# 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 14.1R8 and JunOS 17.1R1 Release

The main recipes covered in this chapter is shown below

* Installing NAPALM and Integration with Ansible.
* Building Ansible Network Inventory.
* Connecting and Authenticating to Network Devices using NAPALM.
* Building Device Configuration.
* Pushing Configuration with NAPALM.
* Building Ansible Role with NAPALM for L3VPN.
* Collecting device Facts with NAPALM.
* Collecting Routing Information using NAPALM.
* Network Validation with NAPALM.
* Validating Network reachability using NAPALM.

**Install NAPALM**

In this recipe we outline how to install NAPALM and integrate it to work with ansible. This task is mandatory since NAPALM is not part of the core modules that is shipped with ansible by default that is why we need to install it and tell ansible where to find it to start working with the specific modules developed by NAPALM team for ansible.

**Getting Ready**

You need to have sudo access on the machine in order to install napalm with ansible as well as have python already installed and python PIP package which we will use to install napalm.

**How to do it..**

* Install napalm-ansible python package as shown below

$ sudo pip3 install napalm-ansible

* Run the command napalm-ansible as shown below

$ napalm-ansible  
  
To ensure Ansible can use the NAPALM modules you will have

to add the following configurtion to your Ansible configuration

file (ansible.cfg):

[defaults]

library = /usr/local/lib/python3.5/dist-packages/napalm\_ansible/modules

action\_plugins = /usr/local/lib/python3.5/dist-packages/napalm\_ansible/plugins/action

For more details on ansible's configuration file visit:

https://docs.ansible.com/ansible/latest/intro\_configuration.html

* Create a New folder called ch6\_netops and create the ansible.cfg file and update it as shown below

$ cat ansible.cfg  
  
[defaults]

inventory=./hosts

retry\_files\_enabled=False

gathering=explicit

host\_key\_checking=False

library = /usr/local/lib/python3.5/dist-packages/napalm\_ansible/modules

action\_plugins = /usr/local/lib/python3.5/dist-packages/napalm\_ansible/plugins/action

**How it works..**

Since the NAPALM package and module is not part of the core modules shipped and installed by default with Ansible, we need to install it to the system in order to start working with the NAPALM ansible modules. NAPALM team has shipped a specific python package to to install NAPALM along with all the Ansible modules and all the dependencies in order to start working with NAPALM from inside Ansible. The installed is napalm-ansible and we use the pip program to install this package and we specifically use the pip3 since we are using python3 and installed the python3-pip to install the pip module.

In order to tell ansible where the ansible module are install we need to include the path for these modules into ansible. NAPALM team also provide a simple instruction on how to find the path where the NAPALM modules are installed and how to integrate it with ansible via the **napalm-ansible** program which when we run it it output the required configuration that we need to include in the **ansible.cfg** file so as ansible can find the NAPALM modules that we will be using.

We update the ansible.cfg file the the output that we obtained from the napalm-ansible command output mainly for the library and action\_plugins options which tell ansible to include these folder in its path when it is searching for modules or action plugins. In the ansible.cfg file we include the normal configuration that we used before in the previous chapters.

**Building Network Inventory**

In this chapter, we will outline how to build and structure our Ansible Inventory to describe our sample SP network setup outlined in this chapter. Building an Ansible inventory is a mandatory step in order to tell ansible how to connect to the managed devices and in case of NAPALM in order to classify the different nodes into the correct vendor type supported by NAPALM.

**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 (ch6\_napalm\_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

mxp02 ansible\_host=172.20.1.6

[junos]

mxpe01

mxpe02

mxp01

mxp02

[iosxr]

xrpe03

[sp\_core:children]

pe

p

**How it works..**

We built the ansible inventory using the ***hosts*** file and we defined multiple groups in order to segment our infrastructure as shown below

* 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 **junos** group to reference all the Juniper Devices in our topology.
* We created the **iosxr** group to reference all the nodes running IOS-XR.

Segmenting and defining groups per each vendor/os is mandatory when working with NAPALM since we use these groups to specify the specific connection setting and also the parameters required for NAPALM to establish network connectivity with the devices. How we are going to use these groups and populate them with the correct parameter to connect to the devices is outlined in the next receipe.

**Connecting to Network Devices using NAPALM**

In this recipe, we will outline how to connect to both Juniper and IOS-XR nodes using NAPALM in order to start interacting with the 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 Juniper Devices configure the username and password as shown below

system {

login {

user ansible {

class super-user;

authentication {

encrypted-password "$1$mR940Z9C$ipX9sLKTRDeljQXvWFfJm1"; ## ansible123

}

}

}

}

* On The Cisco IOS-XR Device configure the username and password as shown below

!

username ansible

group root-system

password 7 14161C180506262E757A60 # ansible123

!

* Enable NETCONF on the Juniper Devices

system {

services {

netconf {

ssh {

port 830;

}

}

}

}

* On the IOS-XR Devices we need to enable SSH as well as enable xml-agent

!

xml agent tty

iteration off

!

xml agent

!

ssh server v2

ssh server vrf default

* On the Ansible machine create the group\_vars directory under the ch6\_napalm\_netops and create the junos.yml and iosxr.yml as shown below

$ cat group\_vars/iosxr.yml  
  
ansible\_network\_os: junos

ansible\_connection: netconf

ansible\_user: ansible

ansible\_ssh\_pass: ansible123  
  
$ cat group\_vars/junos.yml  
  
ansible\_network\_os: iosxr

ansible\_connection: network\_cli

ansible\_user: ansible

ansible\_ssh\_pass: ansible123

**How it works..**

NAPALM uses specific transport API to connect to the different vendor devices as outlined in this chapter introduction, thus in our sample topology it required NETCONF to be enabled on the Juniper devices as well as SSH enabled on Cisco IOS-XR devices. Also the username/password used on the ansible control machine to authenticate with the devices must be configured on the remote nodes. We perform all these steps on the devices in order to make it read for NAPALM to communicate with the devices.

On the Ansible machine we set the **ansible\_connection** per each vendor ( netconf for juniper and network\_cli for iosxr) and we specify the ansible\_network\_os to designate the vendor OS. We specify the username and password via ansible\_user and ansibe\_ssh\_pass. All these parameters are defined under the group\_vars hierarichy under junos.yml and iosxr.yml corresponding to the groups that we defined in our inventory to group the devices on vendor os basics.

We can test and validate that we can communicate with the devices from the ansible control machine using the ansible ping module as shown below

$ ansible all -m ping

mxpe01 | SUCCESS => {

"changed": false,

"ping": "pong"

}

mxpe02 | SUCCESS => {

"changed": false,

"ping": "pong"

}

mxp02 | SUCCESS => {

"changed": false,

"ping": "pong"

}

mxp01 | SUCCESS => {

"changed": false,

"ping": "pong"

}

xrpe03 | SUCCESS => {

"changed": false,

"ping": "pong"

}

**Building Device Configuration**

NAPALM doesn’t provide declarative modules to configure the various system parameters on the managed devices. However, it provide a common API to push text based configuration to all the devices so it requires the configuration for the devices to be present in text format in order to push the required configuration. In this recipe we will create the configuration needed on the devices as we outlined in the previous chapters using ansible template modules along with the JINJA2 templated in order to generate the required configuration. In the next recipe we will outline how to push the configuration to the remote managed devices using NAPALM

**Getting Ready**

As a prerequisite for this recipe, an ansible inventory file must be present.

**How to do it..**

* Create a new playbook called pb\_napalm\_net\_build.yml as show below

$ cat pb\_napalm\_net\_build.yml

---

- name: " Generate and Deploy Configuration on All Devices"

hosts: sp\_core

tasks:

- name: "P1T1: Build Config Directory Structure"

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

delegate\_to: localhost

tags: config

- name: "P1T2: Build Devices configuration"

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

delegate\_to: localhost

tags: config

* Create the tasks folder and create the build\_req\_dir.yml file with the below contents

$ cat tasks/build\_req\_dir.yml  
  
---

- name: "Create Config Directory"

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

run\_once: yes

- name: "Create Tem Directory"

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

run\_once: yes

- name: "Create Per host directory"

file: path={{tmp\_dir}}/{{inventory\_hostname}} state=directory

* Create the build\_config.yml under the tasks folder with the below contents

$ cat tasks/build\_req\_dir.yml

---

- name: "System Configuration"

template:

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

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

- name: "Interface Configuration"

template:

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

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

< -- Output Omitted for brevity -->

- name: "BGP Configuration"

template:

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

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

* Create the **all.yml** file under the group\_vars with the below contents as shown below

$ cat group\_vars/all.yml  
  
tmp\_dir: ./tmp

config\_dir: ./configs  
  
p2p\_ip:  
  
< -- Output Omitted for brevity -->  
  
 xrpe03:

- {port: GigabitEthernet0/0/0/0, ip: 10.1.1.7 , peer: mxp01, pport: ge-0/0/2, peer\_ip: 10.1.1.6}

- {port: GigabitEthernet0/0/0/1, ip: 10.1.1.13 , peer: mxp02, pport: ge-0/0/2, peer\_ip: 10.1.1.12}

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

* Create a specific directory for each host under the host\_vars directory and under each directory create the **bgp.yml** file with the below contents as shown below

$ cat host\_vars/xrpe03/bgp.yml

bgp\_asn: 65400

bgp\_peers:

- local\_as: 65400

peer: 10.100.1.254

remote\_as: 65400

* Create the templates directory and create **junos** and **iosxr** directories inside the template directory.
* Create jinja2 templates for the different configuration sections for the devices like Interfaces, OSPF, MPLS and BGP as shown below

$ cat templates/iosxr/ospf.j2

!

router ospf {{ global.ospf\_pid }}

address-family ipv4 unicast

area 0

interface Loopback0

passive enable

!

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

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

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

network point-to-point

!

{% endfor %}

* Update the playbook pb\_napalm\_net\_build.yml as show below

$ cat pb\_napalm\_net\_build.yml

---

- name: " Generate and Deploy Configuration on All Devices"

hosts: sp\_core

tasks:

< -- Output Omitted for brevity -->  
  
 - name: "P1T3: Remove Old Assembled Config"

file:

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

state: absent

delegate\_to: localhost

tags: config

- name: "P1T4: Assemble The Final configuration"

assemble:

src: "{{tmp\_dir}}/{{ inventory\_hostname }}"

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

delegate\_to: localhost

tags: config

**How it works..**

In this recipe our main goal is to create the configuration that we need to push to the devices in our topology. We are following the same process and procedures that we have used to generate the configuration snipped for the same network topology in Chpater3. for JunOS devices. We are repeating the steps again however we add the templates and the required data to include the IOS-XR devices in our topology.

Below is a quick explanation for the steps as a quick review

* **Modelling the Network via Ansible Variables**

We describe the different aspects of our Network topology like P2P interface, Loopback Interfaces and OSPF parameters under different data structures in the **all.,yml** file under the group\_vars. For any host specific data we use the host\_vars directory to populate all varaibles/paramters which are specific to a specific node and in our case we use this approach for bgp data to outline the bgp\_peers for each node. This expose all these variables (like p2p\_ip and lo\_ip) to the devices in our Ansible Inventory and help us to populate the JINJA2 templates with this data in order to generate the final configuration for our each device in our sample Network.

* **Building the JINJA2 templates**

We place all our JINJA2 templates under the **templates** folder and we segment our JINJA2 templates per the vendor OS in a separate folder and we create a JINJA2 template for each section of the configuration the below snippet outline the directory structure for the templates

templates/

├── iosxr

│   ├── bgp.j2

│   ├── intf.j2

│   ├── l3vpn.j2

│   ├── mgmt.j2

│   ├── mpls.j2

│   └── ospf.j2

└── junos

├── bgp.j2

├── intf.j2

├── l3vpn.j2

├── mgmt.j2

└── mpls.j2

├── mpls.j2

└── ospf.j2

* **Building the Ansible Playbook**

We Create the Ansible playbook which include tasks for

* + Creating the folder structure required to save the output of template module to save the different configuration snippet for each device.
  + Generating the required configuration snippet for each section using the template module.
  + Generating the final configuration for the device using the assemble module.

**Note**  
For a Detailed explanation for the different JINJA2 templates used in this recipe and how the integrate with the Ansible variables defined to generate the final configuration please check the contents of Chapter03 since we are using the exact same Network Topology and the Same Data Structures are used for both JunOS and IOS-XR devices.

Running this playbook will generate the configuration for all the devices in our Ansible Inventory on the configs folder as shown below

lab@NMS:~/net\_automation\_cookbook/ch6\_napalm$ tree configs/

configs/

├── mxp01.cfg

├── mxp02.cfg

├── mxpe01.cfg

├── mxpe02.cfg

└── xrpe03.cfg

**Configuring Network Devices using NAPALM**

In this recipe, we will outline how to push configuration on different Vendor devices using NAPALM Ansible modules. This single Module allow us to have a single common method to push any configuration on any vendor equipment supported by NAPALM and this greatly simplify Ansible playbooks.

**Getting Ready**

To follow along with this recipe, an ansibleAnsible inventory is assumed to be already setupin place and NETCONFNetwork reachability between the Ansible controller and the Network is enabled on all Juniper Devicesestablished. Further, the configuration that will be pushed to the devices is already generated as peroutlined in the previous receiperecipe.

**How to do it..**

* Update the playbook ***pb\_junosnapalm\_net\_build.yml*** file , and add the below highlighted tasks

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

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

hosts: sp\_core

tasks:  
  
< -- Output Omitted for brevity -->  
  
 - name: "P1T5: Deploy Configuration"

napalm\_install\_config:

hostname: "{{ ansible\_host }}"

username: "{{ ansible\_user }}"

password: "{{ ansible\_ssh\_pass }}"

dev\_os: "{{ ansible\_network\_os }}"

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

commit\_changes: "{{commit | default('no')}}"

replace\_config: yes

tags: deploy, never

**How it works..**

As previously outlined, NAPALM provides a single Ansible module to push configuration to the Network devices, it requires the needed configuration to be present in a text file and it connect to the network device and push the configuration to the respective device.

Since we are using a single configuration module that can be used across all the vendor OS devices supported by NAPALM and since NAPALM uses a different connection API to manage the device we need to tell the module the Vendor OS for the device along with other parameters like username/password to login and authenticate with the device.

The napalm\_install\_config module require the below mandatory parameters in order to correctly login to the managed device and push the configuration to it:

* + **hostname**: This is the IP address through which we can reach the device, we supply the value of ansible\_host for this parameter.
  + **username/password**: These are the username and password to connect to the device and we supply the ansible\_user and ansible\_ssh\_pass attributes.
  + **dev\_os**: This parameter provide the vendos OS name that NAPALM require in order choose the correct API to communicate with the device and we provide the ansible\_network\_os parameter.

In order to push the configuration to the device the napalm\_install\_config module use the below parameters to manage the configuration on remote devices

* + **config\_file**: provide the path of the configuration file that contains the configuration that needs to be pushed to the managed device.
  + **Commit\_changes**: whether or not to commit the configuration. NAPALM provides a consistant method for configuration commit even for devices which don’t support it by default like cisco IOS devices.
  + **replace\_config**: this parameter control how to merge between the existing configuration on the device and the configuration in the config\_file. In Our case since we are generating the whole device configuration and all the configuration sections are managed under Ansible, we replace the entire configuration by the configuration that we generate. This will make any configuration on the device not present in our configuration file to be removed.

As per the configuration outlined in this recipe when we run the playbook using the tag deploy NAPALM will connect to the device and push the configuration, however it will not commit the configuration on the remote device since we specify the default value for **commit\_changes** to be no. In case we need to push and commit the configuration on the remote device we can set the value for the **commit** parameter to yes when running the playbook as shown below

$ ansible-playbook pb\_napalm\_net\_build.yml --tags deploy --e commit=yes

**There is More..**

The **napalm\_install\_config module** provide extra options to control how to manage the configuration on the remote devices like configuration Diff. With this option we can collect the difference in the configuration between the running configuration on the device and the configuration that we will push via NAPALM. This option can be enabled as shown below

* Create a folder called **config\_diff** to store the config diff captured by NAPALM as shown below

$ cat group\_vars/all.yml

< -- Output Omitted for brevity -->  
  
config\_diff\_dir: ./config\_diff

$ cat tasks/build\_req\_dir.yml

- name: "Create Config Diff Directory"

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

run\_once: yes

* Update the playbook bp\_napalm\_net\_build.yml as shown below

---

- 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 }}"

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

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

hosts: sp\_core

tasks:  
  
< -- Output Omitted for brevity -->  
  
 - name: "P1T5: Deploy Configuration"

napalm\_install\_config:

hostname: "{{ ansible\_host }}"

username: "{{ ansible\_user }}"

password: "{{ ansible\_ssh\_pass }}"

dev\_os: "{{ ansible\_network\_os }}"

config\_file: "{{config\_dir}}/{{ inventory\_hostname }}.cfg"  
 diff\_file: "{{ config\_diff\_dir}}/{{ inventory\_hostname }}\_diff.txt"

commit\_changes: "{{commit | default('no')}}"

replace\_config: yes

tags: deploy, never

We create a new folder to house all the configuration diff files that we will generate for each device and we add the **diff\_file** parameter to napalm\_install\_config module in order to collect the config diff for each device and save it to the config\_diff directory for each device.

When we run the playbook again with a modified configuration on the devices we can see that the config\_diff files for each device is generated as shown below

$ tree config\_diff/

config\_diff/

├── mxp01\_diff.txt

├── mxpe01\_diff.txt

├── mxpe02\_diff.txt

└── xrpe03\_diff.txt

**Collecting device Facts with NAPALM**

In this recipe we will outline how to collect operational stat from network devices using NAPALM facts Ansible modules. This can be used to validate network state across multi-vendor equipment since NAPALM Ansible facts return a consistent data structure across all vendor OS supported by NAPALM.

**Getting Ready**

To follow along with this recipe, an Ansible inventory is assumed to be already in place and Network reachability between the Ansible controller and the Network is established. Finally , The Network is configured as per the previous recipe.

**How to do it..**

* Create an ansible playbook **pb\_napalm\_get\_facts.yml** with the below contents.

$ cat cat pb\_napalm\_get\_facts.yml

---

- name: " Collect Network Facts using NAPALM"

hosts: sp\_core

tasks:

- name: "P1T1: Collect NAPALM Facts"

napalm\_get\_facts:

hostname: "{{ ansible\_host }}"

username: "{{ ansible\_user }}"

password: "{{ ansible\_ssh\_pass }}"

dev\_os: "{{ ansible\_network\_os }}"

filter:

- bgp\_neighbors

* Update the playbook with the below tasks to validate the data returned by NAPALM facts module

$ cat pb\_napalm\_get\_facts.yml  
  
< -- Output Omitted for brevity -->

- name: Validate All BGP Routers ID is correct

assert:

that: napalm\_bgp\_neighbors.global.router\_id == lo\_ip[inventory\_hostname].split('/')[0]

when: napalm\_bgp\_neighbors

- name: Validate Correct Number of BGP Peers

assert:

that: bgp\_peers | length == napalm\_bgp\_neighbors.global.peers.keys() | length

when: bgp\_peers is defined

- name: Validate All BGP Session Are UP

assert:

that: napalm\_bgp\_neighbors.global.peers[item.peer].is\_up == true

loop: "{{ bgp\_peers }}"

when: bgp\_peers is defined

**How it works..**

We use the ansible module **napalm\_get\_facts** to retrieve the operational state from the network devices. We supply the same parameters (hostname, username/password and dev\_os) that we used with napalm\_install\_config to be able to connect to the devices and collect the required operational state from these devices.

In order to control which information we retrieve using NAPALM we use the filter parameter and supply the required information that we need to retrieve and in this example we are limiting the data retrieved to only **bgp\_neigbors**.

The napalm\_get\_facts module returns the data retrieved from the nodes as ansible facts and this data can be retrieved from the variable **napalm\_bgp\_neighbors** which store all the NAPALM BGP facts retrieved from the device.

The following snippet outline the output from napalm\_bgp\_neigbors retrieved from a **JunOS** devices

ok: [mxpe02] => {

"napalm\_bgp\_neighbors": {

"global": {

"peers": {

"10.100.1.254": {

"address\_family": {

"ipv4": {

"accepted\_prefixes": 0,

"received\_prefixes": 0,

"sent\_prefixes": 0

},

< -- Output Omitted for brevity -->

},

"description": "",

"is\_enabled": true,

"is\_up": true,

"local\_as": 65400,

"remote\_as": 65400,

"remote\_id": "10.100.1.254",

"uptime": 247307

}

},

"router\_id": "10.100.1.2"

}

}

}

The following snippet outline the output from napalm\_bgp\_neigbors retrieved from an **IOS-XR** devices

ok: [xrpe03] => {

"napalm\_bgp\_neighbors": {

"global": {

"peers": {

"10.100.1.254": {

"address\_family": {

< -- Output Omitted for brevity -->

},

"description": "",

"is\_enabled": false,

"is\_up": true,

"local\_as": 65400,

"remote\_as": 65400,

"remote\_id": "10.100.1.254",

"uptime": 247330

}

},

"router\_id": "10.100.1.3"

}

}

}

As we can see the data returned from NAPALM for the BGP information from different network vendor is consistent between different network vendors this simplify parsing this data and allow us to run much simpler playbooks to validate network state.

We use the data returned by NAPALM to compare and validate the operational state of the network against Our Network design that we defined using Ansible variables like **bgp\_peers** in this case. We use the **assert** module to validate multiple BGP information like

* Correct Number of BGP Peers
* BGP Router ID
* All BGP Session are Operational

We use the when statement in the different assert modules in case we have a router in our topology which doesn’t run BGP (mxp02 is an example) so we skip these checks on these nodes.

**See Also..**

The NAPALM get\_fact module can retrieve a huge range of information from the network devices based on the Vendor equipment support and the level of facts supported for this vendor. For example it supports the retrieval of interfaces, IP addresses and LLDP peers for almost all the known networking vendors. For the complete documentation for napalm\_get\_facts module please check the below URL.  
<https://napalm.readthedocs.io/en/latest/integrations/ansible/modules/napalm_get_facts/index.html>

For complete facts/getters supported by NAPALM and their support matrix against vendor equipment please consult the below URL

<https://napalm.readthedocs.io/en/latest/support/>

**Collecting Routing Information using NAPALM**

In this recipe, we will outline how to collect the Routing information for specific destination and how to validate the correct routing setup using NAPALM and Ansible. Validating Routing Setup on network devices is extremely important as it outline the correct forwarding behaviour in our Network and per our Design.

**Getting Ready**

**How to do it..**

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

---

- name: "System Configuration"

template:

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

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

- name: "Interface Configuration"

template:

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

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

**- name: "OSPF Configuration"**

**template:**

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

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

* 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;

}

}

}

}

**Validating Network reachability using NAPALM**

In this recipe, we will outline how to configure MPLS, LDP and RSVPN on Juniper devices.

**Getting Ready**

To follow along with this recipe, an Ansible inventory is assumed to be already in place and Network reachability between the Ansible controller and the Network is established. Finally, The Network is configured as per the previously outline recipe.

**How to do it..**

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

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

- name: "System Configuration"

template:

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

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

- name: "Interface Configuration"

template:

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

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

- name: "OSPF Configuration"

template:

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

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

**- name: "MPLS Configuration"**

**template:**

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

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

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

$ cat host\_vars/core01.yml  
protocols {

ldp {

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

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

{% endfor %}

interface lo0.0;

}

rsvp {

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

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

{% endfor %}

}

mpls {

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

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

{% endfor %}

}

}

**How it works..**

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

protocols {

ldp {

interface ge-0/0/0.0;

interface ge-0/0/1.0;

interface lo0.0;

}

rsvp {

interface ge-0/0/0.0;

interface ge-0/0/1.0;

}

mpls {

interface ge-0/0/0.0;

interface ge-0/0/1.0;

}

}

**Building Ansible Role with NAPALM for L3VPN**

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

**Getting Ready**

**How to do it..**

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

---

- name: "System Configuration"

template:

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

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

- name: "Interface Configuration"

template:

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

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

- name: "OSPF Configuration"

template:

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

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

- name: "MPLS Configuration"

template:

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

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

**- name: "BGP Configuration"**

**template:**

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

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

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

protocols {

{% if bgp\_peers is defined %}

bgp {

group Core {

type internal;

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

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

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

{% endif %}

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

{% if af == 'inet' %}

family inet {

unicast;

}

{% endif %}

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

family inet-vpn {

unicast;

}

{% endif %}

{% if af == 'evpn' %}

family evpn {

signaling;

}

{% endif %}

{% endfor %}

{% for p in bgp\_peers %}

neighbor {{ p.peer}};

{% endfor %}

}

}

{% endif %}

}

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

bgp\_topo:  
 rr: mxp01

af:

- inet

- inet-vpn

peers:

mxp01:

- mxpe01

- mxpe02

- xrpe03

mxpe01:

- mxp01

mxpe02:

- mxp01

xrpe03:

- mxp01

**How it works..**

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

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

protocols {

bgp {

group Core {

type internal;

local-address 10.100.1.254;

cluster 10.100.1.254;

family inet {

unicast;

}

family inet-vpn {

unicast;

}

neighbor 10.100.1.1;

neighbor 10.100.1.2;

neighbor 10.100.1.3;

}

}

}

**Network Validation with NAPALM**

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

**Getting Ready**

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

**How to do it..**

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

---

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

hosts: junos

tasks:

- name: "P1T1: Enable NETCONF"

junos\_netconf:

netconf\_port: 830

state: present

vars:

ansible\_connection: network\_cli

tags: netconf

- name: "P1T2: Build Config Directory Strcuture"

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

delegate\_to: localhost

tags: config

- name: "P1T3: Build Devices configuration"

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

delegate\_to: localhost

tags: config

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

**file:**

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

**state: absent**

**delegate\_to: localhost**

**tags: config**

**- name: "Assemble config"**

**assemble:**

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

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

**delegate\_to: localhost**

**tags: config**

**- name: "Deploy Configuration"**

**junos\_config:**

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

**tags: deploy**

**How it works..**

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

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

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

**There is More..**

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

---

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

hosts: junos

tasks:

- name: "Rollback config"

junos\_config:

rollback: 1

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

**Configuring L3VPN on Juniper Devices**

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

**Getting Ready**

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

**How to do it..**

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

---  
l3vpns:

vpna:

state: present

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

rd: "1:10"

sites:

- node: mxpe01

port: ge-0/0/3.10

ip: 172.10.1.1/24

- node: mxpe02

port: ge-0/0/3.10

ip: 172.10.2.1/24

vpnb:

state: present

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

rd: "1:20"

sites:

- node: mxpe01

port: ge-0/0/3.20

ip: 172.20.1.1/24

- node: mxpe02

port: ge-0/0/3.20

ip: 172.20.2.1/24

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

---

- name: "Deploy L3VPNs on Juniper Devices"

hosts: pe

vars\_files:

- "l3vpn.yml"

tasks:

- name: "Set VPN Interfaces"

set\_fact:

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

l3vpns[item.key].sites |

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

with\_dict: "{{l3vpns}}"

delegate\_to: localhost

- name: "Configure Interfaces for L3VPN Sites"

junos\_config:

lines:

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

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

loop: "{{ l3vpn\_intfs }}"

- name: "Configure IP address for L3VPN Interfaces"

junos\_l3\_interface:

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

ipv4: "{{ item.ip }}"

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

loop: "{{l3vpn\_intfs}}"

tags: intf\_ip

- name: "Configure L3VPNs"

junos\_vrf:

name: "{{ item.key }}"

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

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

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

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

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

with\_dict: "{{l3vpns}}"

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

tags: l3vpn

**How it works..**

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

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

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

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

ansible@mxpe01> show configuration routing-instances

vpna {

instance-type vrf;

interface ge-0/0/3.10;

route-distinguisher 1:10;

vrf-target target:65400:10;

vrf-table-label;

}

vpnb {

instance-type vrf;

interface ge-0/0/3.20;

route-distinguisher 1:20;

vrf-target target:65400:20;

vrf-table-label;

}

**Validate Network Reachability on Juniper Devices**

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

**Getting Ready**

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

**How to do it..**

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

---

- name: "Validate Core Reachability"

hosts: junos

tasks:

- name: "Ping Across All Loopback Interfaces"

junos\_ping:

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

interface: lo0.0

size: 512

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

vars:

ansible\_connection: network\_cli

**How it works..**

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

**Retrieving Operational data from Juniper Devices**

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

**Getting Ready**

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

**How to do it..**

* Install jxmlease python package as shown below.

$ pip3 install jxmlease

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

---

- name: "Get OSPF Status"

hosts: junos

tasks:

- name: "Get OSPF Neigbours Data"

junos\_command:

commands: show ospf neighbor

display: xml

register: ospf\_output

- name: "Extract OSPF Neigbour Data"

set\_fact:

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

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

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

assert:

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

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

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

loop: "{{ospf\_peers}}"

**How it works..**

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

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

"msg": [

{

"rpc-reply": {

"ospf-neighbor-information": {

"ospf-neighbor": [

{

"activity-timer": "34",

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

"neighbor-address": "10.1.1.2",

"neighbor-id": "10.100.1.254",

"neighbor-priority": "128",

"ospf-neighbor-state": "Full"

},

{

"activity-timer": "37",

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

"neighbor-address": "10.1.1.8",

"neighbor-id": "10.100.1.253",

"neighbor-priority": "128",

"ospf-neighbor-state": "Full"

}

]

}

}

}

]

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

**How it works..**

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

---

- name: "Play 1: Execute Operational Commands"

hosts: junos

vars:

log\_folder: "logs"

op\_folder: "op\_data"

op\_cmds:

- show configuration

tasks:

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

block:

- name: "Create folder to store Device config"

file:

path: "{{ log\_folder }}"

state: directory

run\_once: yes

delegate\_to: localhost

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

junos\_command:

commands: "{{ op\_cmds }}"

register: show\_run

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

copy:

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

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