BLOG POSTS: DRAFT, COPYRIGHT © 2019

	Content
Post Title	Ansible for N9K VXLAN EVPN
Post Date	
Attributed To	Peter Welcher
Written By	Peter Welcher
Reviewed By (Name & Date)	
Reviewed By (Name & Date)	

Meta Title (55 characters including spaces)	
Meta Description (156 characters including spaces)	
Target Keywords	
Categories	Technology
Tags	N/A
Call to Action	N/A
Image	Put a brief description of what the image should be or note that a file is attached. DO NOT paste the image into this Word doc; send it as a separate file.

Note: Naming convention for files as they go back and forth

- Original writer names file with "_V1" at the end (e.g., blogtitle_V1)
- First reviewer, makes edits and renames with initials at end (e.g., blogtitle_V1_af)
- If another reviewer, again add initials to end to keep the string of reviewers (e.g., blogtitle_V1_af_pw)
- When original writer gets it post back with edits, she makes revisions and saves the file as V2 (e.g. blogtitle_V2) – then reviewers continue as above with initials
- When post is complete, it is saved with "Final" and the post date at the end (e.g. blogtitle_FINAL_022012)

COPY FOR POST:

I'm currently involved in a VXLAN EVPN Multi-Site design. At some point we'll need to deploy the devices. Anticipating that, I've been tracking: what tools are available to automate deploying that?

I'll confess to a bias towards an Ansible-based solution, since (a) the customer uses Ansible and is trying to get to 100% automated, and (b) I have read up on Ansible and needed more serious hands-on time. Please note, I'm not an Ansible expert – still learning Ansible (and many other things). The subtitle I considered for this blog was "How I Survived Ansible and Made It Work". And you too can make it work.

This blog covers some of what I've found, what I've tried in a minimal lab setting, and how to tweak some of what I found to work.

One objective of this blog is to help those who haven't had much time to get hands-on with automation and Ansible in particular to get started, by demonstrating quick value! To that end, this is a bit long for a blog, more of a tutorial, step-by-step lab.

I've included file snippets and sometimes whole files, for ease of reference. Also because when you look at the various files used, the brevity stands out: how relatively simple files can lead to complex configurations in multiple devices. Of course, that's do to how heavily repetitive VXLAN EVPN configurations are – they lend themselves to a templated approach.

The set of files and some other related materials are also available on github.

RESOURCES

Resource #1: Folks at Cisco have developed a set of Ansible modules for configuring Nexus switches for VXLAN EVPN (and other things). There is a long list of modules for configuring various aspects. Here's the first part of the list:

Nxos

- nxos_aaa_server – Manages AAA server global configuration
- nxos_aaa_server_host – Manages AAA server host-specific configuration
- nxos_acl – Manages access list entries for ACLs
- nxos_acl_interface – Manages applying ACLs to interfaces
- nxos_banner – Manage multiline banners on Cisco NXOS devices
- nxos_bgp – Manages BGP configuration
- nxos_bgp_af – Manages BGP Address-family configuration
- nxos_bgp_neighbor – Manages BGP neighbors configurations
- nxos_bgp_neighbor_af – Manages BGP address-family's neighbors configuration
- nxos_command – Run arbitrary command on Cisco NXOS devices
- nxos_config – Manage Cisco NXOS configuration sections
- nxos_evpn_global – Handles the EVPN control plane for VXLAN
- nxos_evpn_vni – Manages Cisco EVPN VXLAN Network Identifier (VNI)
- nxos_facts – Gets facts about NX-OS switches
- nxos_feature – Manage features in NX-OS switches
- nxos_file_copy – Copy a file to a remote NXOS device
- nxos_gir – Trigger a graceful removal or insertion (GIR) of the switch
- nxos_gir_profile_management – Create a maintenance-mode or normal-mode profile for GIR
- nxos_hsrp – Manages HSRP configuration on NX-OS switches
- nxos_igmp – Manages IGMP global configuration
- nxos_igmp_interface – Manages IGMP interface configuration
- nxos_igmp_snooping – Manages IGMP snooping global configuration
- nxos_install_os – Set boot options like boot, kickstart image and issu
- nxos_interface – Manages physical attributes of interfaces
- nxos_interface_ospf – Manages configuration of an OSPF interface instance
- nxos_ip_interface – Manages L3 attributes for IPv4 and IPv6 interfaces **(D)**
- nxos_l2_interface – Manage Layer-2 interface on Cisco NXOS devices
- nxos_l3_interface – Manage L3 interfaces on Cisco NXOS network devices
- nxos_linkagg – Manage link aggregation groups on Cisco NXOS devices
- nxos_lldp – Manage LLDP configuration on Cisco NXOS network devices
- nxos_logging – Manage logging on network devices
- nxos_ntp – Manages core NTP configuration
- nxos_ntp_auth – Manages NTP authentication
- nxos_ntp_options – Manages NTP options
- nxos_nxapi – Manage NXAPI configuration on an NXOS device

Resource #2: The CiscoLive 2019 (San Diego) presentation BRKDCN-2025 was an interesting read in this regard (and reminded me of the above modules). The speaker, Nicolas Delecroix has some interesting content posted on github. His talk covered using some of that for operations, e.g. bulk NXOS upgrades, compliance checking, and also NETCONF to configure Nexus switches.

Resource #3: There's also this github site.

Resource #4: Cisco's ACI. In this case, the customer specifically has ACI and does not wish to use it for this site.

Resource #5: Cisco's DCNM 11 is yet another possible answer. Your Cisco or favorite consultant's team will likely be glad to demo it via dCloud for you.

> **Caveat:** there can be snags in the demo lab environment, where NXOS features aren't properly loaded or config commands didn't take. This seems like what I've seen in VIRL when I start a model with 10 or more devices in it: a couple come up a bit weird. CPU oversubscription perhaps? Anyway, I've worked with the demo 4 times now and had to fiddle each time with something. To be clear, the issues are **not** a DCNM problem.

Canned Solution / Demo

Early on, Google lead me to this github site. It appears to be a clone of a folder in Resource #3 above, or maybe vice versa.

Initially it looked rather sparse – how could this possibly be a solution? After some Ansible refresh, I realized the approach uses Roles, which are a powerful way to automate, in this case centered on the different, yes, roles in VXLAN EVPN: leaf and spine switches.

Key point: Look again at the files in the previou github site. They really are all that's needed to configure a 2 spine, 4 leaf VXLAN EVPN fabric – and could easily scale to many more leaf switches!

We'll go over how this works in the remainder of this blog.

Test Setup

I don't happen to currently have 6 Nexus switches handy in a lab. How could I test?

My answer was to run Ansible, etc. on an Ubuntu VM on my Mac under Fusion, while simultaneously running a (one!) Nexus 9K virtual VM (version 9.2.3).

The VM's were respectively allocated 5 GB and 6GB of RAM, out of my total 16 GB. I also exited all but the Mac apps I needed to minimize swap / RAM use, although that didn't seem to make much difference.

For the Ansible host, I already had an Ubuntu VM set up. I added pip3 for python, and then used pip3 to install Ansible 2.8. Murphy's Law of nested problems to solve meant I also had to first figure out how to configure Ubuntu 18.04 DNS – hadn't needed it up to this point. I ended up editing /etc/resolv.conf so that apt-get would work, and then apt-get worked to pull pip3.

For the N9Kv, see this link for some basic reference information. I pulled the image from cisco.com here. Open it in Fusion, install socat on Mac, use console connection to do basic setup of IP address on Ethernet1/1 and username/password. In Fusion, I set up the Ethernet1/1 interface in Fusion to be on the same subnet as the Mac.

I then set up key-based authentication in support of Ansible SSH.

On the Ubuntu system:

```
ssh-keygen -t rsa -b 4096 -C "foo@bar.com"
```

(Make that ending email address whatever is appropriate).

When requested, supply the filename to save to, say ~/.ssh/foo_rsa

Then use your admin account SSH password with SCP to copy that file to the Nexus:

```
cd ~/.ssh
scp foo_rsa.pub admin@<ip of NXOS device>:
```

Check for it in bootflash on the Nexus:

```
dir
```

On the Nexus, configure:

```
username foo sshkey file bootflash:///foo_rsa.pub
```

Note: I got wrapped around the axle as this wasn't working. Doh! You have to create username foo with a standard password first! (I'll blame a very un-helpful error message 'Error could not extract information for foo, user may not be present' – which I thought referred to the file contents, plus not found by Googling.)

Git Clone a Code Copy

Now get yourself a working code copy. To make my life easier, I've got a Mac folder that is mounted as a share by Ubuntu. That lets me use the Mac for many things, like Mac sublimetext app, while running tools like Ansible from Ubuntu.

So I went to the right sub-folder of that Ubuntu share, and did:

```
git clone https://github.com/mtarking/cisco-nxos-ansible-vxlan-evpn.git
```

That created a local copy under cisco-nxos-ansible-vxlan-evpn. (Whew!) Let's rename that folder, or at least refer to it, as "mytest" for brevity.

Folders for Ansible Roles

Here is the folder and file structure for this project:

```
mytest/
    ansible.cfg
    Dockerfile
    hosts
    host_vars/
        mt-l1.yml
        mt-l2.yml
        mt-l3.yml
        mt-l4.yml
        mt-s1.yml
        mt-s2.yml
    README.md
    roles/
        leaf/
            tasks/
                main.yml
            vars/
                main.yml
        spine/
            tasks/
                main.yml
            vars/
                main.yml
    site.yml
```

CH-CH-CHANGES

(I hated that song … but obviously part of it is stuck in my brain … ear-worm.)

Some changes to the git clone are needed because:

- The original was likely used with a real lab, and the N9Kv has some differences, like not supporting MTU 9216.
- Some of the nxos Ansible module parameter have changed or gone away: parameters or the allowed values of those parameters.
- I wanted to figure out which controlling settings could be put into which of the files, rather than on one long CLI.

I'll pat myself on the back here, for slogging thru the experimentation and web page re-reading needed to get things working, then refine.

FILES AND CHANGES

Let's look at the Ansible project files and their purposes, and at the same time discuss the changes needed.

/ETC/HOSTS

You do need to edit your /etc/hosts, or deal with hostname resolution, so that the names in the hosts file below resolve to the appropriate lab management addresses. Otherwise you may be trying (and hopefully failing) to reconfigure something named mt-l1 on the Internet. That'd be rude!

MYTEST/ANSIBLE.CFG

I added this file to test whether some of the parameters could be moved to it. I ended up putting the ones I was interested in elsewhere. I understand this currently as the counterpart of ansible-playbook command options. (Yes, I need to better understand which ones can be used where.)

```
[defaults]
inventory=./hosts
```

MYTEST/HOSTS

The hosts file lists the hosts, the routers or switches to be configured.

It is broken up into groups for convenience. I think of the groups as the devices for which a common set of configuration commands are suitable. (with appropriate variable values plugged in). For VXLAN EVPN, that would be leaf and spine switches.

I added the changes shown in red.

```
[all:vars]
ansible_connection=network_cli
ansible_network_os=nxos
user=something
# This makes the provider block happy, is not actually used

[leaf]
mt-l1
#mt-l2
#mt-l3
#mt-l4

#[spine]
#mt-s1
#mt-s2
```

The initial block changes under [all:vars] are because I didn't need the settings that were there, and added the ones shown removes the need to specify those settings on the ansible command line (with -c and -e flags respectively).

All but one of the hosts are commented out because I only have one N9Kv running. I then re-ran with mt-l1 commented out and [spine] and mt-s1 uncommented out.

MYTEST/SITE.YML

The blocks in this file are groups of hosts to be configured, matching those in square brackets in the hosts file above.

```
---

#- hosts: spine
#  roles:
#    - role: spine

- hosts: leaf
  gather_facts: no
  remote_user: admin
  roles:
    - role: leaf

#- hosts: spine
#  gather_facts: no
#  remote_user: admin
#  roles:
#    - role: spine

- hosts: all
  gather_facts: no
  remote_user: admin
```


```
#  tasks:
#    - name: save config
#      nxos_save_config: host={{ inventory_hostname }}
```

I understand this as a top-level tasks list. It basically (and implicitly) says to do the leaf tasks to the leaf hosts, then do the spine tasks to the spine hosts.

My idea here is to do a leaf switch run, then comment that block out and uncomment the spine block. This is because of having the single N9Kv. Normally, if you had 6 switches, you'd just leave both blocks uncommented.

The tasks block at the end isn't needed because the leaf and spine tasks do the config save.

I added the gather_facts and remote_user fields. They apparently have to be in the tasks, and this seemed like the way to do that with the least repetition.

MYTEST/ROLES/LEAF/VARS/MAIN.YML

I've included the whole revised file.

I left nxos_provider in since it is all over the tasks files, and since having it means being able to switch the mode of connection. Right now, the modules recommend using network_cli.

```
---

  nxos_provider:
    username: "{{ user }}"
    # 'user' is set in the hosts file, otherwise needs
    # to be on CLI invocation
    #password: "{{ pwd }}" -- pjw: using PKI
    transport: network_cli
    host: "{{ inventory_hostname }}"

  features:
  - { feature: bgp }
  - { feature: ospf }
  - { feature: "nv overlay" }
  - { feature: pim }
  - { feature: interface-vlan }
  - { feature: vnseg_vlan }

  l3_interfaces:
  - { interface: Ethernet1/49 }
  - { interface: Ethernet1/50 }
  - { interface: Ethernet1/51 }
  - { interface: Ethernet1/52 }
```

```
  l2_interfaces:
  # pjw: Eth1/1 is my mgmt connection
  - { interface: Ethernet1/2 }
  - { interface: Ethernet1/3 }

  ospf_process_id: Underlay

  ospf_area: 0

  asn: 65001

  address_families:
  - { afi: l2vpn, safi: evpn }
  - { afi: ipv4, safi: unicast }

  bgp_neighbors:
  - { remote_as: 65001, neighbor: 10.2.1.1, update_source: Loopback0 }
  - { remote_as: 65001, neighbor: 10.2.1.2, update_source: Loopback0 }

  rp_address: 100.2.1.1

  # I cut the # of VLANs and VRFs to speed things up -- pjw

  vlans_l2vni:
  - { vlan_id: 11, vni_id: 10011, addr: 10.0.11.1, mask: 24, mcast_grp:
239.0.0.11, vrf: Tenant-1 }
  - { vlan_id: 12, vni_id: 10012, addr: 10.0.12.1, mask: 24, mcast_grp:
239.0.0.12, vrf: Tenant-1 }
  #- { vlan_id: 13, vni_id: 10013, addr: 10.0.13.1, mask: 24, mcast_grp:
239.0.0.13, vrf: Tenant-1 }

... (more of the same omitted) ...

  vlans_l3vni:
  - { vlan_id: 10, vni_id: 10000, vrf: Tenant-1 }
  - { vlan_id: 20, vni_id: 20000, vrf: Tenant-2 }
...

  vrfs:
  - { vrf: Tenant-1, vni_id: 10000, afi: ipv4, safi: unicast }
  - { vrf: Tenant-2, vni_id: 20000, afi: ipv4, safi: unicast }
...
```

I replaced Ethernet1/1 with 1/3, since my management connection is 1/1, and changing it breaks SSH connectivity.

I commented out a lot of the lines to speed up the configuration run. I felt that having fewer than 5 Tenants and many VLANs still demonstrates what the final configuration will look like. I've included them here to make the point about how easily the vars file can be scaled up to add more interfaces, vlans, VRF's, etc. No magic, just templates and loops.

MYTEST/ROLES/LEAF/TASKS/MAIN.YML

For brevity, I'm not including the whole file. It is the long list of configuration tasks that need to be done to build a leaf switch configuration.

I commented out the MTU 9216 block: the N9Kv doesn't like that and the Ansible script fails.

Changed:

```
- name: CONFIGURE L2 INTERFACES/SWITCHPORTS (ACCESS)
  nxos_switchport:
    interface: "{{ item.interface }}"
    mode: access
    access_vlan: 11
    provider: "{{ nxos_provider }}"
  with_items: "{{ l2_interfaces }}"
```

This was commented out in the original, and just needs access-vlan without an 's' on the end. I figured, why not include it back in, and it seems to work fine (unless applied to the interface Ethernet1/1 the SSH session connects via, of course).

Here's how to understand this block:

- Take the list named l2_interface (which is defined in the associate leaf vars main.yml file) above, relevant part shown above.
- Loop through the L2 interfaces list values.
- Run the Ansible module "nxos_switchport" with parameters interface, mode, and access vlan. The interface parameter is to be set to the interface parameter of the current loop item.

Here's another block, which configures tenant VRFs into the BGP block.

```
- name: CONFIGURE TENANT VRFs (cont'd)
  nxos_vrf_af:
    vrf: "{{ item.vrf }}"
    afi: ipv4
    #safi: unicast
    route_target_both_auto_evpn: yes
    state: present
    #m_facts: true
    provider: "{{ nxos_provider }}"
  with_items: "{{ vrfs }}"
```

The nxos_vrf_af module doesn't support safi now, nor m_facts, so I commented those parameters out. Otherwise Ansible complains about them.

One more fix:

```
  - name: SAVE RUN CONFIG TO STARTUP CONFIG
    nxos_config:
      save_when: always
      provider: "{{ nxos_provider }}"
```

The nxos Ansible module now wants “save_when” not “save”, and “always” not “yes”.

MYTEST/ROLES/SPINE/TASKS/MAIN.YML

Ditto re MTU and save (i.e. similar to the leaf/tasks/main.yml changes).

MYTEST/ROLES/SPINE/VARS/MAIN.YML

This has the same changes to the provider block as above (not shown).

I commented out the Ethernet1/1 part since that’s my management connection to the VM, added.

I also changed Ethernet 2/1 to 1/5, etc. – there are no 2/x interfaces in the virtual N9K.

```
l3_interfaces:
#- { interface: Ethernet1/1 }
# pjw: this is my mgmt connection, not needed
# for config-building demo purposes
- { interface: Ethernet1/2 }
- { interface: Ethernet1/3 }
- { interface: Ethernet1/4 }
- { interface: Ethernet1/5 }
- { interface: Ethernet1/6 }
- { interface: Ethernet1/7 }
- { interface: Ethernet1/8 }
```

RUNNING ANSIBLE

I’ve been running Ansible in the top-level folder “mytest” as follows:

```
ansible-playbook site.yml
```

With the above files and folder structure, that’s all it takes!

Conclusion

Sample final configurations for one leaf and one spine can be found on github. I also included sample configurations built by DCNM 11 on dCloud for VXLAN EVPN Multi-Site. If you want to compare, look at just the spine and leaf configurations.

If time permits, I may explore what else is needed to do VXLAN EVPN Multi-Site with Ansible. It certainly looks feasible to do it using the provided Ansible modules.

I'd like to thank those who created the original github repos and/or presented on this topic. That (a) helped get me moving on working with Ansible, and (b) helped get me started. That's the intent of this blog: I'm still learning, but I'm hoping to share to help others become rapidly productive within this Ansible / VXLAN context.

Comments

Comments are welcome, both in agreement or constructive disagreement about the above. I enjoy hearing from readers and carrying on deeper discussion via comments. Thanks in advance!

Hashtags: #CiscoChampion #TechFieldDay #TheNetCraftsmenWay

Twitter: @pjwelcher

Disclosure Statement

[INSERT the usual IMAGES HERE: 20 Year CCIE and Cisco Champions **2019** as per recent blogs]

NetCraftsmen Services

Did you know that NetCraftsmen does network /datacenter / security / collaboration design / design review? Or that we have deep UC&C experts on staff, including @ucguerilla? For more information, contact us at <<insert suitable link here>>.

Social Media:

Facebook::

Twitter::

LinkedIn::

Google+::