# Chapter 4 (JunOS SP)

**Chapter Introduction**

In this chapter, we will outline how to automate Juniper devices based on JunOS in a typical service provider environment. We will explore how to interact with Juniper devices using Ansible and how to provision different services and protocols on JunOS devices using various Ansible modules. We will base our illustration based on the below sample network diagram of a basic SP network.

Below are the Software releases that this chapter is based on

* Ansible Machine Running Ubuntu 16.04
* Ansible 2.8
* Juniper vMX running JunOS 18.1R1 Release

The main recipes covered in this chapter is shown below

* Building Ansible Network Inventory.
* Connecting and Authentication to JunOS Devices.
* Enabling NETCONF on JunOS Devices.
* Configuring Generic System options on JunOS Devices .
* Configuring Physical Interfaces on JunOS Devices.
* Configuring IP addresses on JunOS Devices.
* Configuring OSPF on JunOS Devices.
* Configuring LDP and RSVP on JunOS Devices.
* Configuring BGP on JunOS Devices.
* Configuring L3VPN Service on JunOS Devices.
* Configuring L2VPN Service on JunOS Devices.
* Validating Network reachability on JunOS devices.
* Retrieving Operational Data from JunOS Devices
* Retrieving JunOS Device facts.

**Building Network Inventory**

In this chapter, we will outline how to build and structure the Ansible Inventory to describe the sample SP network setup outlined above.

**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\_junos\_netops***.

**How to do it..**

* Inside the new folder (ch4\_junos\_netops) we create ***hosts*** file with the below content

$ cat hosts  
[PE]

mxpe01 ansible\_host=172.20.1.3

mxpe02 ansible\_host=172.20.1.4

xrpe03 ansible\_host=172.20.1.5

[P]

mxp01 ansible\_host=172.20.1.2

[core:children]

pe

p

[all:vars]

ansible\_user=ansible

ansible\_ssh\_pass=ansible123

* 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 ***PE*** group which reference all the MPLS PE nodes in our topology.
* We created the ***P*** group which group reference all the MPLS P nodes in our topology.
* We created the ***core*** group which group reference both the PE and P 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 to Juniper Devices**

In this recipe, we will outline how to connect to Juniper Devices from Ansible via SSH in order to start managing the devices from Ansible. We are going to use SSH keys to authenticate the Ansible machine to the JunOS devices.

**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 configured.

**How to do it..**

* On the ansible machine create the private and public ssh keys as shown below

vagrant@ ubuntu-xenial$ ssh-keygen -t rsa -b 2048

Generating public/private rsa key pair.

Enter file in which to save the key (/home/vagrant/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/vagrant/.ssh/id\_rsa.

Your public key has been saved in /home/vagrant/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:OjgvAfIxkJb6OyHbOV2BMjjgSWud8EfMLpAZz5BT19c vagrant@ubuntu-xenial

* We Capture the public key that was created in the previous step

vagrant@ubuntu-xenial$ cat ~/.ssh/id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC2LErP6viusNDONeTYZ/OuG3YNM8aL77JiitcPy5isyvHRaVLf/KSKkxsSSniUaUdtaqJ2ZZwU57jIawI+AC/w7hrOdIaMkj6q6tUT3XNGZ6rW/KKb/lCd3UR+IidFpLmGQ+p+oNfGh4fkVzgW1x0gD75W/ykZKw5sSO9CE10wMiFFj+20p2LIPOJdAMIWZFhwxreusdFrcTai0gZrM55cw14hRGyNH8SiVzJiI7DVB+8y3ZSFo8xMjdIvhAn7IFQT9KrMCSOct4pr4x7gSvg0AR8HbIZmNRMp/fKrWGl2sKyo/8JKuBfqhCc6rc4qZeo2ZnzJZyYJtqzM4Kr+7G2F vagrant@ubuntu-xenial

* On the Juniper Devices we add a new User and designate that we will use ssh keys and we copy the ssh keys that was created on the ansible machine to the device as shown below

[edit system login]

ansible@mxpe01# show

user vagrant {

uid 2001;

class super-user;

authentication {

ssh-rsa "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCklobVn0yI1YEonSPdB3oUWBhi3ndvIFVVzWN8bO26xPFOxAuRSN4+vXUF/7l8WdDXxSlQ8eW9mrudpOjQF3LC1IchVmM6ODT17Zn39qP7CZqk9NHv3kZGk4yUuKc42pwTXor7tTg0kGEEWWwZrzlaIEAIUArLFKa7+yN1DyB8ZO53gUHlh6bkrskW2JpJ3J+StcQzh64nrOFfv9VEGlXTACo82CusyzZXhtrWI7hioIZgCOnUt7oS0q/7980hjBc+WqTlyGlj/CQF/KI9owHgCGbEKXwJjrccgE751kj5cMwW/ytSI1Cu/NEsQ1+Er/IinkhHB58bepAlhsXkLBLp vagrant@ubuntu-xenial"; ## SECRET-DATA

}

}

**How it works..**

We start by creating the public and private SSH keys on the ansible control machine using the ssh-keygen commands for the current user (vagrant user) in this case . We then copy the contents of the public key stored in the ~/.ssh/id\_rsa.pub file so as to use it in configuring the user on the Juniper devices.

On the JunOS devices we create a user with the same name as the user on the ansible control machine (vagrant) and we specify that we will use ssh-rsa keys for authentication. We past the contents of the public key that we copied from the b=ansible control machine and we include in the ssh-rsa configuration parameter and enclose it using “”.

With this configuration in place we are able to login to the network devices with no password from the ansible control machine.

**Enable NETCONF on JunOS Devices**

In this recipe, we will outline how to enable NETCONF protocol on JunOS Devices. This task is critical since we will use the NETCONF API in all the future recipe to control the JunOS Device. The NETCONF API provides several advantages compared with the traditional SSH access method and that is why we will use it in all our interaction with the JunOS Devices.

**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\_deploy\_net.yml as show below

$ cat pb\_deploy\_net.yml

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

hosts: junos

tasks:

- name: "P1T1: Enable NETCONF"

junos\_netconf:

netconf\_port: 830

state: present

* Create a new directory called group\_vars and create junos.yml file as shown below

$ mkdikr group\_vars && touch group\_vars/junos.yml

$ cat group\_vars/junos.yml

ansible\_network\_os: junos

ansible\_connection: network\_cli

**How it works..**

In order to start interacting with the JunOS Devices via NETCONF we need to enable it first, thus we need to SSH into the device initially and enable NETCONF. That is why in this receipe we are using the network\_cli ansible connection in order to connect with the JunOS Devices via traditional SSH and we need to set the ansible\_network\_os as junos. All these setting are stored in the junos.yml file under the group\_vars directory so as ansible apply this setting on all JunOS Devices in the inventory.

We create a new Playbook called pb\_deploy\_net.yml and in the first task we use the junos\_netconf module to enable the NETCONF protocol on the remote JunOS Devices. We state the NETCONF port that will be used (by default it is 830) and we outline that this configuration must be present on the remote devices via state: present directive.

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

vagrant@mxpe01# show system services

ssh;

netconf {

ssh {

port 830;

}

}

**There is more..**

After enabling the NETCONF on all the Juniper devices we need to start using this new API access method instead of the legacy SSH method. We change the **ansible\_connection** attribute in **group\_vars/junos.yml** as shown below

$ cat group\_vars/junos.yml

ansible\_network\_os: junos

**ansible\_connection: netconf**

We specify **netconf** as the new connection type to communicate with the Juniper devices and this is what we will use in all the subsequent recipes.

**Configuring Interfaces on IOS Devices**

In this recipe, we will outline how to configure the basic interface properties on Cisco IOS-based devices like setting the interface description, Interface MTU and enabling the interfaces. We will configure all the links within our topology as having Link MTU of 1500 and to be full duplex.

**Getting Ready**

To follow along with this recipe, an ansible inventory is assumed to be already setup and we will continue to build on the previous recipe to configure the interfaces as per our intended setup outlined in the start of this chapter.

**How to do it..**

* In the ***network.yml*** file (under group\_vars folder)  and add the following content

$ cat group\_vars/network.yml  
<-----Snippet ------->  
intf\_duplex: full  
intf\_mtu: 1500

* Create a new file called ***lan.yml*** under the group\_vars folder with the following data

$ cat group\_vars/lan.yaml  
  
interfaces:  
 core01:  
 - name: Ethernet0/1  
 description: access01\_e0/1  
 mode: trunk  
 - name: Ethernet0/2  
 description: access02\_e0/1  
 mode: trunk  
 - name: Ethernet0/3  
 description: core01\_e0/3  
 mode: trunk  
<-------- Snippet ------------>  
 access01:  
 - name: Ethernet0/1  
 description: core01\_e0/1  
 mode: trunk  
 - name: Ethernet0/2  
 description: core02\_e0/1  
 mode: trunk  
 - name: Ethernet0/3  
 description: Data\_vlan  
 mode: access  
 vlan: 10  
<----------- Snippet ------------->

* Update the ***pb\_build\_network.yml*** file with the following tasks

[223] → cat pb\_build\_network.yml  
---  
- name: "PLAY 1: Configure All Lan Switches"  
 hosts: lan  
 tags: lan  
 tasks:  
<------------Snippet --------------------->  
 - name: "P1T3: Configure Interfaces"  
 ios\_interface:  
 name: "{{ item.name }}"  
 description: "{{ item.description }}"  
 duplex: "{{ intf\_duplex }}"  
 mtu: "{{ intf\_mtu }}"  
 state: up  
 loop: "{{ interfaces[inventory\_hostname] }}"  
 register: ios\_intf

**How it works..**

**There is More..**

**Configuring L2 VLANs on IOS Devices**

In this recipe we will outline how to configure L2 VLANs on Cisco IOS devices as per the network topology discussed in the intro in this chapter.

**Getting Ready**

We will be building on the pervious receipes discussed in this chapter to continue to configrie the L2 VLANs on all the Lan devices within our sample topololgy.

**How to do it..**

* On ***the lan.yml*** file (under group\_vars folder) add the following

$ cat group\_vars/lan.yaml

vlans:

- name: Data

vlan\_id: 10

- name: Voice

vlan\_id: 20

- name: Web

vlan\_id: 100

* Update the ***pb\_build.yml*** playbook with the following task

---

- name: "PLAY 1: Configure All Lan Devices"

hosts: lan

tags: lan

connection: network\_cli

tasks:

🡨------- Snippet ------🡪

- name: "P1T4: Create L2 VLANs"

ios\_vlan:

vlan\_id: "{{ item.vlan\_id }}"

name: "{{ item.name }}"

loop: "{{ vlans }}"

tags: vlan

**How it works..**

On the lan.yml file we define a vlans list data structure which holds all the VLANs we will need to configure on all our core and access switches. This variable will be available for all the core and access switches and ansible will use this variable in order to provision the required VLANs on the remote devices.

We use another declarative module called ***ios\_vlan*** which takes the vlan definition (its name and the vlan-id) and configure these VLANs on the remote managed device. Its pulls the existing configuration from the device and compare it with the list of devices that need to be present and only push the delta.

We use the loop construct to go through all the items in the vlans and configure all the respective vlans on all the devices.

After running this task on the devices below is the output from one of the access switches

access01#sh vlan

VLAN Name Status Ports

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

1 default active Et1/0, Et1/1, Et1/2, Et1/3

10 Data active Et0/3

20 Voice active

100 Web active

**Configuring Trunk and Access Interfaces**

In this recipe, we will show how to configure access and trunk interfaces on Cisco IOS-based devices and how to map interfaces to access vlan as well as how to allow specific vlans on the trunks.

Following our sample topology we will configure the interfaces on the devices as shown in this table we are only showing the VLANs for access01 and core01 the other devices are the exact replica.

|  |  |  |  |
| --- | --- | --- | --- |
| **Device** | **Interface** | **Mode** | **Vlans** |
| Core01 | Ethernet0/1 | Trunk | 10,20,100 |
| Core01 | Ethernet0/2 | Trunk | 10,20,100 |
| Core01 | Ethernet0/3 | Trunk | 10,20,100,200 |
| Access01 | Etherent0/1 | Trunk | 10,20,100 |
| Access01 | Ethenet0/2 | Trunk | 10,20,100 |
| Access01 | Ethernet0/3 | Access | 10 |

**Getting Ready**

**How to do it..**

* On the lan.yml file under group\_vars folder add the following information

interfaces:

core01:

- name: Ethernet0/1

description: access01\_e0/1

mode: trunk

- name: Ethernet0/2

description: access02\_e0/1

mode: trunk

- name: Ethernet0/3

description: core01\_e0/3

mode: trunk

access01:

- name: Ethernet0/1

description: core01\_e0/1

mode: trunk

- name: Ethernet0/2

description: core02\_e0/1

mode: trunk

- name: Ethernet0/3

description: Data\_vlan

mode: access

vlan: 10

* Create a new ***core.yml*** file under group\_vars and include the following in it

core\_vlans:

- name: l3\_core\_vlan

vlan\_id: 200

interface: Ethernet0/3

* Update the pb\_build\_network.yml playbook with the following

---

- name: "PLAY 1: Configure All Lan Devices"

hosts: lan

tags: lan

tasks:

< ------- Snippet ----- >

- name: "P1T5: Configure L2 Trunks"

ios\_l2\_interface:

name: "{{ item.name }}"

mode: "{{ item.mode }}"

trunk\_allowed\_vlans: "{{ vlans | map(attribute='vlan\_id') | join(',') }}"

state: present

loop: "{{ interfaces[inventory\_hostname] | selectattr('mode','equalto','trunk') | list }}"

- name: "P1T6: Enable dot1q Trunks"

ios\_config:

lines:

- switchport trunk encapsulation dot1q

parents: interface {{item.name}}

loop: "{{ interfaces[inventory\_hostname] | selectattr('mode','equalto','trunk') | list }}"

tags: dot1q

- name: "P1T7: Configure Access Ports"

ios\_l2\_interface:

name: "{{ item.name }}"

mode: "{{ item.mode}}"

access\_vlan: "{{ item.vlan }}"

state: present

loop: "{{ interfaces[inventory\_hostname] | selectattr('mode','equalto','access') | list }}"

**How it works..**

We are using the same data structure in the lan.yml file the defines all the interfaces within the LAN network and describe their type (Access/trunk) and in case of Access ports we define which access interface is part of which vlan. We will reference this list data structure to configure the Access and Trunk ports on all the Devices within the lan group.

The Interfaces within the LAN are one of two options

* **Access**
  + We use the ***ios\_l2\_interface*** with *access\_vlan* parameter to specify configure the correct access vlan on the interface.
  + We select only the access interfaces per each device using the ***selectattr*** jinja2 filter and we match only on interface with mode access and we loop over this list for each device.
* **Trunk**
  + We use again the ***ios\_l2\_inteface*** with the parameter *trunk\_allowed\_vlans* to add all the VLANs on the trunk ports on both access and Core switches.
  + We select only the trunks ports using again the sellectattr jinja2 filter from the interfaces data structure per node.
  + We need to configure these trunks as dot1q ports however this attribute is still not enabled on the ios\_l2\_interface. Thus, we use another module ***ios\_config*** to send the required Cisco IOS command.

**There is More..**

**Configuring Interface IP addresses**

In this recipe we will explore how to configure the IP address on the Cisco IOS Devices. We will use the sample topology to configure the VLAN interfaces on both the Core switches as well we will outline how to configure VRRP between the Core Switches for all the VLAN interfaces. We will configure the following IP addresses as per the below table.

|  |  |  |
| --- | --- | --- |
| **Interface** | **Prefix** | **VRRP IP Address** |
| VLAN10 | 10.1.10.0/24 | 10.1.10.254 |
| VLAN20 | 10.1.20.0/24 | 10.1.20.254 |
| VLAN100 | 10.1.100.0/24 | 10.1.100.254 |

**Getting Ready**

This recipe assumes that the Interface and VLANs are configured as per the previous recipes in this chapter.

**How to do it..**

* Add in the core.yml file (under the group\_vars folder) the following

$ cat group\_vars/core.yml

< ---- Snippet ------ >

svi\_interfaces:

- name: Vlan10

ipv4: 10.1.10.0/24

vrrp: yes

ospf: passive

- name: Vlan20

ipv4: 10.1.20.0/24

vrrp: yes

ospf: passive

- name: Vlan100

ipv4: 10.1.100.0/24

vrrp: yes

ospf: passive

* Create ***core01.yml*** and ***core02.yml*** files under host\_vars folder and add the following

$ cat host\_vars/core01.yml

hst\_svi\_id: 1

hst\_vrrp\_priority: 100  
  
$ cat host\_vars/core02.yml

hst\_svi\_id: 2

hst\_vrrp\_priority: 50

* Update the pb\_build\_network.yml playbook with this section

- name: "PLAY 2: Configure Core Switches"

hosts: core

tags: l3\_core

tasks:

< ----- Snippet ------ >

- name: "P2T1: Create L3 VLAN Interfaces"

ios\_l3\_interface:

name: "{{item.name }}"

ipv4: "{{item.ipv4 | ipv4(hst\_svi\_id)}}"

loop: "{{svi\_interfaces}}"

tags: l3\_svi  
  
 - name: "P2T2: Enable the VLAN Interfaces"

ios\_interface:

name: "{{ item.name }}"

state: up

loop: "{{ svi\_interfaces }}"

- name: "P2T3: Create VRRP Configs"

ios\_config:

parents: interface {{ item.name }}

lines:

- vrrp {{item.name.split('Vlan')[1]}} priority {{ hst\_vrrp\_priority }}

- vrrp {{item.name.split('Vlan')[1]}} ip {{item.ipv4 | ipv4(254)|ipaddr('address')}}

loop: "{{svi\_interfaces | selectattr('vrrp','equalto',true) | list }}"

**How it works..**

In this section we are configuring the IP addressed for the L3 VLAN interfaces on the Core switches as well as configuring VRRP on all the L3 VLAN interfaces for providing L3 redundancy.

We are using a new list data structure called **svi\_interfaces** which describe the Interface with L3 IP addresses and also some added info regarding VRRP on these interface as well as OSPF which we will use in our next recipe. We also setup two new variable on each core routers called **hst\_svi\_id** and **hst\_vrrp\_priority** which we will use in the playbook to control the IP address on each core switch as well as the VRPP priority.

We use the ios\_l3\_interface ansible module to set the ipv4 addresses on the VLAN interfaces on each core switch we loop over the svi\_interfaces data structure and for each VLAN we configure the ipv4 address on the corresponding VLAN interface. We determine which which IP address is configured on each router using the hst\_svi\_id and the ipv4 address filter [{{item.ipv4 | ipv4(hst\_svi\_id)}}] . so for example for VLAN 10 we will assign 10.1.10.1/24 on core01 and 10.1.10.2/24 for core02.

When first creating the VLAN interface on Cisco IOS devices, it is in shutdown state so we need to enable them so we use the ios\_interface module to enable the interfaces.

For the VRRP part we return to use the **ios\_config** module to setup the VRRP config on all the VLAN interfaces and we use the hst\_vrrp\_priority to correctly setup core01 as the Master VRRP for all the VLANs.

Below is a sample of the config that is pushed on the devices after running the playbook

Core01  
========

!

interface Vlan10

ip address 10.1.10.1 255.255.255.0

vrrp 10 ip 10.1.10.254

!  
  
Core02  
=======  
!

interface Vlan10

ip address 10.1.10.2 255.255.255.0

vrrp 10 ip 10.1.10.254

vrrp 10 priority 50

**Configuring OSPF on IOS Devices**

In this recipe we will outline how to configure OSPF on Cisco IOS devices with ansible. Using our sample network topology we will setup OSPF between Core Switches and WAN Routers as well as advertise the SVI interface via OSPF.

**Getting Ready**

This recipe assumes that all the Interfaces are already configured with the correct IP addresses following the same procedures outlined in the previous recipes.

**How to do it..**

* Update the core.yml file (under group\_vars folder) with the below data

core\_l3\_links:

core01:

- name: Ethernet1/0

description: wan01\_Gi2

ipv4: 10.3.1.0/30

ospf: yes

ospf\_metric: 100

peer: wan01

core02:

- name: Ethernet1/0

description: wan02\_Gi2

ipv4: 10.3.1.4/30

ospf: yes

ospf\_metric: 200

peer: wan02

* Update the update the pb\_build\_network.yml file with the below data

- name: "PLAY 2: Configure Core Switches"

hosts: core

tags: l3\_core

tasks:

< -------- Snippet -------- >

- name: "P2T9: Configure OSPF On Interfaces"

ios\_config:

parents: interface {{ item.name }}

lines:

- ip ospf {{ ospf\_process }} area {{ ospf\_area }}

- ip ospf network point-to-point

- ip ospf cost {{item.ospf\_metric | default(ospf\_metric)}}

loop: "{{ (svi\_interfaces + core\_l3\_links[inventory\_hostname]) | selectattr('ospf') | list }}"

- name: "P2T10: Configure OSPF Passive Interfaces"

ios\_config:

parents: router ospf {{ ospf\_process }}

lines: passive-interface {{item.name}}

loop: "{{ (svi\_interfaces + core\_l3\_links[inventory\_hostname]) | selectattr('ospf','equalto','passive') | list }}"

**How it works..**

We created another dictionary data structure in the core.yml file that describes the L3 links between the Core switches and the WAN routers. We specified whether they will run OSPF and what is the OSPF metric on these links.

We created two separate tasks using the **ios\_config** in order to push the OSPF related configuration on each device. In the first task we configured the Interface related parameters under each interface and we looped over both the svi\_interface and core\_l3\_interfaces data structures to enable OSPF on all the OSPF enabled interfaces. We used the sellectattr jinja2 filter to select all the interface which has the ospf attribute.

In the last task we apply the passive interface configuration on all the interface which has the passive flag enabled on them, we again use the sellectattr filter to filter on only those interfaces with which are flagged as passive.

**Collecting IOS Device facts**

In this recipe we will outline how to collect several information from the devices which ansible denote it as facts. some of this information is the serial number, IOS version and all the interfaces on the devices. Ansible execute several commands on the managed IOS devices in order to collect this information.

**Getting Ready**

The ansible controller must have IP connectivity towards the managed network devices and SSH must be enabled on the IOS devices.

**How to do it..**

* Create a new playbook called **pb\_collect\_facts.yml** in the same folder ios\_netops with the below info

---

- name: "PLAY 1: Collect Device Facts"

hosts: core,wan

tasks:

- name: "P1T1: Gather Device Facts"

ios\_facts:

register: device\_facts

run\_once: yes

- debug: var=device\_facts

**How it works..**

We run this new playbook against all nodes within the core and wan group and we use the **ios\_facts** module to collect the several information from the managed IOS devices. In this recipe we use the debug module to print out the information that was collected from the ios\_facts module. Below is a subset of the information that was discovered.

ok: [core01 -> localhost] => {

"ansible\_facts": {

"net\_all\_ipv4\_addresses": [

"172.20.1.20",

"10.1.10.1",

"10.1.200.1",

"10.3.1.1",

"10.1.20.1",

"10.100.1.1",

"10.1.100.1"

],

"net\_hostname": "core01",

"net\_interfaces": {  
< ---------- Snippet ------------ >

"Loopback0": {

"bandwidth": 8000000,

"description": null,

"duplex": null,

"ipv4": [

{

"address": "10.100.1.1",

"subnet": "32"

}

],

"lineprotocol": "up",

"macaddress": null,

"mediatype": null,

"mtu": 1514,

"operstatus": "up",

"type": null

},

"Vlan10": {

"bandwidth": 1000000,

"description": null,

"duplex": null,

"ipv4": [

{

"address": "10.1.10.1",

"subnet": "24"

}

],

"lineprotocol": "up",

"macaddress": "aabb.cc80.e000",

"mediatype": null,

"mtu": 1500,

"operstatus": "up",

"type": "Ethernet SVI"

},

},

"net\_iostype": "IOS",

"net\_memfree\_mb": 884124.484375,

"net\_memtotal\_mb": 975524.8125,

"net\_serialnum": "67109088",

"net\_system": "ios",

"net\_version": "15.1",

}  
< ------------ Snippet ------------ >

}

From the above output we can see some of the main facts that the ios\_facts module has captured from the devices like

* **net\_all\_ipv4\_addresses**, this list data structure contains all the ipv4 addresses that are configured on all the interfaces on the IOS device.
* **net\_interfaces**, this dictionary data structure capture the status of all the Interfaces on this device and their operational state as well as other important information like description and their operational state.
* **net\_serialnum**, this capture the serial number of the device
* **net\_version**, this capture the IOS version running on this device.

For more information regarding the ios\_facts module please check the documentation https://docs.ansible.com/ansible/latest/modules/ios\_facts\_module.html

**There is More..**

Using the information that is collected from the **ios\_facts** module we can generate structured reports for the current state of network and use these reports in further tasks. In this section we will outline how to modify our playbook to build this report.

* We add a new task in the **pb\_collect\_facts.yml** playbook as shown below

- name: "P1T2: Write Device Facts"

blockinfile:

path: ./facts.yml

create: yes

block: |

device\_facts:

{% for host in play\_hosts %}

{% set node = hostvars[host] %}

{{ node.ansible\_net\_hostname }}:

serial\_number: {{ node.ansible\_net\_serialnum }}

ios\_version: {{ node.ansible\_net\_version }}

{% endfor %}

all\_loopbacks:

{% for host in play\_hosts %}

{% set node = hostvars[host] %}

{% if node.ansible\_net\_interfaces is defined %}

{% if node.ansible\_net\_interfaces.Loopback0 is defined %}

- {{ node.ansible\_net\_interfaces.Loopback0.ipv4[0].address }}

{% endif %}

{% endif %}

{% endfor %}

run\_once: yes

delegate\_to: localhost

We use the **blockinfile** module to build a YAML file called facts.yml and we use JINJA2 expressions within the blockinfile module to customize and select the information we want to capture from the ansible facts that was captured from the ios\_facts task. When we run the pb\_collect\_facts.yml playbook we generate the facts.yml file which has the following data

device\_facts:

wan01:

serial\_number: 90L4XVVPL7V

ios\_version: 16.06.01

wan02:

serial\_number: 9UOFOO7FH19

ios\_version: 16.06.01

core01:

serial\_number: 67109088

ios\_version: 15.1

core02:

serial\_number: 67109104

ios\_version: 15.1

all\_loopbacks:

- 10.100.1.3

- 10.100.1.4

- 10.100.1.1

- 10.100.1.2

**Validating Network Reachability on IOS Devices**

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

**Getting Ready**

This receipe is built based on the network setup that was outlined in the chapter summary and I am assuming that the network is already build following all the previous recipes in this chapter.

**How to do it..**

* Create a new playbook called pb\_net\_validate.yml and populate it as shown below

---

- name: "PLay 1: Validate Network Reachability"

hosts: core,wan

vars:

host\_id: 10

packet\_count: 10

tasks:

- name: "P1T1: Get all SVI Prefixes"

set\_fact:

all\_svi\_prefixes: "{{ svi\_interfaces | selectattr('vrrp') |

map(attribute='ipv4') | list }}"

run\_once: yes

delegate\_to: localhost

tags: svi

- name: "P1T2: Ping Hosts in all VLANs"

ios\_ping:

dest: "{{ item | ipaddr(10) | ipaddr('address') }}"

loop: "{{ all\_svi\_prefixes }}"

tags: svi

**How it works..**

In this playbook we are using the ios\_ping module which loges into each node in the playbook hosts and ping the destination specified by the dest attribute. In this sample paybook we would like to validate network reachability to a single host within the data ,voice and web vlans. In order to build all the VLAN prefixes we we set in the first task a new variable called all\_svi\_prefixes and use multiple jinja2 filters to collect only prefixes which are running VRRP (so as to remove any core VLANs) and we get only the ipv4 attribute for these SVI interfaces. Below is the contents of this new variable after running the first task

ok: [core01 -> localhost] => {

"all\_svi\_prefixes": [

"10.1.10.0/24",

"10.1.20.0/24",

"10.1.100.0/24"

]

}

We supply this new list data structure to the ios\_ping module and we specify that we need to ping the 10th host within each Subnet. As long as the ping succeed the task will succeed however if there is a conectivty problem from the Router/Switch to this host the task will fail as show below

TASK [P1T2: Ping Hosts in all VLANs] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [core01] => (item=10.1.10.0/24)

ok: [core02] => (item=10.1.10.0/24)

ok: [wan01] => (item=10.1.10.0/24)

ok: [wan02] => (item=10.1.10.0/24)

ok: [core01] => (item=10.1.20.0/24)

ok: [core02] => (item=10.1.20.0/24)

ok: [core01] => (item=10.1.100.0/24)

ok: [wan01] => (item=10.1.20.0/24)

ok: [wan02] => (item=10.1.20.0/24)

ok: [core02] => (item=10.1.100.0/24)

ok: [wan01] => (item=10.1.100.0/24)

ok: [wan02] => (item=10.1.100.0/24)

**Retrieving Operational data from IOS 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**

**How to do it..**

* Create a new playbook called pb\_op\_cmds.yml and populate it as shown below

---

- name: "Play 1: Execute Operational Commands"

hosts: network

vars:

config\_folder: "configs"

op\_folder: "op\_data"

op\_cmds:

- show ip ospf neighbor

- show ip route

tasks:

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

block:

- name: "Create folder to store Device config"

file:

path: "{{ config\_folder }}"

state: directory

- name: "Create Folder to store operational commands"

file:

path: "{{ op\_folder }}"

state: directory

run\_once: yes

delegate\_to: localhost

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

ios\_command:

commands: show running-config

register: show\_run

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

copy:

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

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

- name: "P1T4: Create Folder per Device"

file:

path: "{{ op\_folder}}/{{ inventory\_hostname }}"

state: directory

delegate\_to: localhost

- name: "P1T5: Get Operational Data from Devices"

ios\_command:

commands: "{{ item }}"

register: op\_output

loop: "{{ op\_cmds }}"

- name: "P1T6: Save output per each node"

copy:

content: "{{ item.stdout[0] }}"

dest: "{{ op\_folder}}/{{ inventory\_hostname }}/{{item.item | replace(' ', '\_')}}.txt"

loop: "{{ op\_output.results }}"

**How it works..**

In this receipe we are using the **ios\_commands** module in order to execute operational commands on the IOS devices and save to text files. In order to achieve this goal we followed the following steps

* We create the folders which we will store the output into, we create a folder called **configs** to store the running config of all the devices and also created an **op\_data** to store the output of the operational commands that we will get from the devices.
* We then execute the show running command on all the IOS devices in our inventory and we register the output in a new variable ( show\_run).
* We use the copy module to save the output from the previous task into a file per devices, the output from the command run is saved in the stdout variable, and since we executed a single command the stdout is only having a single item (stdout[0]). Once we execute this task we see that the configs folder is populated as shown below

$ tree configs/

configs/

├── access01.cfg

├── access02.cfg

├── core01.cfg

├── core02.cfg

├── isp01.cfg

├── wan01.cfg

└── wan02.cfg

* For the next part we create a folder per node to store the output from the multiple show commands that we will execute on the IOS devices.
* We use the ios\_commands module to execute the show commands on the devices and save all the output in a new variable (op\_output).
* We use again the copy module to write the output of these commands on a separate file per device. We take the command (show ip route ) and we create a file with it as show\_ip\_route.txt.
* After running this task we can see that this is the current structure of the op\_data folder

$ tree op\_data/

op\_data/

├── access01

│   ├── show\_ip\_ospf\_neighbor.txt

│   └── show\_ip\_route.txt

├── access02

│   ├── show\_ip\_ospf\_neighbor.txt

│   └── show\_ip\_route.txt

├── core01

│   ├── show\_ip\_ospf\_neighbor.txt

│   └── show\_ip\_route.txt

├── core02

│   ├── show\_ip\_ospf\_neighbor.txt

│   └── show\_ip\_route.txt

├── isp01

│   ├── show\_ip\_ospf\_neighbor.txt

│   └── show\_ip\_route.txt

├── wan01

│   ├── show\_ip\_ospf\_neighbor.txt

│   └── show\_ip\_route.txt

└── wan02

├── show\_ip\_ospf\_neighbor.txt

└── show\_ip\_route.txt

* We can check the content of one of the files to confirm that all the data is stored

$ head op\_data/core01/show\_ip\_ospf\_neighbor.txt

Neighbor ID Pri State Dead Time Address Interface

10.100.1.3 0 FULL/ - 00:00:37 10.3.1.2 Ethernet1/0

10.100.1.2 0 FULL/ - 00:00:36 10.1.200.2 Vlan200