4. [Building Data Center Networks with Arista and Ansible](https://epic.packtpub.com/index.php?module=oss_Chapters&action=DetailView&record=db226a2e-c013-85d9-20ff-5cc2c0052772)

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.



# Technical Requirements

The Code for all the recipes in this chapter can be found in the below GitHub Repo

<https://github.com/PacktPublishing/Network-Automation-Cookbook/tree/master/ch4_arista>

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 Interfaces 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 Ansible 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 ***ch4\_arista***.

**How to do it..**

1. Inside the new folder (ch4\_arista) 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

1. 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..**

Defining an ansible inventory is a mandatory in order to describe and classify the network devices in our network that should be managed by Ansible. In the Ansible Inventory we also specify the IP addresses through which Ansible will communicate with these managed devices using the **ansible\_host** parameter.

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..**

1. Inside the folder ch4\_arista create the folder group\_vars.
2. 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

1. 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

!

1. 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..**

1. 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

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

$ cat group\_vars/all.yml

global:

mgmt\_vrf: MGMT

1. 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 Arista Devices**

In this recipe, we will outline how to configure some basic system options like hostname, DNS servers and provision users on Arista devices. We will understand how to setup all these system level parameters using the various Ansible modules available and we will outline the different ways to manage these parameters.

**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..**

1. 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

1. 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"

1. Create a new folder called **templates** and create a folder named **eos** inside this templates folder.
2. 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

!

1. 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 separate 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 the 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 outlines 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. We will outline how to use the various Ansible modules available to interact with the interfaces on arista devices and how to setup the interfaces on all the arista devices in our sample network topology.

**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..**

1. Add the below contents to the **all.yml** file inside the group\_vars directory that describe 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 brevity -->

leaf03:

< -- Output Omitted for brevity -->

leaf04:

< -- Output Omitted for brevity -->

spine01:

< -- Output Omitted for brevity -->

spine02:

< -- Output Omitted for brevity -->

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

1. 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]}}

!

1. 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

!

**There is More..**

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. We are going to build the eBGP peering setup using the Point to Point IP address between the leaf switches and spine switches and the BGP ASN assignment is as shown in the below table.

|  |  |
| --- | --- |
| **Node** | **BGP ASN** |
| Spine01 | 65100 |
| Spine02 | 65100 |
| Leaf01 | 65001 |
| Leaf02 | 65002 |
| Leaf03 | 65003 |
| Leaf04 | 65004 |

**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..**

1. Create a host\_vars directory and create a folder for *each device* in our inventory. Under each folder create a new YAML file **underlay\_bgp.yml** with the below contents. Below is an example for ***leaf01*** device in our inventory

## Leaf01 BGP Data ###

bgp\_asn: 65001

bgp\_peers:

- peer: spine01

peer\_ip: 172.31.1.0

remote\_as: 65100

- peer: spine02

peer\_ip: 172.31.1.10

remote\_as: 65100

1. 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 %}

1. 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

!

1. 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..**

Since we are going to run eBGP between the leaf and spine nodes and each leaf switch in our topology will have his own BGP ASN, we include all this data on a per host basis using the **host\_vars** folder and we create a folder for each node to include all the data for this host. We create a YAML file to hold the BGP information for each device, thus in case we need to add more host specific data for another protocol we can easily add a new file. The file layout for the host\_vars folder will be as shown below

$ tree host\_vars  
host\_vars

├── leaf01

│   └── underlay\_bgp.yml

├── leaf02

│   └── underlay\_bgp.yml

├── leaf03

│   └── underlay\_bgp.yml

├── leaf04

│   └── underlay\_bgp.yml

├── spine01

│   └── underlay\_bgp.yml

└── spine02

└── underlay\_bgp.yml

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

For each device we generate a prefix-list for the BGP advertisement that it will advertise to its peers as per the below criteria

1. For Spine switches we advertise all the leaf loopback IP addresses along with the Spine loopback Interface.
2. For the leaf switches we advertise only the loopback IP address.

The below snippet outlines the BGP configuration generated for leaf01 device after running the playbook with the new task.

$ cat configs/leaf01/04\_bgp.cfg  
  
!

route-map loopback permit 10

match ip address prefix-list loopback

!

ip prefix-list loopback

seq 10 permit 10.100.1.1/32

!

router bgp 65001

router-id 10.100.1.1

maximum-paths 2

bgp bestpath tie-break router-id

neighbor SPINE peer-group

neighbor SPINE description "Peer Group for All SPINE Nodes"

neighbor SPINE graceful-restart-helper

neighbor SPINE send-community standard extended

neighbor SPINE maximum-routes 100000 warning-only

neighbor 172.31.1.0 peer-group SPINE

neighbor 172.31.1.0 remote-as 65100

neighbor 172.31.1.10 peer-group SPINE

neighbor 172.31.1.10 remote-as 65100

redistribute connected route-map loopback

!

address-family ipv4

neighbor SPINE activate

neighbor SPINE route-map loopback out

!

**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**

This recipe assumes the Point to Point IP addresses and Loopback Interfaces have been configured as per previous recipes also the Underlay BGP configuration is already generated as per the previous policy.

**How to do it..**

1. Create a new jinja2 file **overlay\_bgp.j2** under the **templates/eos** directory with the below data

$ cat templates/eos/overlay\_bgp.yml  
  
{% set bgp\_evpn\_grp = 'LEAF\_EVPN' if 'spine' in inventory\_hostname else 'SPINE\_EVPN' %}

{% set bgp\_ipv4\_grp = 'LEAF' if 'spine' in inventory\_hostname else 'SPINE' %}

service routing protocols model multi-agent

!

router bgp {{bgp\_asn}}

neighbor {{ bgp\_evpn\_grp }} peer-group

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

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

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

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

neighbor {{ bgp\_evpn\_grp }} ebgp-multihop 2

neighbor {{ bgp\_evpn\_grp }} update-source Loopback0

{% for p in bgp\_peers %}

neighbor {{ lo\_ip[p.peer].split('/')[0]}} peer-group {{ bgp\_evpn\_grp }}

neighbor {{ lo\_ip[p.peer].split('/')[0]}} remote-as {{p.remote\_as}}

{% endfor %}

!

address-family evpn

neighbor {{ bgp\_evpn\_grp }} activate

!

address-family ipv4

no neighbor {{ bgp\_evpn\_grp }} activate

!

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

$ cat tasks/build\_config.yml  
  
< -- Output Omitted for brevity -->

- name: "Overlay BGP EVPN Configuration"

template:

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

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

**How it works..**

We use the same methodology similar to how we configured the Overlay eBGP by utilizing a JINJA2 template to generate the needed BGP EVPN configuration for the arista devices in our inventory and below is a sample of the BGP EVPN configuration for leaf01 switch.

service routing protocols model multi-agent

!

router bgp 65001

neighbor SPINE\_EVPN peer-group

neighbor SPINE\_EVPN description "Peer Group for All SPINE\_EVPN EVPN Nodes"

neighbor SPINE\_EVPN graceful-restart-helper

neighbor SPINE\_EVPN send-community extended

neighbor SPINE\_EVPN maximum-routes 100000 warning-only

neighbor SPINE\_EVPN ebgp-multihop 2

neighbor SPINE\_EVPN update-source Loopback0

neighbor 10.100.1.254 peer-group SPINE\_EVPN

neighbor 10.100.1.254 remote-as 65100

neighbor 10.100.1.253 peer-group SPINE\_EVPN

neighbor 10.100.1.253 remote-as 65100

!

address-family evpn

neighbor SPINE\_EVPN activate

!

address-family ipv4

no neighbor SPINE\_EVPN activate

!

**Configuring VXLANs and VLANs on Arista Devices**

In this recipe we will outline how to configure VLANs on arista Switches as well as build the VXLAN tunnels across the Spine/Leaf fabric in order to transport these VLANs. The VLANs that we will build is as shown in the below table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Node** | **Interface** | **Interface Type** | **VLANs** |
| Leaf01 | Ethernet1 | Access | 10 |
| Leaf02 | Ethernet1 | Access | 20 |
| Leaf03 | Ethernet1 | Access | 10 |
| Leaf03 | Ethernet2 | Access | 20 |
| Leaf04 | Ethernet1 | Access | 10 |
| Leaf04 | Ethernet2 | Access | 20 |

**Getting Ready**

This recipe is assuming that underlay and overlay BGP configuration is already generated as per the previous recipes.

**How to do it..**

1. Create a new YAML file **vxlan.yml** that capture the VLANs membership data and populate it as shown below

access\_interfaces:

leaf01:

- name: Ethernet1

mode: access

vlan: 10

leaf02:

- name: Ethernet1

mode: access

vlan: 20

leaf03:

- name: Ethernet1

mode: access

vlan: 10

- name: Ethernet2

mode: access

vlan: 20

leaf04:  
< -- Output Omitted for brevity -->

1. Create a new jinja2 file called **vxlan.j2** under the **templates/eos** directory with the below data and add the below section for the VLAN and interface config template

{% set access\_data = access\_interfaces[inventory\_hostname] %}

{% set all\_vlans = access\_data | map(attribute='vlan') | list %}

{% for vlan in all\_vlans %}

!

vlan {{ vlan }}

name VLAN\_{{ vlan }}

{% endfor %}  
!

{% for port in access\_data|sort(attribute='name') %}

interface {{ port.name }}

description "Access"

{% if port.mode == 'access' %}

switchport mode access

switchport access vlan {{port.vlan}}

{% else %}

switchport mode trunk

switchport trunk allowed vlan {{port.vlan}}

{% endif %}

{% endfor %}

!

1. Update the jinja2 file **vxlan.j2** under the **templates/eos** with the VXLAN section as show below

!

interface Vxlan1

vxlan source-interface Loopback0

vxlan udp-port 4789

{% for vlan in all\_vlans %}

vxlan vlan {{ vlan }} vni 10{{vlan}}

{% endfor %}

!  
!

router bgp {{bgp\_asn}}

!

{% for vlan in all\_vlans %}

vlan {{ vlan }}

rd {{lo\_ip[inventory\_hostname].split('/')[0]}}:10{{vlan}}

route-target both 10{{vlan}}:10{{vlan}}

redistribute learned

{% endfor %}

!

!

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

- name: "VLAN/VXLAN Configuration"

template:

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

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

when: inventory\_hostname in access\_interfaces.keys()

1. Include the **vxlan.yml** file in the playbook **pb\_arista\_dc\_fabric.yml** file so as to render the jinja2 templates correctly.

---

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

hosts: arista

**vars\_files: vxlan.yml**

tasks:  
< -- Output Omitted for brevity -->

**How it works..**

We capture all the Access VLAN information in a YAML file (vxlan.yml) and we generate the required VLAN configuration using JINJA2 templates as done in the previous receipes. We generate the VLAN to VXLAN mapping as well as the BGP EVPN RT configuration for each VLAN/VNI domain using the following logic

1. VNI = 10 + VLAN
2. RT = 10 + VLAN

The below configuration is the VLAN and VXLAN configuration snippet for leaf01 as an example

!

vlan 10

name VLAN\_10

!

!

interface Vxlan1

vxlan source-interface Loopback0

vxlan udp-port 4789

vxlan vlan 10 vni 1010

!

!

interface Ethernet1

description "Access"

switchport mode access

switchport access vlan 10

!

!

router bgp 65001

!

vlan 10

rd 10.100.1.1:1010

route-target both 1010:1010

redistribute learned

!

!

**Deploying Configuration on Arista Devices**

In this recipe we will outline how to push the configuration to the arista devices. We will use the configuration that we have generated in the previous recipes in order to provision the required configuration on our network devices in our topology. We will understand how to interact with the arista configuration using the suitable Ansible module in order to correctly provision the devices as per intended network design.

**Getting Ready**

This recipe requires eAPI to be enabled on the arista devices.

**How to do it..**

1. In the ***pb\_arista\_dc\_fabric.yml*** file, add the below tasks to generate the final config for the devices

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

hosts: arista  
 **vars\_files: vxlan.yml**

tasks:

< -- Output Omitted for brevity -->

**- 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**

1. In the ***pb\_arista\_dc\_fabric.yml*** file, add the below task to deploy the configuration to the arista switches

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

hosts: arista  
 **vars\_files: vxlan.yml**

tasks:

< -- Output Omitted for brevity -->

**- name: "Deploy Configuration"**

**eos\_config:**

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

**replace: config**

**tags: deploy**

**How it works..**

In the previous recipes we generated different sections of the configuration for the arista switches like interfaces, Underlay/Overlay BGP and VLAN/VXLAN. 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 to the arista devices however instead of pushing the configuration for each section on its own, 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 arista 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 **eos\_config** module to push the newly assemble file to the remote devices in our ansible inventory. We use the **replace** directive with the **config** option in order to replace all the configuration on the target device with the new configuration that we specify in the **src** option. Thus the configuration on the devices is completely managed and controlled by Ansible. This also means that if there is any configuration that was implemented outside our ansible playbook , this configuration will be deleted once we run the playbook and push the new configuration to the devices.

**Retrieving Operational data from Arista Devices**

In this recipe we will outline how to execute operational commands on Arista devices and use the output to validate the state of the devices.

**Getting Ready**

eAPI must be enabled on the arista devices in order to follow along with this recipe.

**How to do it..**

1. Create a new playbook called **pb\_get\_vlans.yml** and populate it as shown below to run “show vlan” on all leaf switches and store the output

---

- name: " Play 1: Retrieve All VLANs from Arista Switches"

hosts: leaf

vars\_files: vxlan.yml

tasks:

- name: "Get All VLANs"

eos\_command:

commands: show vlan | json

register: show\_vlan

1. Update the playbook **pb\_get\_vlans.yml** and populate it with the below task to compare and validate that correct VLANs are configured on the devices

- name: "Validate VLANs are Present"

assert:

that: item.vlan in show\_vlan.stdout[0].vlans.keys()

fail\_msg: "VLAN:{{ item.vlan }} is NOT configured "

success\_msg: "VLAN:{{ item.vlan }} is configured "

loop: "{{ access\_interfaces[inventory\_hostname] }}"

delegate\_to: localhost

**How it works..**

We execute operational commands on the arista switches using the **eos\_command** ansible module and in order to return structured output we use the **json** keywork in the command in order to return the JSON output of the operational command (if supported). In this example we use send the show vlan command to get the list of vlans configured on the devices and we collect the output in the variable show\_vlan. The below snippet outline the output we get from the devices which is stored in this variable

ok: [leaf01] => {

"show\_vlan": {

< -- Output Omitted for brevity -->

"stdout": [

{

"vlans": {

"1": {

"dynamic": false,

"interfaces": {

< -- Output Omitted for brevity -->

},

"name": "default",

"status": "active"

},

"10": {

"dynamic": false,

"interfaces": {

"Ethernet1": {

"privatePromoted": false

},

"Vxlan1": {

"privatePromoted": false

}

},

"name": "VLAN\_10",

"status": "active"

}

}

}

]

We use the **assert** module to validate that the VLANs that we specified as input to our playbooks in the **vxlan.yml** file are all configured and operational on the switches by comparing this data with the output that we retrieved from the devices using the eos\_command which is stored in the variable show\_vlan.