A close up of a sign

Description automatically generated 

Cisco Live Barcelona 2020

DEVWKS-2626: Simple and Scalable NX-OS Automation with Ansible

Gerard Sheehan, Product Manager, IBNG, [gsheehan@cisco.com](mailto:gsheehan@cisco.com)

[1 Lab topology and access 2](#_Toc23153596)

[1.1 Topology 2](#_Toc23153597)

[1.2 Environment 3](#_Toc23153598)

[2 Ansible Configuration 3](#_Toc23153599)

[3 Our first Ansible playbook: VLAN provisioning 4](#_Toc23153600)

[4 Idempotency 5](#_Toc23153601)

[5 Map the VLAN to a VNI 6](#_Toc23153602)

[6 Add the L2VNI to the EVPN overlay 6](#_Toc23153603)

[7 Remove the L2VNI 8](#_Toc23153604)

[8 Inventory for clean separation of logic and resources 8](#_Toc23153605)

[9 Performance at scale: the “aggregate” feature 10](#_Toc23153606)

[9.1 Performance issue with the classic loop 10](#_Toc23153607)

[9.2 Let’s aggregate 12](#_Toc23153608)

[10 Our good old CLI 13](#_Toc23153609)

[10.1 Basic usage of nxos\_config 13](#_Toc23153610)

[10.2 Leverage an existing file with CLIs 15](#_Toc23153611)

[10.3 Jinja2 integration for efficiency and modularity 16](#_Toc23153612)

[11 Conclusion 17](#_Toc23153613)

[12 For More Information 18](#_Toc23153614)

[13 Advanced topic: Ansible as a NETCONF YANG engine 18](#_Toc23153615)

[13.1 First payload 18](#_Toc23153616)

[13.2 Templated payload 22](#_Toc23153617)

# Lab topology and access

## Topology

Welcome to this NX-OS programmability lab! Our goal today is to explore the unique advantages of Ansible with NX-OS via real-world data center use cases.

We’re going to run the Ansible controller in a Docker container. The Ansible controller will automate NX-OS via the management network.

A screenshot of a video game

Description automatically generated

## Environment

Our laptop is already setup with 3 terminal windows:

1. One on our Docker container, from which we will run Ansible.
2. One on the first Nexus 9K.
3. One on the second Nexus 9K.

If the windows are missing, ask the presenter.

# Ansible Configuration

Go to the terminal window which has a shell prompt *root@studentXX:~#*, where *XX* is our student ID.

The Ansible Controller will be executed from this shell.

*/etc/ansible/hosts* contains the hosts that Ansible can talk to. They’re organized by groups. We will use the group *nxos\_vteps*, which contains our switches. Let’s take a look:

root@student42:~# head /etc/ansible/hosts

[nxos\_vteps]

n9kv-1

n9kv-2

The access method and credentials of our group is defined in */etc/ansible/group\_vars/nxos\_vteps*. The setup script has also preconfigured our student username for *ansible\_user*. Let’s take a look:

root@student42:~# cat /etc/ansible/group\_vars/nxos\_vteps

---

ansible\_network\_os: nxos

ansible\_connection: network\_cli

ansible\_user: student42

ansible\_ssh\_pass: cisco

The host and group files allow us to factorize the definition of the host groups and their credentials.

In this lab, the password is stored in clear-text for simplicity purposes. In a real-world deployment, we should either encrypt the password via Ansible Vault, or use SSH certificates.

We’re now ready to automate our NX-OS network with Ansible!

# Our first Ansible playbook: VLAN provisioning

VLAN provisioning is one of the most common actions that a network admin needs to do. It’s also something that is repeatable and dynamic. Therefore, it’s the perfect task to start automating with Ansible.

Go to *~/Ansible/Workshop*:

root@student42:~# cd ~/Ansible/Workshop

Look at the Ansible playbook *01.vlan.yml*. This playbook provisions a VLAN on every VTEP. Take some time to understand its structure and don’t hesitate to ask any questions. The following examples are with VLAN 42 which would be different for every student.

---

- name: Create VLAN

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Create VLAN

nxos\_vlan:

vlan\_id: 42

admin\_state: up

The playbook leverages the module *nxos\_vlan*, which is documented here: [https://docs.ansible.com/ansible/latest/modules/nxos\_vlan\_module.html#nx os-vlan-module](https://docs.ansible.com/ansible/latest/modules/nxos_vlan_module.html#nx	os-vlan-module).

On the terminal window connected to n9kv-1 switch, check that our VLAN does not already exist (**replace 42 by our VLAN ID**.)

n9kv-1# sh vlan id 42

VLAN 42 not found in current VLAN database

We can verify that the VLAN does not exist either on the second switch, if we want to.

Go back to the container shell, and execute the playbook with the command *ansible-playbook 01.vlan.yml*. We will see an output like this:

root@student42:~/Ansible/Workshop# ansible-playbook 01.vlan.yml

PLAY [Create VLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Create VLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-2]

changed: [n9kv-1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

Verify that the VLAN has been provisioned on the switches (**use your VLAN ID instead of 42**):

n9kv-1# sh vlan id 42

VLAN Name Status Ports

---- -------------------------------- --------- -------------------------------

42 VLAN0042 active

# Idempotency

Remove the VLAN from **one** switch. Keep the VLAN on the other switch. **Use your VLAN ID instead of 42.**

n9kv-1# conf

Enter configuration commands, one per line. End with CNTL/Z.

n9kv-1(config)# no vlan 42 # REPLACE BY OUR VLAN

Run the playbook again. Notice how Ansible is only touching the switch on which the VLAN was removed:

root@student42:~/Ansible/Workshop# ansible-playbook 01.vlan.yml

PLAY [Create VLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Create VLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [n9kv-2]

changed: [n9kv-1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=0 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

Ansible is actually checking if the VLAN already exists on the switch, before trying to create it. If the VLAN is already there, nothing is done.

This is called idempotency: the result of performing an operation once is exactly the same as the result of performing it repeatedly without any intervening actions. It’s a core value-prop of Ansible, and we fully leverage it in every NX-OS Ansible module that we have developed.

This is achieved natively without any modification to the playbook, which is only 8 lines long. Can we imagine how many lines of code this would require if we were to implement this manually in Python?

# Map the VLAN to a VNI

VXLAN EVPN is the de-facto overlay technology of choice. So, let’s augment the VLAN provisioning to map the VLAN to a L2VNI. We’ll reuse the previous task and add a parameter to the *nxos\_vlan* module invocation. Look at the playbook *02.vlan\_vni.yml*:

---

- name: Create VLAN

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Create VLAN

nxos\_vlan:

vlan\_id: 42

mapped\_vni: 20042

admin\_state: up

In the playbook, the VLAN and VNI IDs have already been replaced with the right values.

Run the playbook:

root@student42:~/Ansible/Workshop# ansible-playbook 02.vlan\_vni.yml

PLAY [Create VLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Create VLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-2]

changed: [n9kv-1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

And check that the mapping has been configured on the VTEPs. **Use our VLAN ID instead of 42.**

n9kv-1# sh run vlan

vlan 42

vn-segment 20042

We don’t need to check on all VTEPs... We can start to trust Ansible now 😀.

# Add the L2VNI to the EVPN overlay

Let’s now add the L2VNI to the overlay (represented by the nve1 interface), and make it part of EVPN. Native Ansible NX-OS modules exists for that purpose. Get familiar with the modules by browsing their documentation:

<https://docs.ansible.com/ansible/latest/modules/nxos_vxlan_vtep_vni_module.html>  
<https://docs.ansible.com/ansible/latest/modules/nxos_evpn_vni_module.html>

Take a look at the playbook *03.evpn.yml*.

---

- name: Create L2VNI

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Add L2VNI to Overlay

nxos\_vxlan\_vtep\_vni:

interface: nve1

vni: 20042

multicast\_group: 239.239.239.100

suppress\_arp: true

- name: Add L2VNI to EVPN

nxos\_evpn\_vni:

vni: 20042

route\_distinguisher: auto

route\_target\_both: auto

Let’s execute the playbook, but this time we’ll take a peek underneath the hood. Invoke ansible-playbook with the option *-vvv* for an increased verbosity:

root@student42:~/Ansible/Workshop# ansible-playbook 03.evpn.yml -vvv

We’re now able to see what exact CLIs is Ansible sending to the switch. For example:

changed: [n9kv-1] => {

"changed": true,

"commands": [

"interface nve1",

"member vni 20042",

"no mcast-group",

"mcast-group 239.239.239.100",

"suppress-arp",

"member vni 20042"

],

Verify that the configuration has been properly pushed to the VTEPs:

n9kv-1# sh run int nve1

interface nve1

no shutdown

host-reachability protocol bgp

**member vni 20042**

**suppress-arp**

**mcast-group 239.239.239.100**

n9kv-1# sh run bgp

...

evpn

**vni 20042 l2**

**rd auto**

**route-target import auto**

**route-target export auto**

# Remove the L2VNI

Let’s assume that the hosts using our L2VNI have been decommissioned from the fabric. Now the L2VNI needs to be removed from all VTEPs.

Take a look at the playbook *04.evpn\_destroy.yml*:

---

- name: Destroy L2VNI

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Remove L2VNI from Overlay

nxos\_vxlan\_vtep\_vni:

interface: nve1

vni: 20042

state: absent

Since we’re removing something, we might want to make sure the commands are right before actually applying them. Run the playbook first with the *-vvv* and *--check* options:

root@student42:~/Ansible/Workshop# ansible-playbook 04.evpn\_destroy.yml -vvv --check

changed: [n9kv-2] => {

"changed": true,

"commands": [

"interface nve1",

"no member vni 20042"

],

Once it looks good to us, execute the playbook without the options:

root@student42:~/Ansible/Workshop# ansible-playbook 04.evpn\_destroy.yml

Removing the L2VNI from EVPN would follow the same logic.

# Inventory for clean separation of logic and resources

Up to now, for simplicity reasons, we have included the resources in the playbook. However, it’s always cleaner to separate the logic and resources.

Therefore, we now define the VNIs in a separate file:

root@student42:~/Ansible/Workshop# cat 05.vnis.yml

---

l2vnis:

- { id: 30042, mcast\_group: 239.239.239.103 }

- { id: 30142, mcast\_group: 239.239.239.103 }

root@student42:~/Ansible/Workshop#

And in the playbook *05.evpn\_resource.yml*, we import this file. Consequently, the variable *l2vnis* becomes available in the playbook. We invoke the same NX-OS modules as earlier, but this time we loop on the list of L2 VNIs. For every element (referred by *item*), we access its attributes *id* and *mcast\_group*.

---

- name: Create L2VNI

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Include VNIs list

include\_vars:

file: 05.vnis.yml

- name: Add L2VNI to Overlay

nxos\_vxlan\_vtep\_vni:

interface: nve1

vni: "{{ item.id }}"

multicast\_group: "{{ item.mcast\_group }}"

suppress\_arp: true

loop: "{{ l2vnis }}"

- name: Add L2VNI to EVPN

nxos\_evpn\_vni:

vni: "{{ item.id }}"

route\_distinguisher: auto

route\_target\_both: auto

loop: "{{ l2vnis }}"

Let’s run it:

root@student42:~/Ansible/Workshop# ansible-playbook 05.evpn\_resource.yml

PLAY [Create L2VNI] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Include VNIs list] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [93180-EX-1]

TASK [Add L2VNI to Overlay] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [93180-EX-1] => (item={u'id': 30042, u'mcast\_group': u'239.239.239.103'})

changed: [93180-EX-1] => (item={u'id': 30142, u'mcast\_group': u'239.239.239.103'})

TASK [Add L2VNI to EVPN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [93180-EX-1] => (item={u'id': 30042, u'mcast\_group': u'239.239.239.103'})

changed: [93180-EX-1] => (item={u'id': 30142, u'mcast\_group': u'239.239.239.103'})

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

93180-EX-1 : ok=3 changed=2 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

If we needed separate resources depending on the host, we could place the resource file in */etc/ansible/host\_vars/<host-name>.* Ansible will automatically pick up the right variable depending on the switch being programmed.

# Performance at scale: the “aggregate” feature

## Performance issue with the classic loop

Let’s assume that we have a lot of static routes to configure. We’ll do that using the module [*nxos\_static\_route*](https://docs.ansible.com/ansible/latest/modules/nxos_static_route_module.html), and a loop. Take a look at the playbook *06.static\_route\_classic.yml*.

---

- name: Create static routes

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Create static routes

nxos\_static\_route:

prefix: "{{ item.prefix }}"

next\_hop: "{{ item.next\_hop }}"

vrf: 42

loop:

- { prefix: 1.1.1.1/32, next\_hop: 192.168.0.1 }

- { prefix: 2.2.2.2/32, next\_hop: 192.168.0.1 }

- { prefix: 3.3.3.3/32, next\_hop: 192.168.0.1 }

There’s 20 routes total. They get created under a VRF that is unique to us, and is named after our student ID.

Run the playbook and measure how much time it takes using *time*:

root@student42:~/Ansible/Workshop# **time** ansible-playbook 06.static\_route\_classic.yml

PLAY [Create static routes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Create static routes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-1] => (item={u'prefix': u'1.1.1.1/32', u'next\_hop': u'192.168.0.1'})

changed: [n9kv-2] => (item={u'prefix': u'1.1.1.1/32', u'next\_hop': u'192.168.0.1'})

changed: [n9kv-1] => (item={u'prefix': u'2.2.2.2/32', u'next\_hop': u'192.168.0.1'})

...

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

real 0m27.624s

user 0m10.908s

sys 0m3.229s

root@student42:~/Ansible/Workshop#

We should see a number around 30/40 seconds. As we can see, it’s a bit long. And it would be increase linearly with the number of routes. The same holds true anytime we use a loop like this, because it essentially executes all the tasks one by one:

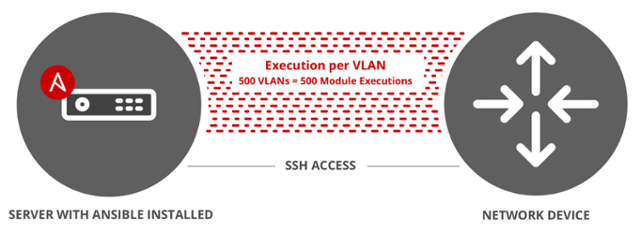


Figure 1 - Loop Method – source: https://www.ansible.com/blog/accelerate-ansible-networking-aggregate-resources

If we remove some of our routes from a switch and run the playbook again with *-vvv*, we would see that indeed Ansible executes the module every time for every route!

changed: [n9kv-2] => (item={u'prefix': u'1.1.1.1/32', u'next\_hop': u'192.168.0.1'}) => {

"changed": true,

"commands": [

"vrf context 42",

"ip route 1.1.1.1/32 192.168.0.1"

],

...

changed: [n9kv-2] => (item={u'prefix': u'2.2.2.2/32', u'next\_hop': u'192.168.0.1'}) => {

"changed": true,

"commands": [

"vrf context 42",

"ip route 2.2.2.2/32 192.168.0.1"

],

## Let’s aggregate

Starting from Ansible 2.4, the *aggregate* option brings a better performance by aggregating the items as a single task instead of a collection of discrete loops.

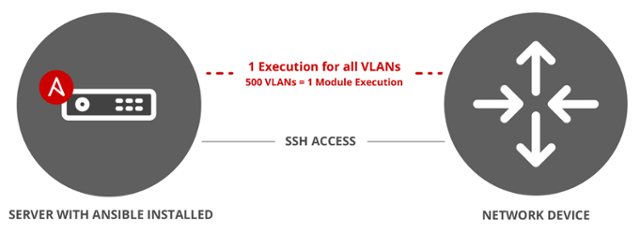


Figure 2 - Aggregate Method – source: https://www.ansible.com/blog/accelerate-ansible-networking-aggregate-resources

Let’s see how aggregation can help us here. First, let’s remove our VRF from the switches:

root@student42:~/Ansible/Workshop# ansible-playbook 07.vrf\_remove.yml

Now, let’s look at the playbook *08.static\_route\_aggregate.yml* and check how we now use the *aggregate* feature.

---

- name: Create static routes with aggregate

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Create static routes

nxos\_static\_route:

vrf: 42

aggregate:

- { prefix: 1.1.1.1/32, next\_hop: 192.168.0.1 }

- { prefix: 2.2.2.2/32, next\_hop: 192.168.0.1 }

- { prefix: 3.3.3.3/32, next\_hop: 192.168.0.1 }

...

OK, let’s run the playbook and time it!

root@student42:~/Ansible/Workshop# time ansible-playbook \

08.static\_route\_aggregate.yml

PLAY [Create static routes with aggregate] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Create static routes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-1]

changed: [n9kv-2]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

real 0m13.485s

user 0m2.671s

sys 0m0.617s

root@student42:~/Ansible/Workshop#

13 seconds vs 28 seconds: it’s a 50+% improvement. Not bad, isn’t it? Those numbers may vary depending on our environment, but in general we’ll see this type of improvement, if not better.

Again - if we’re curious, we could remove the whole VRF, and re-run the playbook with *-vvv* to see that the commands get aggregated:

changed: [10.60.0.107] => {

"changed": true,

"commands": [

"vrf context student10",

"ip route 1.1.1.1/32 192.168.0.1",

"vrf context student10",

"ip route 2.2.2.2/32 192.168.0.1",

"vrf context student10",

"ip route 3.3.3.3/32 192.168.0.1",

# Our good old CLI

## Basic usage of nxos\_config

While there is an extensive coverage of NX-OS features in Ansible, there are still small gaps. This particularly true for the newest functionalities. For such features, we can use the [*nxos\_config*](https://docs.ansible.com/ansible/latest/modules/nxos_config_module.html) Ansible module. We’ve already used the module [*nxos\_command*](https://docs.ansible.com/ansible/latest/modules/nxos_command_module.html) - this one is similar for configuration commands, with a few more advanced capabilities.

Let’s use it to configure Streaming Telemetry, which is not currently supported by a native Ansible module. We want to add a destination group as shown below.

telemetry

destination-group STUDENT\_ID

ip address 1.1.1.1 port 80 protocol HTTP encoding JSON

Let’s check the playbook *09.telemetry\_cli.yml*.

---

- name: Configure a streaming telemetry destination

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Configure destination

nxos\_config:

parents:

- telemetry

- destination-group 42

lines:

- ip address 1.1.1.1 port 80 protocol HTTP encoding JSON

Run it:

root@student42:~/Ansible/Workshop# ansible-playbook 09.telemetry\_cli.yml

PLAY [Configure a streaming telemetry destina\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Configure destination] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-1]

changed: [n9kv-2]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

Now confirm that the destination-group has been created. Replace 42 by our student ID.

n9kv-2# sh run telemetry | section "destination-group 42"

destination-group 42

ip address 1.1.1.1 port 80 protocol HTTP encoding JSON

n9kv-2#

Note that even though the parent cli *destination-group* didn’t exist, Ansible created it. Pretty cool right?

One more thing. It’s not because we use CLI in Ansible that we lost idempotency. Idempotency is still here and to verify it, we can remove the destination-group from only one VTEP. If we feel curious, let’s run the playbook again and we’ll notice that only this VTEP was touched by Ansible:

root@student42:~/Ansible/Workshop# ansible-playbook 09.telemetry\_cli.yml

PLAY [Configure a streaming telemetry destination] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Configure destination] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [n9kv-1]

changed: [n9kv-2]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=0 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

## Leverage an existing file with CLIs

It’s common to have existing repositories of config CLIs. It doesn’t necessarily make sense to throw them away and rewrite them as Ansible playbooks from scratch. We could instead keep them, and use Ansible as the transport. So, in addition to the unsupported feature use cases, *nxos\_config* can be also used to take advantage of Ansible in this situation. It also allows us to re-use all the Ansible switch groups and credentials definitions.

Let’s assume that we’ve been using Streaming Telemetry before we started to automate our network with Ansible. We have an existing file containing the telemetry configuration, that the network admin sends to the switch via copy/paste or SSH. And this time, there’s more collectors. It’s in the file *10.telemetry\_cli.txt*. Take a look at it:

root@student42:~/Ansible/Workshop# cat 10.telemetry\_cli.txt

telemetry

destination-group STUDENT\_ID

ip address 1.1.1.10 port 80 protocol HTTP encoding JSON

ip address 1.1.1.1 port 80 protocol HTTP encoding JSON

ip address 1.1.1.2 port 80 protocol HTTP encoding JSON

ip address 1.1.1.3 port 80 protocol HTTP encoding JSON

ip address 1.1.1.4 port 80 protocol HTTP encoding JSON

ip address 1.1.1.5 port 80 protocol HTTP encoding JSON

ip address 1.1.1.6 port 80 protocol HTTP encoding JSON

ip address 1.1.1.7 port 80 protocol HTTP encoding JSON

ip address 1.1.1.8 port 80 protocol HTTP encoding JSON

ip address 1.1.1.9 port 80 protocol HTTP encoding JSON

root@student42:~/Ansible/Workshop#

*nxos\_config* is able to take in input a file containing the CLIs to be sent to the switch. Take a look at the playbook *10.telemetry\_cli\_file.yml*:

---

- name: Configure streaming telemetry destinations

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Configure destinations

nxos\_config:

src: 10.telemetry\_cli.txt

Execute the playbook and check the configuration on a switch:

root@student42:~/Ansible/Workshop# ansible-playbook 10.telemetry\_cli\_file.yml

PLAY [Configure streaming telemetry destinations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Configure destinations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-1]

changed: [n9kv-2]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

n9kv-1# sh run telemetry | section "destination-group 42"

destination-group 42

ip address 1.1.1.10 port 80 protocol HTTP encoding JSON

ip address 1.1.1.1 port 80 protocol HTTP encoding JSON

ip address 1.1.1.2 port 80 protocol HTTP encoding JSON

ip address 1.1.1.3 port 80 protocol HTTP encoding JSON

ip address 1.1.1.4 port 80 protocol HTTP encoding JSON

ip address 1.1.1.5 port 80 protocol HTTP encoding JSON

ip address 1.1.1.6 port 80 protocol HTTP encoding JSON

ip address 1.1.1.7 port 80 protocol HTTP encoding JSON

ip address 1.1.1.8 port 80 protocol HTTP encoding JSON

ip address 1.1.1.9 port 80 protocol HTTP encoding JSON

n9kv-1#

## Jinja2 integration for efficiency and modularity

Those CLI text files can actually be more intelligent and become generic templates, thanks to Jinja2. This is how the previous telemetry configuration can be made cleaner and factorized via a Jinja2 *for* loop. The file is *11.telemetry\_cli.j2*:

telemetry

destination-group {{ my\_destination\_group }}

{% for i in range(1, 11) %}

ip address 1.1.1.{{ i }} port 80 protocol HTTP encoding JSON

{% endfor %}

Make sure that we understand that template, and ask questions to the presenter if it’s not 100% clear.

Now the good news is that Ansible integrates very well with Jinja2! Therefore, we can directly leverage these templates without the need for any change.

We define the Jinja2 variables inside the playbook. Ansible will use them to generate the final configuration based on the template, and push it to the switches. Ansible becomes a real engine for templatized configs.

In this example, we’ll define the value of *my\_destination\_group* in the Ansible playbook.

But first, go to one switch and remove our destination group. Replace 42 with your student ID.

n9kv-1# conf

Enter configuration commands, one per line. End with CNTL/Z.

n9kv-1(config)# telemetry

n9kv-1(config-telemetry)# no destination-group 42 # REPLACE 42 BY OUR ID

Take a look at the playbook *11.telemetry\_cli\_jinja.yml*.

---

- name: Configure streaming telemetry destinations

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Configure destinations

nxos\_config:

src: 11.telemetry\_cli.j2

vars:

my\_destination\_group: 42

Run it:

root@student42:~/Ansible/Workshop# ansible-playbook 11.telemetry\_cli\_jinja.yml

PLAY [Configure streaming telemetry destinations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Configure destinations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [n9kv-2]

changed: [n9kv-1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=0 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

Verify on the VTEPs that the Jinja2 template resulted in the right configuration:

n9kv-1# sh run telemetry | section "destination-group 42"

destination-group 42

ip address 1.1.1.10 port 80 protocol HTTP encoding JSON

ip address 1.1.1.1 port 80 protocol HTTP encoding JSON

ip address 1.1.1.2 port 80 protocol HTTP encoding JSON

ip address 1.1.1.3 port 80 protocol HTTP encoding JSON

ip address 1.1.1.4 port 80 protocol HTTP encoding JSON

ip address 1.1.1.5 port 80 protocol HTTP encoding JSON

ip address 1.1.1.6 port 80 protocol HTTP encoding JSON

ip address 1.1.1.7 port 80 protocol HTTP encoding JSON

ip address 1.1.1.8 port 80 protocol HTTP encoding JSON

ip address 1.1.1.9 port 80 protocol HTTP encoding JSON

n9kv-1#

# Conclusion

We are done with the lab. We have used Ansible to automate the provisioning of a variety of NX-OS functionalities, including VXLAN BGP EVPN and Streaming Telemetry.

We experimented different Ansible methods:

* Abstracting the CLI.
* Or re-using existing CLIs, static or templated.

And we did it at scale.

Good job!

We can reuse the Docker container at any time, since it’s publicly available from the Docker Hub: <https://hub.docker.com/r/ndelecro/nx-os-programmability/>.

If we don’t have a physical Nexus switch available, we can install one on our laptop as well using [Nexus 9Kv and VirtualBox](https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/9-x/nx-osv/configuration/guide/b_Cisco_Nexus_9000v_9x/b_Cisco_Nexus_9000v_9x_chapter_010.html).

Enjoy!

# For More Information

The documentation for all Ansible NX-OS modules is at:  
<https://docs.ansible.com/ansible/latest/modules/list_of_network_modules.html#nxos>.

There’s also an extensive Ansible support for ACI. See the following links:  
<https://docs.ansible.com/ansible/latest/scenario_guides/guide_aci.html>  
<https://blogs.cisco.com/developer/automating-cisco-aci-with-ansible-eliminates-repetitive-day-to-day-tasks>

<http://ansible.ciscolive.com> is an online Ansible lab for both NX-OS and ACI.

# Advanced topic: Ansible as a NETCONF YANG engine

## First payload

Just like CLIs, we might have existing NETCONF payloads. Or they might want to standardize on NETCONF payloads to avoid being too dependent on Ansible. In both cases, Ansible can really help because it provides a module named [*netconf\_config*](https://docs.ansible.com/ansible/latest/modules/netconf_config_module.html) that is able to deal with NETCONF. We can click on the link to go to the documentation and get familiar with it.

*Netconf\_config* takes a NETCONF payload in input. Let’s assume that we don’t have a payload already. So, let’s generate it ourselves! Open a web browser, and point it to the IP of one of our switches to access the sandbox. Use our NX-OS credentials to authenticate:

* User = studentXX, where XX is our student ID
* Password = cisco

A screenshot of a social media post

Description automatically generated

This time, we’re going back to VXLAN L2VNI automation. Copy/paste the following CLIs in the CLI box. Replace the 42 in both the VLAN ID and the VNI ID by our student ID. For example, student 10 will have VLAN 10 and VNI 20010.

vlan 42

vn-segment 20042

name vlan-42

For example:

A screenshot of a social media post

Description automatically generated

On the right-hand side, we’ll notice options for the Method and Message format. For the Method, choose *RESTCONF (Yang)*. For the Message format, choose *xml.*

A screenshot of a cell phone

Description automatically generated

Now click on *Convert*. The YANG payload has been automatically generated from our CLI input.

A screenshot of a social media post

Description automatically generated

For convenience, this payload has already been placed in the file *12.vxlan\_yang.txt*. The *config* tags and the XML namespace have been added. Take a look at it:

<config>

<System xmlns="http://cisco.com/ns/yang/cisco-nx-os-device">

<bd-items>

<bd-items>

<BD-list>

<fabEncap>vlan-42</fabEncap>

<accEncap>vxlan-20042</accEncap>

<name>vlan-42</name>

</BD-list>

</bd-items>

</bd-items>

</System>

</config>

Let’s now use Ansible to send this payload via NETCONF to the VTEPs. This is done by the playbook *12.vxlan\_netconf.yml*:

---

- name: VXLAN NETCONF

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: VXLAN

netconf\_config:

datastore: running

content: "{{ lookup('file', './12.vxlan\_yang.xml') }}"

Edit */etc/ansible/group\_vars/nxos\_vteps* to set *ansible\_connection* to *netconf*:

---

ansible\_network\_os: nxos

ansible\_connection: **netconf**

ansible\_user: student42

ansible\_ssh\_pass: cisco

Remove our VLAN from the switches, and execute the playbook:

root@student42:~/Ansible/Workshop# ansible-playbook 12.vxlan\_netconf.yml

PLAY [VXLAN NETCONF] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [VXLAN] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-2]

changed: [n9kv-1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=1 changed=1 unreachable=0 failed=0

n9kv-2 : ok=1 changed=1 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

## Templated payload

Let’s try to make it a bit more generic. Today, *netconf\_config* cannot deal with Jinja2 directly, but we can generate the payload separate from a generic template, using the [*template*](https://docs.ansible.com/ansible/latest/modules/template_module.html) module. It will enable us to avoid hardcoding the variables in the payload file, and instead define them in the playbook.

The templatized YANG payload is *13.vxlan\_yang.j2*:

<config>

<System xmlns="http://cisco.com/ns/yang/cisco-nx-os-device">

<bd-items>

<bd-items>

<BD-list>

<fabEncap>vlan-{{vlan\_id}}</fabEncap>

<accEncap>vxlan-{{vxlan\_id}}</accEncap>

<name>{{vlan\_name}}</name>

</BD-list>

</bd-items>

</bd-items>

</System>

</config>

With a first task, we’re going to have Ansible generate the payload from the Jinja2 template and the variables provided. And the second task is exactly the same as earlier.

Let’s look at the playbook *13.vxlan\_netconf\_generic.yml*.

---

- name: VXLAN NETCONF

hosts: nxos\_vteps

gather\_facts: no

tasks:

- name: Generate the YANG payload from the template

template:

src: 13.vxlan\_yang.j2

dest: 13.vxlan\_yang\_payload.xml

vars:

vlan\_id: 42

vxlan\_id: 20042

vlan\_name: vlan-42

- name: Send the YANG payload to the switch via NETCONF

netconf\_config:

datastore: running

src: 13.vxlan\_yang\_payload.xml

Remove the VLAN from one of the switches, and run the playbook:

root@student42:~/Ansible/Workshop# ansible-playbook 13.vxlan\_netconf\_generic.yml

PLAY [VXLAN NETCONF] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Generate the YANG payload from the template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [n9kv-1]

ok: [n9kv-2]

TASK [Send the YANG payload to the switch via NETCONF] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [n9kv-2]

changed: [n9kv-1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n9kv-1 : ok=2 changed=2 unreachable=0 failed=0

n9kv-2 : ok=2 changed=0 unreachable=0 failed=0

root@student42:~/Ansible/Workshop#

We can see the YANG payload that Ansible has automatically generated and sent for us:

root@student42:~/Ansible/Workshop# cat 13.vxlan\_yang\_payload.xml

<config>

<System xmlns="http://cisco.com/ns/yang/cisco-nx-os-device">

<bd-items>

<bd-items>

<BD-list>

<fabEncap>vlan-42</fabEncap>

<accEncap>vxlan-20042</accEncap>

<name>vlan-42</name>

</BD-list>

</bd-items>

</bd-items>

</System>

</config>

root@student42:~/Ansible/Workshop#