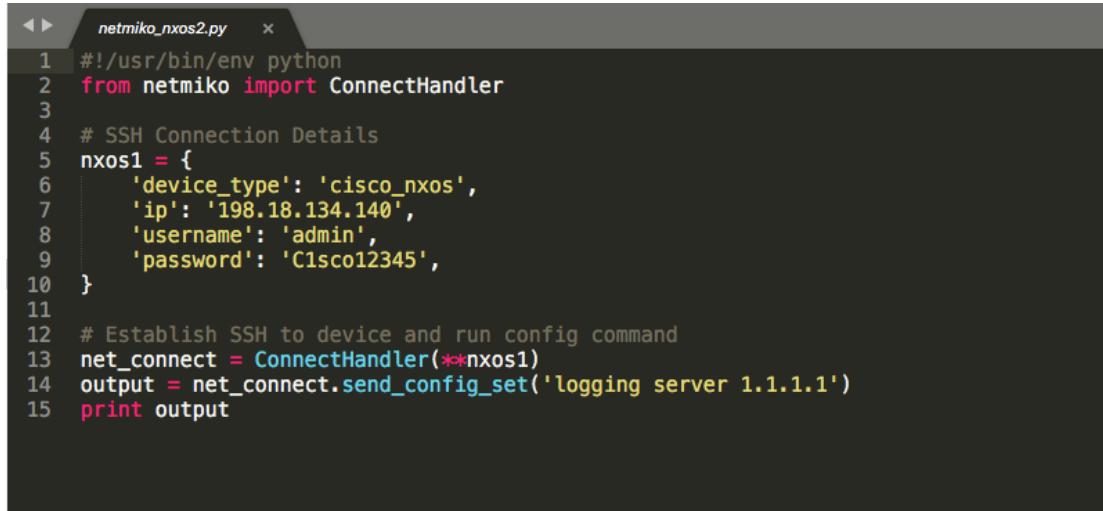
*Note: After launching the script it will take 20-30 seconds for the program to complete all the tasks and print the output.*

### Step 3.3: Execute a configuration command using netmiko on one device

Now we will launch another simple program to configure the nexus device.

*Launch the python script from the command prompt*

```
C:\Users\demouser\Desktop\Scripts  
python netmiko_nxos2.py
```



```
netmiko_nxos2.py  x  
1 #!/usr/bin/env python  
2 from netmiko import ConnectHandler  
3  
4 # SSH Connection Details  
5 nxos1 = {  
6     'device_type': 'cisco_nxos',  
7     'ip': '198.18.134.140',  
8     'username': 'admin',  
9     'password': 'Cisco12345',  
10 }  
11  
12 # Establish SSH to device and run config command  
13 net_connect = ConnectHandler(**nxos1)  
14 output = net_connect.send_config_set('logging server 1.1.1.1')  
15 print output
```

*Login to nxosv-1 switch using SSH client and verify the script configured the device correctly.*

### Step 3.4: Send multiple configuration commands to one device

In this step we will use a file named ‘basic\_config’ to send few lines of configuration commands to the nxosv-1 switch. The script will iterate through the configuration commands using a for loop and configure the device accordingly. This is where programming is very useful, it can do repeated tasks any no. of times without any typo or human errors.

*basic\_config file:*

```
logging console
logging server 1.1.1.2
vlan 10
name python_vlan_10
vlan 20
name python_vlan_20
```

*Launch the python script from the command prompt*

```
C:\Users\demouser\Desktop\Scripts
python netmiko_nxos3.py
```

```
netmiko_nxos3.py
1 #!/usr/bin/env python
2
3 from netmiko import ConnectHandler
4
5 # Sending multiple lines of config stored in a file
6 with open('basic_config') as f:
7     commands_to_send = f.read().splitlines()
8
9 # SSH Connection details
10 nxos1 = {
11     'device_type': 'cisco_nxos',
12     'ip': '198.18.134.140',
13     'username': 'admin',
14     'password': 'C1sc012345',
15 }
16
17 all_devices = [nxos1]
18
19 # Iterate through device list and configure the devices
20 for devices in all_devices:
21     net_connect = ConnectHandler(**devices)
22     output = net_connect.send_config_set(commands_to_send)
23     print output
```

*Login to nxosv-1 switch using SSH client and verify the script configured the device correctly.*

### Step 3.5: Send multiple configuration commands to many devices

In this step we will use a file named ‘more\_config’ to send more lines of configuration commands to multiple devices. Now we have extracted manually hardcoding the configuration commands, device ips and connection details like username and passwords out of the main script. Especially hardcoding username and password on the scripts are not advisable. The script requests the user to enter the login credentials to establish a SSH connection. The script now iterates through the lines in more\_config and device\_ip in a for loop to configure the device accordingly.

#### *more\_config file:*

```
logging console
logging server 1.1.1.3
ip access-list TEST_ACL
permit ip 1.1.1.0 0.0.0.255 any
permit ip 2.2.2.0 0.0.0.255 any
permit ip 3.3.3.0 0.0.0.255 any
vlan 101
name python_vlan_101
vlan 102
name python_vlan_102
feature ospf
router ospf 1
interface Ethernet1/1
ip router ospf 1 area 0
ip ospf network point-to-point
```

#### *devices\_list file:*

```
198.18.134.140
198.18.134.142
```

*Launch the python script from the command prompt*

```
C:\Users\demouser\Desktop\Scripts
python netmiko_nxos4.py
```

```

netmiko_nxos4.py
1 #!/usr/bin/env python
2
3 from netmiko import ConnectHandler
4 from getpass import getpass
5
6 # SSH username and password provided by user
7 username = raw_input('Enter your SSH username: ')
8 password = getpass()
9
10 # Sending multiple lines of config stored in a file
11 with open('more_config') as f:
12     commands_list = f.read().splitlines()
13
14 # Sending device ip's stored in a file
15 with open('devices_list') as f:
16     devices_list = f.read().splitlines()
17
18 # Iterate through device list and configure the devices
19 for devices in devices_list:
20     print 'Connecting to device ' + devices
21     ip_address_of_device = devices
22
23 # SSH Connection details
24 nxos_device = {
25     'device_type': 'cisco_nxos',
26     'ip': ip_address_of_device,
27     'username': username,
28     'password': password
29 }
30
31 net_connect = ConnectHandler(**nxos_device)
32 output = net_connect.send_config_set(commands_list)
33 print output

```

*Login to nxosv-1 and nxosv-3 switch using SSH client and verify the script configured the device correctly.*

Now you can alter the configuration lines by editing more\_config file or manage additional devices by simply adding the device ip to the devices\_list. This is the power python programming where you initially start with very simple program, streamline the logic and operation and then modify the script to start building complex tasks.

### Step 3.6: Collect multiple show command output from many devices

In this step we will use a python script to quickly verify whether all the devices are configured correctly. This is very similar to precheck and postcheck validation task when we make a configuration change in the network. Here we are using a file named 'show\_command' to send multiple show commands to multiple devices and capture the output for config verification.

#### *show\_command file:*

```
show vlan brief
show runn | in logging
show ip access-lists TEST_ACL
show ip ospf neighbors
```

*Launch the python script from the command prompt*

```
C:\Users\demouser\Desktop\Scripts
python netmiko_nxos5.py
```

```
netmiko_nxos5.py  x
1 #!/usr/bin/env python
2
3 from netmiko import ConnectHandler
4 from getpass import getpass
5
6 # SSH username and password provided by user
7 username = raw_input('Enter your SSH username: ')
8 password = getpass()
9
10 # Sending list of show commands stored in a file
11 with open('show_command') as f:
12     show_commands = f.readlines()
13
14 # Sending device ip's stored in a file
15 with open('devices_list') as f:
16     devices_list = f.read().splitlines()
17
18 # Iterate through device list
19 for devices in devices_list:
20     print 'Connecting to device ' + devices
21     ip_address_of_device = devices
22
23     # SSH Connection details
24     nxos_device = {
25         'device_type': 'cisco_nxos',
26         'ip': ip_address_of_device,
27         'username': username,
28         'password': password
29     }
30     # Iterate through show command list
31     net_connect = ConnectHandler(**nxos_device)
32     for command in show_commands:
33         output = net_connect.send_command(command)
34         print command + output + '\n'
```

*Validate the show command output printed in the command prompt.*

### **Step 3.7: Reset the lab configurations**

After completing all the above mentioned steps, please run the reset script to reconfigure the devices to its original state.

*Launch the python script from the command prompt*

```
C:\Users\demouser\Desktop\Scripts  
python netmiko_resetlab.py
```

*Login to nxosv-1 and nxosv-3 via SSH and verify whether all the configurations are removed correctly.*

There are lot more you can do with netmiko library. Pls refer the below mentioned URL for more information

<https://pynet.twb-tech.com/blog/automation/netmiko.html>

## Conclusion

You have successfully completed the lab. Please handover the WISP Lab card to the front desk. Thanks for attending this lab, please share your valuable feedback.

## Reference

*Additional Reading and Reference:*

<https://developer.cisco.com/video/net-prog-basics/>  
<https://learninglabs.cisco.com/tracks>

<http://shop.oreilly.com/product/0636920042082.do>

