# Chapter 4 (JunOS SP)

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

1. Ansible Machine Running Ubuntu 16.04
2. Ansible 2.8
3. Juniper vMX running JunOS 14.1R8 and JunOS 17.1R1 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.
* Deploying Configuration on JunOS Devices.
* Configuring L3VPN Service on JunOS Devices.
* Validating Network reachability on JunOS devices.
* Retrieving Operational Data from JunOS Devices

**Building Network Inventory**

In this recipe, we will outline how to build and structure the Ansible Inventory to describe the sample SP network setup outlined above. The Ansible inventory is a pivotal part in ansible as it outlines and group devices that should be managed by Ansible.

**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..**

1. 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

[p]

mxp01 ansible\_host=172.20.1.2

mxp02 ansible\_host=172.20.1.6

[junos]

mxpe01

mxpe02

mxp01

mxp02

[core:children]

pe

p

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..**

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..**

1. 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

1. 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

1. 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 paste 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..**

1. Create a new playbook called pb\_junos\_net\_build.yml as show below

$ cat pb\_junos\_net\_build.yml

- 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

1. 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: netconf

**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 recipe 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. Since we are going to use the netconf API in all interactions with Juniper Devices in all coming recipes , we enabled the network\_cli only for the junos\_netconf task in this playbook via the **vars** attribute. However for all future tasks that we will add in this playbook we will use the netconf connection specified in the ansible\_connection attribute in the group\_vars/junos.yml file .

We create a new Playbook called pb\_junos\_net\_build.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;

}

}

**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 NETCONF is enabled on all Juniper Devices as per the previous receipe.

**How to do it..**

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

$ cat pb\_junos\_net\_build.yml  
  
---

- 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**

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/junos\_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 **junos** inside this templates folder.
2. Create a new jinja2 file **mgmt..j2** inside the templates/junos directory as shown below

$ cat templates/junos/mgmt.j2

system {

host-name {{inventory\_hostname}};

root-authentication {

encrypted-password "{{ global.root\_pwd}}"; ## SECRET-DATA

}

login {

{% for user in global.users if user.hash is defined %}

user {{ user.username }} {

class super-user;

authentication {

encrypted-password "{{user.hash}}"; ## SECRET-DATA

}

}

{% endfor %}

{% for user in global.users if user.ssh\_key is defined %}

user {{ user.username }} {

class {{ user.role }};

authentication {

ssh-rsa "{{lookup('file',user.ssh\_key)}}"; ## SECRET-DATA

}

}

{% endfor %}

}

services {

ssh;

netconf {

ssh {

{{ global.netconf\_port }};

}

}

}

}

1. Create the **all.yml** file under the group\_vars directory with the below contents

global:

dns:

- 172.20.1.1

- 172.20.1.15

root\_pwd: $1$ciI4raxU$XfCVzABJKdALim0aWVMql0

netconf\_port: 830

users:

- role: super-user

ssh\_key: ~/.ssh/id\_rsa.pub

username: vagrant

- hash: $1$mR940Z9C$ipX9sLKTRDeljQXvWFfJm1

passwd: 14161C180506262E757A60

role: super-user

username: ansible

**How it works..**

In order to configure the various system parameters on Juniper 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 junos\_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 mxp01 router which is output after running the playbook.

system {

host-name mxp01;

name-servers {

172.20.1.1;

172.20.1.15;

}

root-authentication {

encrypted-password "$1$ciI4raxU$XfCVzABJKdALim0aWVMql0"; ## SECRET-DATA

}

login {

user ansible {

class super-user;

authentication {

encrypted-password "$1$mR940Z9C$ipX9sLKTRDeljQXvWFfJm1"; ## SECRET-DATA

}

}

user vagrant {

class super-user;

authentication {

ssh-rsa "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC1AfJkNkEqZYTZehltoEZwc7MyKipFlEBrTOQQVlnMqQ0tLWaPnnXKhRsplojCGyIXmAI0QijTcKzk8G87fA1UlAIaaPosDHMJtOs2RU7Cjfy0qvdEPGbyaELDNsiejfNToPqtloOtdNqnYL2W+N7jF1lqjZUmpMYlwCx3x49/NFj4Xyu9dTgZyz58/2z/Hc4Z8got5kYaGnHqopanSRDhHRX2d9CvEeULiLTtH6aRtZdAP9D0DsKjEx+Y5twLEW3MXWPI2vV7apOV2O/4eRyiL0v/c+mQxJuxsoog3hBCPf7E2mw/E4rKETr4LM7byKnXatGBnqy3nHwIyvT9AqSD vagrant@ubuntu-xenial"; ## SECRET-DATA

}

}

}

services {

ssh;

telnet;

netconf {

ssh {

port 830;

}

}

}

}

At this stage we have generated the system configuration for all the juniper devices 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 juniper devices.

**There is More..**

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

---

- name: "Conifgure Basic System config"

junos\_system:

hostname: "{{ inventory\_hostname }}"

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

state: present

- name: "Configure Users"

junos\_user:

name: "{{ item.username }}"

role: "{{ item.role }}"

sshkey: "{{ lookup ('file', item.ssh\_key) }}"

state: present

with\_items: "{{ global.users | selectattr('ssh\_key','defined') | list }}"

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 juniper device the jinja2 template approach is more advices.

**Configuring Interfaces on JunOS Devices**

In this recipe we will outline how to configure different Interface parameters on Juniper devices like interface MTU and the Loopback and Point to Point IP addresses on all the Juniper devices in our sample topology.

**Getting Ready**

We will be building on the previous recipe outlined in this chapter

**How to do it..**

1. On 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"**

1. Add the below contents to the **all.yml** file inside the group\_vars directory

p2p\_ip:

mxp01:

- {port: ge-0/0/0, ip: 10.1.1.2 , peer: mxpe01, pport: ge-0/0/0, peer\_ip: 10.1.1.3}

- {port: ge-0/0/1, ip: 10.1.1.4 , peer: mxpe02, pport: ge-0/0/0, peer\_ip: 10.1.1.5}

- {port: ge-0/0/2, ip: 10.1.1.6 , peer: xrpe03, pport: GigabitEthernet0/0/0/0, peer\_ip: 10.1.1.7}

- {port: ge-0/0/3, ip: 10.1.1.0 , peer: mxp02, pport: ge-0/0/3, peer\_ip: 10.1.1.1}

mxp02:

<-- Output Trimmed for brevity ------>  
 mxpe01:

<-- Output Trimmed for brevity ------>

mxpe02:

<-- Output Trimmed for brevity ------>

xrpe03:

<-- Output Trimmed for brevity ------>

lo\_ip:

mxp01: 10.100.1.254/32

mxp02: 10.100.1.253/32

mxpe01: 10.100.1.1/32

mxpe02: 10.100.1.2/32

xrpe03: 10.100.1.3/32

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

interfaces {

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

{{ intf.port.split('.')[0] }} {

description "peer:{{intf.peer}} -- peer\_port: {{intf.pport}}"

{% if intf.vlan is defined and intf.vlan != 0 %}

vlan-tagging;

encapsulation flexible-ethernet-services;

unit {{intf.vlan}} {

vlan-id {{intf.vlan}};

family inet {

address {{intf.ip}}/{{global.p2p\_prefix}};

}

family mpls;

}

}

{% else %}

unit 0 {

family inet {

address {{intf.ip}}/{{global.p2p\_prefix}};

}

family mpls;

}

}

{% endif %}

{% endfor %}

lo0 {

unit 0 {

family inet {

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

}

}

}

**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 of Juniper devices 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\_junos\_net\_build** we are importing all the tasks inside the tasks/junos\_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 mxp02 device after running the playbook.

interfaces {

ge-0/0/0 {

description "peer:mxpe01 -- peer\_port: ge-0/0/1"

unit 0 {

family inet {

address 10.1.1.8/31;

}

family mpls;

}

}

ge-0/0/1 {

description "peer:mxpe02 -- peer\_port: ge-0/0/1"

unit 0 {

family inet {

address 10.1.1.10/31;

}

family mpls;

}

}

<-- Output Trimmed for brevity ------>

lo0 {

unit 0 {

family inet {

address 10.100.1.253/32;

}

}

}

**There is More..**

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

---

- name: "Configure the Physical Interfaces"

junos\_interface:

name: "{{ item.port }}"

enabled: true

description: "peer:{{item.peer}} remote\_port:{{item.pport }}"

mtu: "{{ global.mtu | default(1500) }}"

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

- name: "Configure IP Addresses"

junos\_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 OSPF On Juniper Devices**

In this recipe, we will outline how to configure OSPF on Juniper devices as the IGP protocol in our sample network toplolgy along with different OSPF parameters like ospf link-type and OSPF interface cost.

**How to do it..**

1. 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"**

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

protocols {

ospf {

area {{global.ospf\_area}} {

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

interface {{ intf.port }} {

interface-type p2p;

metric {{intf.cost | default(100)}};

}

{% endfor %}

interface lo0.0 {

passive;

}

}

}

}

**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 MPLS on Juniper devices**

In this recipe, we will outline how to configure MPLS and some of the replated protocols like LDP and RSVPN on Juniper devices. We will outline how to generate the required MPLS configuration using Ansible and JINJA2 templates.

**How to do it..**

1. 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"**

1. 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. We will outline how to setup BGP and BGP Route Reflectors as part of our sample topololgy along with all the required BGP Address families to support VPN Services.

**How to do it..**

1. 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"**

1. 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 %}

<-- Output Trimmed for brevity ------>

{% endfor %}

{% for p in bgp\_peers %}

neighbor {{ p.peer}};

{% endfor %}

}

}

{% endif %}

}

1. 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 variables 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 the configuration that we have generated via JINJA2 templates in all the previous sections on juniper devices using ansible.

**Getting Ready**

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

**How to do it..**

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

---

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

hosts: junos

tasks:

<-- Output Trimmed 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**

**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 model and configure L3VPNs on Juniper devices using various ansible modules.

**Getting Ready**

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

**How to do it..**

1. 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

1. Create a new playbook called **pb\_junos\_l3vpn.yml** with the below tasks to configure the PE-CE Links.

---

- 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 }}"

1. Add the below tasks in **pb\_junos\_l3vpn.yml** to setup the P2P IP address on the PE-CE Links

- 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

1. Add the below task in **pb\_junos\_l3vpn.yml** to configure the VRFs on the PE nodes.

- 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 grape 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. This will enable us to validate that the network reachability across our sample network topology.

**Getting Ready**

This recipe assumes 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..**

1. 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 Juniper devices and store these outputs 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..**

1. Install jxmlease python package as shown below.

$ pip3 install jxmlease

1. 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.

**There is More..**

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"