# Chapter 3 (Arista DataCenter)

**Chapter Introduction**

In this chapter, we will outline how to automate Arista switches in a typical Data Centre environment in a spine and leaf architecture. We will explore how to interact with Arista devices using Ansible and how to deploy VLANs and VXLAN in a BGP/EVPN Setup on the Arista switches using various Ansible modules. We will base our illustration based on the below sample network diagram of a basic Spine/Leaf Data Centre Network.



Below are the Software releases that this chapter is based on

* Ansible Machine Running Ubuntu 16.04
* Ansible 2.8
* Arista vEOS running EOS 4.20.1F

The main recipes covered in this chapter is shown below

* Building Ansible Network Inventory.
* Connecting and User Authentication to Arista Devices.
* Enabling eAPI on Arista Devices.
* Configuring Basic System settings on Arista Devices.
* Configuring Physical Interfaces on Arista Devices.
* Configuring IP addresses on Arista Devices.
* Configuring Underlay BGP on Arista Devices.
* Configuring Overlay BGP EVPN on Arista Devices.
* Configuring VLANs on Arista Devices.
* Configuring VXLAN on Arista Devices.
* Retrieving Operational Data from Arista Devices

**Building Network Inventory**

In this chapter we outline how to build an Ansible Inventory for the Arista Spine/Leaf Data Centre Network

**Getting Ready**

We create a new folder that will host all the files that we will create in this chapter. The new folder is named ***ch3\_arista\_netops***.

**How to do it..**

* Inside the new folder (ch3\_arista\_netops) we create ***hosts*** file with the below content

$ cat hosts  
  
[leaf]

leaf01 ansible\_host=172.20.1.41

leaf02 ansible\_host=172.20.1.42

leaf03 ansible\_host=172.20.1.43

leaf04 ansible\_host=172.20.1.44

[spine]

spine01 ansible\_host=172.20.1.35

spine02 ansible\_host=172.20.1.36

[arista:children]

leaf

spine

* create ansible.cfg file as shown below

$ cat ansible.cfg  
[defaults]  
inventory=./hosts  
retry\_files\_enabled=False  
gathering=explicit  
host\_key\_checking=False

**How it works..**

We built the ansible inventory using the ***hosts*** file and we defined multiple groups in order to group the different devices in our topology into these groups as follows

* We created the ***leaf*** group which reference all the leaf switches in our topology.
* We created the ***spine*** group which group reference all the spine switches in our topology.
* We created the ***arista*** group which group reference both the leaf and spine groups.

Finally, we create the ***ansible.cfg*** file and configure it to point to our ***hosts*** file to be used as ansible inventory file and we disable the setup module which is not needed when running ansible against network nodes.

**Connecting and Authenticating to Arista Devices**

In this recipe, we will outline how to connect to Arista Devices from Ansible via SSH in order to start managing the devices from Ansible. We are going to use Username and passwords to authenticate to the Arista Devices in our topololgy.

**Getting Ready**

In order to follow along with this recipe, an ansible inventory file should be constructed as per the previous recipe, also IP reachability between the Ansible Control machine and all the devices in the network must be implemented.

**How to do it..**

* Inside the folder ch3\_arista\_netops create the folder group\_vars.
* Inside the group\_vars folder create the YAML file **arista.yml** with the below contents

ansible\_network\_os: eos

ansible\_connection: network\_cli

ansible\_user: ansible

ansible\_ssh\_pass: ansible123

* On the Arista Switches we configure the username/password and enable SSH as shown below

!

username ansible privilege 15 role network-admin secret sha512 $6$mfU4Ei0AORd6rage$5YObhOI1g0wNBK5onaKDpYJhLZ9138maJKgcOznzFdpM25Tf3rb0PWSojUSMtRQY0Y7.cexCFj5aFLY17tuNU1

!  
  
!

management ssh

idle-timeout 300

authentication mode password

login timeout 300

!

* On the Arista Switches configure the management Interface with the correct IP addresses and place them in the required management VRF as shown below

vrf definition MGMT

!  
ip routing vrf MGMT  
!

interface Management1

vrf forwarding MGMT

ip address $Ansible\_host$

no lldp transmit

no lldp receive

!

**How it works..**

We specify the username and password that we will configure on all the arista switches in the **arista.yml** file under the group\_vars directory. This will apply these parameters to all the arista switches in our inventory. On the arista switches we setup this username and password and enable SSH as well as setup the correct IP address (the one used in the ansible\_host in our inventory) on the management interface and configure the management VRF and associate the management interface with this VRF.  
At this stage we are using the **network\_cli** connection method so as to use SSH to connect to arista switches and we can verify that that the ansible controller can reach and correctly login to the devices with the below command

$ ansible arista -m ping  
  
leaf03 | SUCCESS => {

"ansible\_facts": {

"discovered\_interpreter\_python": "/usr/bin/python3"

},

"changed": false,

"ping": "pong"

}

leaf04 | SUCCESS => {

"ansible\_facts": {

"discovered\_interpreter\_python": "/usr/bin/python3"

},

"changed": false,

"ping": "pong"

}  
<-- Output Omitted for bevirty -->

**Enable eAPI on Arista Devices**

In this recipe, we will outline how to enable eAPI on Arista Devices. eAPI is a REST API on Arista devices which simplify the management of Arista Devices and provide a consistent and robust API to manage arista devices. This task is critical since we will use the eAPI in all the future recipe to manage the arista Device.

**Getting Ready**

As a prerequisite for this recipe, an ansible inventory file must be present as well as the SSH Authentication is deployed and working as per the previous recipe.

**How to do it..**

* Create a new playbook called **pb\_arista\_dc\_fabric.ym**l as show below

$ cat pb\_arista\_dc\_fabric.yml

- name: " Play 1: Deploy Config on All Arista Switches"

hosts: arista

tasks:

- name: "P1T1: Enable eAPI"

eos\_eapi:

https\_port: 443

https: yes

http: yes

state: started

vrf: "{{global.mgmt\_vrf}}"

vars:

ansible\_connection: network\_cli

tags: eapi

* Create a new file **all.yml** inside the group\_vars as shown below

$ cat group\_vars/all.yml

global:

mgmt\_vrf: MGMT

* Update the **arista.yml** file inside the group\_vars as shown below

$ cat group\_vars/arista.yml | grep ansible\_connection

ansible\_connection: httpapi

**How it works..**

In order to start interacting with the Arista Devices via eAPI we need to enable it first, thus we need to SSH into the device initially and enable eAPI. That is why in this recipe we are using the network\_cli ansible connection in order to connect with the Arista Devices via traditional SSH. Since we are going to use the eAPI API in all interactions with Arista Devices in all coming recipes , we enabled the network\_cli only for the eos\_eapi task in this playbook via the **vars** attribute. We modify the ansible\_connection attribute in the **arista.yml** to use the **httpapi** (eAPI) connection so as to use this API in all the future tasks when we interact with the devices.

We create a new Playbook called pb\_arista\_dc\_fabric.yml and in the first task we use the eos\_eapi module to enable the eAPI protocol on the remote Arista Devices. We specify the HTTPs port that we will use and we specify that we will enable HTTPs. We also specify that the eAPI should be enabled on the management vrf that we configured on the managment Interface.

Once we run the playbook we will see that all the Arista Devices are configured with eAPI as shown below

!

management api http-commands

no shutdown

!

vrf MGMT

no shutdown

!

**Configuring Generic System Options on Juniper Devices**

In this recipe, we will outline how to configure some generic system options like hostname, DNS servers and provision users on Juniper devices

**Getting Ready**

To follow along with this recipe, an ansible inventory is assumed to be already setup and eAPI is enabled on all arista Devices as per the previous recipe.

**How to do it..**

* Update the **all.yml** file inside the group\_vars folder as shown below

$ cat group\_vars/all.yml  
  
config\_dir: configs

global:

mgmt\_vrf: MGMT

site: DC1

users:

- name: ansible

password: ansible123

privilege: 15

role: network-admin

* Create a new folder called **tasks** and add the below files

$ cat tasks/build\_config\_dir.yml

---

- name: "Create Config Directory"

file: path={{config\_dir}} state=directory

run\_once: yes

- name: "Create Per host directory"

file: path={{config\_dir}}/{{inventory\_hostname}} state=directory  
  
  
$ cat tasks/build\_config.yml

---

- name: "System Configuration"

template:

src: "{{ansible\_network\_os}}/mgmt.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/00\_mgmt.cfg"

* Create a new folder called **templates** and create a folder named **eos** inside this templates folder.
* Create a new jinja2 file **mgmt.j2** inside the templates/eos directory as shown below

$ cat templates/junos/mgmt.j2  
  
!

hostname {{global.site|lower}}-{{inventory\_hostname}}

!

aaa authorization exec default local

!

{% for user in global.users%}

username {{user.name}} privilege {{user.privilege}} role {{user.role|default('network-admin')}} secret {{user.password}}

{% endfor%}

!

vrf definition {{global.mgmt\_vrf}}

!

interface Management1

vrf forwarding {{global.mgmt\_vrf}}

ip address {{ansible\_host}}/{{global.mgmt\_prefix}}

no lldp transmit

no lldp receive

!

* In the ***pb\_arista\_dc\_fabric.yml*** file , add the below highlighted tasks

$ cat pb\_arista\_dc\_fabric.ym  
  
< -- Output Omitted for brevity -->

- name: "P1T2: Build Config Directory Strcuture"

import\_tasks: "tasks/build\_config\_dir.yml"

delegate\_to: localhost

tags: config

- name: "P1T3: Build Devices configuration"

import\_tasks: "tasks/junos\_build\_config.yml"

delegate\_to: localhost

tags: config

**How it works..**

In order to configure the various system parameters on Arista devices like DNS, users and SNMP we will utilize a Jinja2 template to generate the required system configuration for each node defined in our ansible inventory.

We start with a task to build the configuration directory and create a sperate folder for each node under the configuration directory. All these tasks are grouped in the yaml file called build\_config\_dir.yml and we use the import\_task directive to import all these tasks in our main playbook.

We generate the basic system configuration for each device using the template task which is defined in the yaml file build\_config.yml which is again is imported to our main playbook using the import task directive. The template task uses the jinja2 template ***mgmt.j2*** and output the rendered system configuration for each device into a new text file 00\_mgmt.cfg.

All the variables in the jina2 template are retrieved from the variables declared and defined in the ***all.yml*** file which applies to all the devices in our ansible inventory. The choice of declaring all these variables in the all.yml file since all the system parameters and common across all the devices in our network.

Below is the management configuration fie for spine01 switch which is output after running the playbook.

!

hostname dc1-spine01

!

aaa authorization exec default local

!

username ansible privilege 15 role network-admin secret sha512 $6$mfU4Ei0AORd6rage$5YObhOI1g0wNBK5onaKDpYJhLZ9138maJKgcOznzFdpM25Tf3rb0PWSojUSMtRQY0Y7.cexCFj5aFLY17tuNU1

!

vrf definition MGMT

!

interface Management1

vrf forwarding MGMT

ip address 172.20.1.35/28

no lldp transmit

no lldp receive

!

At this stage we have generated the system configuration for all the arista switches in our inventory however we still didn’t push this configuration to the devices. In later recipes we will outline how to push the configuration to the arista devices.

**There is More..**

Ansible also provides declarative modules to configure various system level parameters on arista devices. The below sample configuration outline two modules which can be used to set the hostname and DNS on arista devices as well as to provision system users on the devices.

---

- name: "Conifgure Basic System config"

eos\_system:

hostname: " {{global.site|lower}}-{{inventory\_hostname}}"

name\_servers: "{{ global.dns }}"

state: present

- name: "Configure Users"

eos\_user:

name: "{{ item.username }}"

role: "{{ item.role | default("network-admin") }}"  
 privilege: "{{ item.privilege | default(15)}}"

configured\_password: "{{ item.password }}"

state: present

with\_items: "{{ global.users }}"

The downside of these modules are they only cover very specific parts in the system configuration parts, thus in order to have more control on the configuration of the arista device the jinja2 template approach is more advices.

**Configuring Interfaces on Arista Devices**

In this recipe we will outline how to configure different Interface parameters on Arista devices like interface description and IP address information.

**Getting Ready**

We are assuming the Network inventory is already in place and eAPI is already enabled on arista switches as per the previous recipes.

**How to do it..**

* Add the below contents to the **all.yml** file inside the group\_vars directory that descripe the Interfaces on our sample dc fabric network.

p2p\_ip:

leaf01:

- {port: Ethernet8, ip: 172.31.1.1 , peer: spine01, pport: Ethernet1, peer\_ip: 172.31.1.0}

- {port: Ethernet9, ip: 172.31.1.11 , peer: spine02, pport: Ethernet1, peer\_ip: 172.31.1.10}

leaf02:

< -- Output Omitted for bevirty -->

leaf03:

< -- Output Omitted for bevirty -->

leaf04:

< -- Output Omitted for bevirty -->

spine01:

< -- Output Omitted for bevirty -->

spine02:

< -- Output Omitted for bevirty -->

lo\_ip:

leaf01: 10.100.1.1/32

leaf02: 10.100.1.2/32

leaf03: 10.100.1.3/32

leaf04: 10.100.1.4/32

spine01: 10.100.1.254/32

spine02: 10.100.1.253/32

* Create a new jinja2 file **intf.j2** under the **templates/eos** directory with the below data

{% set node\_intfs = p2p\_ip[inventory\_hostname] %}

{% for p in node\_intfs| sort(attribute='port') %}

!

interface {{p.port}}

description "{{global.site}} | Rpeer: {{p.peer}} | Rport: {{p.pport}}"

no switchport

ip address {{p.ip}}/{{global.p2p\_prefix}}

{% endfor %}

!

!

interface Loopback0

ip address {{lo\_ip[inventory\_hostname]}}

!

* On the file **build\_config.yml** inside the **tasks** folder add the below highlighted task

$ cat tasks/build\_config.yml

< -- Output Omitted for bevirty -->

- name: "Interface Configuration"

template:

src: "{{ansible\_network\_os}}/intf.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/01\_intf.cfg"

**How it works..**

We define all the data for all the interfaces in our sample network topology under two main data structures in the **all.yml** file. We use a the **p2p\_ip** dictionary to model all the point to point IP addresses in our network and we use the **lo\_ip** dictionary to specify the loopback IP addresses for our nodes.

We use a jinja2 template to capture the Interface configuration for both the spine and leaf nodes and we use the data defined in the above two data structures to render these template into the required interface configuration for each device in our ansible inventory.

In our main playbook **pb\_arista\_dc\_fabric.yml** we are importing all the tasks inside the **tasks/build\_config.yml** file and this will generate all the configuration snippets for all the sections outlined in this file (management, interfaces , etc…).

Below is the generated interface configuration for spine01 device after running the playbook.

!

interface Ethernet1

description "DC1 | Rpeer: leaf01 | Rport: Ethernet8"

no switchport

ip address 172.31.1.0/31

!

interface Ethernet2

description "DC1 | Rpeer: leaf02 | Rport: Ethernet8"

no switchport

ip address 172.31.1.2/31

!

< -- Output Omitted for brevity -->  
!

interface Loopback0

ip address 10.100.1.254/32

!

**How it works..**

Ansible also provides declarative modules to configure interface parameters on arista devices as shown in the below section

---

- name: "Configure the Physical Interfaces"

eos\_interface:

name: "{{ item.port }}"

enabled: true

description: "{{global.site}} | Rpeer:{{item.peer}} | Rport:{{item.pport}}"

with\_items: "{{p2p\_ip[inventory\_hostname]}}"

- name: "Configure IP Addresses"

eos\_l3\_interface:

name: "{{ item.port }}"

ipv4: "{{ item.ip }}/{{ global.p2p\_prefix }}"

state: present

with\_items: "{{ p2p\_ip[inventory\_hostname] }}"

Using either of the two techniques (jinja2 or declarative approach) is possible however the jinja2 templates gives you more control regarding any configuration parameter which might not be addressed in the declarative modules.

**Configuring BGP on Arista Devices**

In this recipe, we will outline how to configure eBGP as the underlay routing protocol for our sample leaf/spine DC fabric.

**Getting Ready**

In this recipe we are assuming that the interface and IP address information is already configured as per the previous recipe.

**How to do it..**

* Create a new jinja2 file **underlay\_bgp.j2** under the **templates/eos** directory with the below data for the prefix-list template

{% set bgp\_grp = 'LEAF' if 'spine' in inventory\_hostname else 'SPINE' %}

!

route-map loopback permit 10

match ip address prefix-list loopback

!

{% if 'spine' in inventory\_hostname %}

!

ip prefix-list loopback

{% for node,ip in lo\_ip.items() | sort %}

{% if 'leaf' in node or inventory\_hostname in node %}

seq {{loop.index + 10 }} permit {{ip}}

{% endif %}

{% endfor %}

!

{% else %}

!

ip prefix-list loopback

seq 10 permit {{lo\_ip[inventory\_hostname]}}

!

{% endif %}

* Update the jinja2 file **underlay\_bgp.j2** under the **templates/eos** directory with the BGP template as shown below

!

router bgp {{bgp\_asn}}

router-id {{lo\_ip[inventory\_hostname].split('/')[0]}}

maximum-paths 2

bgp bestpath tie-break router-id

neighbor {{ bgp\_grp }} peer-group

neighbor {{ bgp\_grp }} description "Peer Group for All {{bgp\_grp}} Nodes"

neighbor {{ bgp\_grp }} graceful-restart-helper

neighbor {{ bgp\_grp }} send-community standard extended

neighbor {{ bgp\_grp }} maximum-routes 100000 warning-only

{% for p in bgp\_peers %}

neighbor {{ p.peer\_ip}} peer-group {{ bgp\_grp }}

neighbor {{ p.peer\_ip}} remote-as {{p.remote\_as}}

{% endfor %}

redistribute connected route-map loopback

!

address-family ipv4

neighbor {{ bgp\_grp }} activate

neighbor {{ bgp\_grp }} route-map loopback out

!

* In the file **build\_config.yml** inside the **tasks** folder add the below highlighted task

---

< -- Output Omitted for brevity -->

- name: "Underlay BGP Configuration"

template:

src: "{{ansible\_network\_os}}/underlay\_bgp.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/03\_bgp.cfg"

**How it works..**

We use the same interface data that was declared in the **p2p\_ip** data structure in **all.yml** file in order to provision the ospf configuration on the network devices in our sample network. We use a new jinja2 template to **ospf.j2** file under the templates/junos to capture the ospf configuration parameters (ospf cost, ospf interface type, etc..) that need to be implemented on the juniper devices.

Under the **tasks/juniper\_build\_config.yml** file we add a new task which uses the jinja2 template **ospf.j2** to render the jinja2 template and output the ospf configuration section for each device outlined in our ansible inventory.

The below snippet outlines the ospf configuration generated for mxpe01 device after running the playbook with the new task.

$ cat configs/mxpe01/02\_ospf.cfg

protocols {

ospf {

area 0 {

interface ge-0/0/0 {

interface-type p2p;

metric 100;

}

interface ge-0/0/1 {

interface-type p2p;

metric 100;

}

interface lo0.0 {

passive;

}

}

}

}

**Configuring BGP EVPN on Arista devices**

In this recipe, we will outline how to configure BGP EVPN as the control plane for VXLAN tunnels across our spine/leaf DC fabric in our sample topology using ansible.

**Getting Ready**

**How to do it..**

* In the file **junos\_build\_config.yml** inside the **tasks** folder add the below highlighted task

$ cat tasks/junos\_build\_config.yml  
---

- name: "System Configuration"

template:

src: "{{ansible\_network\_os}}/mgmt.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/00\_mgmt.cfg"

- name: "Interface Configuration"

template:

src: "{{ansible\_network\_os}}/intf.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/01\_intf.cfg"

- name: "OSPF Configuration"

template:

src: "{{ansible\_network\_os}}/ospf.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/02\_ospf.cfg"

**- name: "MPLS Configuration"**

**template:**

**src: "{{ansible\_network\_os}}/mpls.j2"**

**dest: "{{config\_dir}}/{{ inventory\_hostname }}/03\_mpls.cfg"**

* Create a new jinja2 file **mpls.j2** under the **templates/junos** directory with the below data

$ cat host\_vars/core01.yml  
protocols {

ldp {

{% for intf in p2p\_ip[inventory\_hostname]|sort(attribute='port') %}

interface {{intf.port}}.{{intf.vlan|default('0')}};

{% endfor %}

interface lo0.0;

}

rsvp {

{% for intf in p2p\_ip[inventory\_hostname]|sort(attribute='port') %}

interface {{intf.port}}.{{intf.vlan|default('0')}};

{% endfor %}

}

mpls {

{% for intf in p2p\_ip[inventory\_hostname]|sort(attribute='port') %}

interface {{intf.port}}.{{intf.vlan|default('0')}};

{% endfor %}

}

}

**How it works..**

We use the same methodology similar to how we configured the interfaces and OSPF by using a JINJA2 template to generate the needed MPLS configuration the juniper devices in our inventory and below is a sample of the MPLS configuration for mxpe02 router.

protocols {

ldp {

interface ge-0/0/0.0;

interface ge-0/0/1.0;

interface lo0.0;

}

rsvp {

interface ge-0/0/0.0;

interface ge-0/0/1.0;

}

mpls {

interface ge-0/0/0.0;

interface ge-0/0/1.0;

}

}

**Configuring BGP on Juniper Devices**

In this recipe we will outline how to configure BGP on Juniper devices.

**Getting Ready**

**How to do it..**

* In the file **junos\_build\_config.yml** inside the **tasks** folder add the below highlighted task

---

- name: "System Configuration"

template:

src: "{{ansible\_network\_os}}/mgmt.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/00\_mgmt.cfg"

- name: "Interface Configuration"

template:

src: "{{ansible\_network\_os}}/intf.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/01\_intf.cfg"

- name: "OSPF Configuration"

template:

src: "{{ansible\_network\_os}}/ospf.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/02\_ospf.cfg"

- name: "MPLS Configuration"

template:

src: "{{ansible\_network\_os}}/mpls.j2"

dest: "{{config\_dir}}/{{ inventory\_hostname }}/03\_mpls.cfg"

**- name: "BGP Configuration"**

**template:**

**src: "{{ansible\_network\_os}}/bgp.j2"**

**dest: "{{config\_dir}}/{{ inventory\_hostname }}/04\_bgp.cfg"**

* Create a new jinja2 file **mpls.j2** under the **templates/junos** directory with the below data

protocols {

{% if bgp\_peers is defined %}

bgp {

group Core {

type internal;

local-address {{ lo\_ip[inventory\_hostname] | ipaddr('address')}};

{% if bgp\_topo.rr == inventory\_hostname %}

cluster {{ lo\_ip[inventory\_hostname].split('/')[0] }};

{% endif %}

{% for af in bgp\_topo.af %}

{% if af == 'inet' %}

family inet {

unicast;

}

{% endif %}

{% if af == 'inet-vpn' %}

family inet-vpn {

unicast;

}

{% endif %}

{% if af == 'evpn' %}

family evpn {

signaling;

}

{% endif %}

{% endfor %}

{% for p in bgp\_peers %}

neighbor {{ p.peer}};

{% endfor %}

}

}

{% endif %}

}

* In the all.yml file under group\_vars add the below section

bgp\_topo:  
 rr: mxp01

af:

- inet

- inet-vpn

peers:

mxp01:

- mxpe01

- mxpe02

- xrpe03

mxpe01:

- mxp01

mxpe02:

- mxp01

xrpe03:

- mxp01

**How it works..**

Using a similar approach to all the previous recipes we use a JINJA2 template to generate the BGP configuration for the juniper devices. We use a new data structure in the **all.yml** file in order to describe our BGP logical topology and varaibles in the bgp.j2 template file call the data from this data structure in order to compile the exact configuration for each node.

Below is a sample of the BGP configuration for mxp01 Router which is the Route Reflector in our topology.

protocols {

bgp {

group Core {

type internal;

local-address 10.100.1.254;

cluster 10.100.1.254;

family inet {

unicast;

}

family inet-vpn {

unicast;

}

neighbor 10.100.1.1;

neighbor 10.100.1.2;

neighbor 10.100.1.3;

}

}

}

**Deploying Configuration on Juniper Devices**

In this recipe we will outline how to push configuration on juniper devices using ansible.

**Getting Ready**

This recipe requires NETCONF to be enabled on the juniper devices.

**How to do it..**

* In the ***pb\_junos\_net\_build*** file , add the below highlighted tasks

---

- name: " Play 1: Deploy Config on All JunOS Devices"

hosts: junos

tasks:

- name: "P1T1: Enable NETCONF"

junos\_netconf:

netconf\_port: 830

state: present

vars:

ansible\_connection: network\_cli

tags: netconf

- name: "P1T2: Build Config Directory Strcuture"

import\_tasks: "tasks/build\_config\_dir.yml"

delegate\_to: localhost

tags: config

- name: "P1T3: Build Devices configuration"

import\_tasks: "tasks/junos\_build\_config.yml"

delegate\_to: localhost

tags: config

**- name: "Remove Old Assembled Config"**

**file:**

**path: "{{config\_dir}}/{{ inventory\_hostname }}.cfg"**

**state: absent**

**delegate\_to: localhost**

**tags: config**

**- name: "Assemble config"**

**assemble:**

**src: "{{config\_dir}}/{{ inventory\_hostname }}"**

**dest: "{{config\_dir}}/{{ inventory\_hostname }}.cfg"**

**delegate\_to: localhost**

**tags: config**

**- name: "Deploy Configuration"**

**junos\_config:**

**src: "{{config\_dir}}/{{ inventory\_hostname }}.cfg"**

**tags: deploy**

**How it works..**

In the previous recipes we generated different sections of the configuration for juniper devices like interfaces, OSPF, MPLS and BGP. At this point we have these sections for each node in our inventory saved in a sperate folder per node. We need to push the configuration on the juniper devices however instead of pushing the interface configuration, then the OSPF configuration and so forth, we will use the assemble module to group all the configuration snippets and combine them into a single file which we will push to the juniper device.

We use the **assemble** module and provide it with the folder which has all the configuration snippets for each router and we also provide the name of the output file. Since the assemble module is not idempotent, we use the file module in the first task in order to clear any old file from previous run of the playbook. Thus, on each run of the playbook the assemble module will construct a fresh new file for each router in our inventory.

In the last task we use the **junos\_config** module to push the newly assemble file to the remote devices in our ansible inventory.

**There is More..**

The **junos\_config** module also supports the rollback feature supported by JunOS, thus we can build a small playbook to rollback the configuration in case needed as shown below

---

- name: " Play 1: Rollback Config on JunOS"

hosts: junos

tasks:

- name: "Rollback config"

junos\_config:

rollback: 1

In the above playbook we rollback to the last version of the configuration however by changing the number in the rollback attribute we can control to which version of the configuration we want to rollback.

**Configuring L3VPN on Juniper Devices**

In this recipe we will outline how to configure L3VPN on Juniper devices.

**Getting Ready**

NETCONF must be enabled on the Juniper devices so as to use the ansible modules in this recipe.

**How to do it..**

* Create a new file called **l3vpn.yml** with the below contents.

---  
l3vpns:

vpna:

state: present

rt: "target:{{bgp\_asn}}:10"

rd: "1:10"

sites:

- node: mxpe01

port: ge-0/0/3.10

ip: 172.10.1.1/24

- node: mxpe02

port: ge-0/0/3.10

ip: 172.10.2.1/24

vpnb:

state: present

rt: "target:{{bgp\_asn}}:20"

rd: "1:20"

sites:

- node: mxpe01

port: ge-0/0/3.20

ip: 172.20.1.1/24

- node: mxpe02

port: ge-0/0/3.20

ip: 172.20.2.1/24

* Create a new playbook called **pb\_junos\_l3vpn.yml** with the below contents.

---

- name: "Deploy L3VPNs on Juniper Devices"

hosts: pe

vars\_files:

- "l3vpn.yml"

tasks:

- name: "Set VPN Interfaces"

set\_fact:

l3vpn\_intfs: "{{ l3vpn\_intfs|default([]) +

l3vpns[item.key].sites |

selectattr('node','equalto',inventory\_hostname) | list}}"

with\_dict: "{{l3vpns}}"

delegate\_to: localhost

- name: "Configure Interfaces for L3VPN Sites"

junos\_config:

lines:

- set interfaces {{ item.port.split('.')[0]}} vlan-tagging

- set interfaces {{ item.port}} vlan-id {{ item.port.split('.')[1] }}

loop: "{{ l3vpn\_intfs }}"

- name: "Configure IP address for L3VPN Interfaces"

junos\_l3\_interface:

name: "{{ item.port.split('.')[0]}}"

ipv4: "{{ item.ip }}"

unit: "{{ item.port.split('.')[1] }}"

loop: "{{l3vpn\_intfs}}"

tags: intf\_ip

- name: "Configure L3VPNs"

junos\_vrf:

name: "{{ item.key }}"

rd: "{{item.value.rd}}"

target: "{{ item.value.rt }}"

interfaces: "{{ l3vpns[item.key].sites |

map(attribute='port') | list }}"

state: "{{ item.value.state }}"

with\_dict: "{{l3vpns}}"

when: inventory\_hostname in (l3vpns[item.key].sites | map(attribute='node') | list)

tags: l3vpn

**How it works..**

We create a new YAML file called **l3vpn.yml** which describes the l3vpn topology and data that we want to implement on all the Juniper devices on our topology. We include this file in the new playbook that we create to provision the l3vpns on our network devices.

In the playbook pb\_junos\_l3vpn.yml we use the data from the l3vpn.yml to grap the data required to provision the l3vpn, and this playbook is divided into multiple tasks which provision the following sections

* We use the **junos\_config** to configure all the interfaces which is part of the l3vpns to be ready to configure vlans on these interfaces.
* We use the **junos\_l3\_interface** module to apply the ip addresses on all these interfaces which is part of our l3vpn model.
* We use the **junos\_vrf** to configure the correct routing-instances on the nodes as per our l3vpn data model.

The below outline the l3vpn configuration that is applied on mxpe01 after running this playbook

ansible@mxpe01> show configuration routing-instances

vpna {

instance-type vrf;

interface ge-0/0/3.10;

route-distinguisher 1:10;

vrf-target target:65400:10;

vrf-table-label;

}

vpnb {

instance-type vrf;

interface ge-0/0/3.20;

route-distinguisher 1:20;

vrf-target target:65400:20;

vrf-table-label;

}

**Validate Network Reachability on Juniper Devices**

In this recipe, we will outline how to validate network reachability via ping using ansible on juniper devices

**Getting Ready**

This recipe assumes that that the network is already built as outlined in all the previous recipes and we will use ping to validate network reachability across all our nodes.

**How to do it..**

* Create a new playbook called **pb\_junos\_ping.yml** with the below contents

---

- name: "Validate Core Reachability"

hosts: junos

tasks:

- name: "Ping Across All Loopback Interfaces"

junos\_ping:

dest: "{{ item.value.split('/')[0] }}"

interface: lo0.0

size: 512

with\_dict: "{{lo\_ip}}"

vars:

ansible\_connection: network\_cli

**How it works..**

We use the **junos\_ping** module in order to ping from all the nodes in our network inventory to all the loopback interfaces defined in the **lo\_ip** data structure defined in the all.yml file. This module connects to each device and execute ping to all the destinations and validate that ping packets are reaching it intended destination. In case there is a network reachability problem, the module will fail.

**Retrieving Operational data from Juniper Devices**

In this recipe we will outline how to execute operational commands on IOS devices and store these output into text files for further processing.

**Getting Ready**

NETCONF must be enabled on the juniper devices in order to follow along with this recipe. Further

**How to do it..**

* Install jxmlease python package as shown below.

$ pip3 install jxmlease

* Create a new playbook called **pb\_get\_ospf\_peers.yml** and populate it as shown below

---

- name: "Get OSPF Status"

hosts: junos

tasks:

- name: "Get OSPF Neigbours Data"

junos\_command:

commands: show ospf neighbor

display: xml

register: ospf\_output

- name: "Extract OSPF Neigbour Data"

set\_fact:

ospf\_peers: "{{ ospf\_output.output[0]['rpc-reply']\

['ospf-neighbor-information']['ospf-neighbor'] }}"

- name: "Validate All OSPF Peers are in Full State"

assert:

that: item['ospf-neighbor-state'] == 'Full'

fail\_msg: "Peer on Interface {{item['interface-name']}} is Down"

success\_msg: "Peer on Interface {{item['interface-name']}} is UP"

loop: "{{ospf\_peers}}"

**How it works..**

One of the advantages of using NETCONF API to interact with Juniper devices is that we can get structured output for all the operational commands that we execute on the juniper devices. The output that device return to us over the NETCONF session is XML and Ansible uses a python library called jxmlease to decode this XML and transform it to JSON for better representation. That is why our first task was to install the jxmlease python package.

We use the junos\_command module to send operational commands to a juniper device and we specify that we need XML as the output format that get return from the node. This XML data structure is transformed to JSON using the jxmlease package by ansible. we save this data using register to a new variable called ospf\_output. Below is a sample of the JSON data that is returned from this command

"msg": [

{

"rpc-reply": {

"ospf-neighbor-information": {

"ospf-neighbor": [

{

"activity-timer": "34",

"interface-name": "ge-0/0/0.0",

"neighbor-address": "10.1.1.2",

"neighbor-id": "10.100.1.254",

"neighbor-priority": "128",

"ospf-neighbor-state": "Full"

},

{

"activity-timer": "37",

"interface-name": "ge-0/0/1.0",

"neighbor-address": "10.1.1.8",

"neighbor-id": "10.100.1.253",

"neighbor-priority": "128",

"ospf-neighbor-state": "Full"

}

]

}

}

}

]

All this data structure is contained in the **ospf\_output.output[0]** variable and we use set\_fact module to grap the ospg-neigbour data. After that we use the **Assert** module to loop through all the OSPF peers in this data structure and validate that the OSPF neighbour state is equal to Full. In case all the OSPF peers are in Full state, the task will succeed, however if the OSPF state is in any other state , the task will fail.

**How it works..**

In case we need to get the operational data from JunOS devices in CLI format for log collection we can use junos\_command module without the display xml option as shown below.

---

- name: "Play 1: Execute Operational Commands"

hosts: junos

vars:

log\_folder: "logs"

op\_folder: "op\_data"

op\_cmds:

- show configuration

tasks:

- name: "P1T1: Build Directories to Store Data"

block:

- name: "Create folder to store Device config"

file:

path: "{{ log\_folder }}"

state: directory

run\_once: yes

delegate\_to: localhost

- name: "P1T2: Get Running configs from Devices"

junos\_command:

commands: "{{ op\_cmds }}"

register: show\_run

- name: "P1T3: Save Running Config per Device"

copy:

content: "{{ show\_run.stdout[0] }}"

dest: "{{ log\_folder }}/{{ inventory\_hostname }}.cfg"