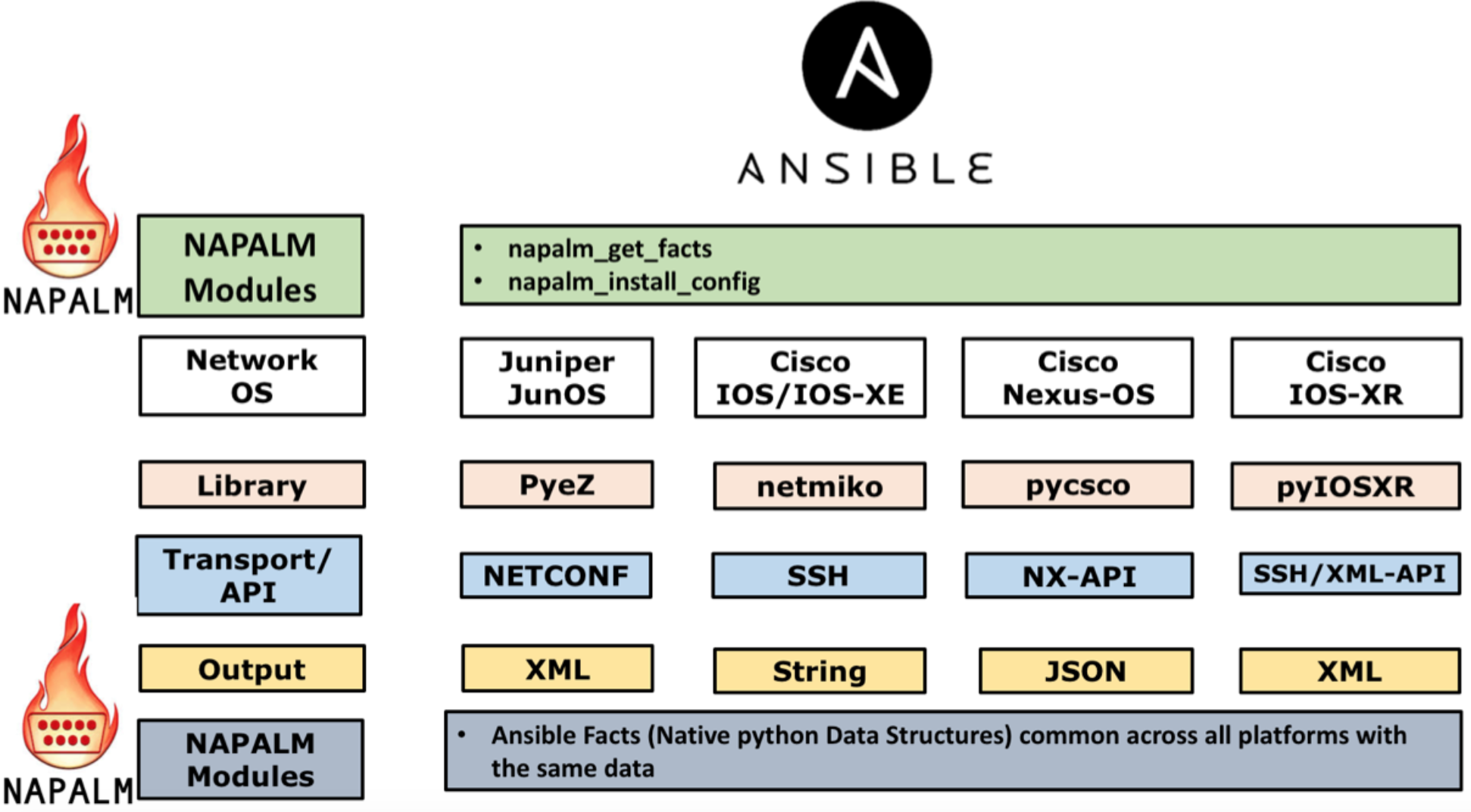
**Chapter Introduction**

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) as the name implies is a multi-vendor python library to interact with different vendor equipment and it provides a consistent method to interact with all these devices irrespective of the vendor equipment which is managed.

As we have shown in all the previous chapters how to interact with different network devices using ansible, however for each vendor OS we use a different ansible module which support this OS and also we saw that the data returned from each vendor OS is completely different. Although that writing playbook for multi-vendor devices is still possible however this require multiple different modules and we need to account for the different data structures returned by these devices. This is the main point that NAPALM tries to address. NAPALM tries to provide a similar ansible module to interact with multiple vendor OS and the data returned by NAPALM from these different vendor OS is normalized and is consistent.

NAPALM interact with each device according to the most common API supported by this node and the API which is widely adopted by the community. The below diagram outline how NAPALM interact with the most common Network devices and the libraries used in NAPALM to interact with these APIs on the devices



Since NAPALM tries to provide a similar and consistent method to interact with network equipment, it supports a specific vendor devices and also it support only the major and most common tasks that is carried on these devices like device configuration, retrieving Operational state for interfaces , BGP and LLDP and many other. For more information regarding the supported devices as well as the supported methods when interacting with these devices please check the below links

<https://napalm.readthedocs.io/en/latest/support/index.html>  
  
In this chapter, we will outline how to automate a multi-vendor Network using NAPALM and ansible.  
We wil outline how to manage the configuration on these different vendor OS as well as how to retrieve operational state from these devices.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
* Cisco IOS-XR 6.1.2

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.
* Collecting device Facts with NAPALM.
* Collecting Routing Information using NAPALM.
* Validating Network reachability using NAPALM.
* Network Validation with 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**

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 a new YAML file **napalm\_route\_validation.yml** to include the routes that we want to validate as shown below

$ cat napalm\_route\_validation.yml  
  
route\_validation:

mxpe01:

- route: 10.100.1.254/32

next\_hop: ['10.1.1.2']

protocol: OSPF

mxpe02:

- route: 10.100.1.254/32

next\_hop: ['10.1.1.4']

protocol: OSPF

* We Create a new playbook **pb\_napalm\_get\_route.yml** and populate it as shown below

$ cat pb\_napalm\_get\_routes.yml  
  
---

- name: " Collect Routing Data via NAPALM"

hosts: junos:&pe

vars\_files: napalm\_route\_validation.yml

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:

- route\_to

args:

route\_to:

destination: "{{ item.route }}"

loop: "{{ route\_validation[inventory\_hostname] }}"

* Update the playbook with the below tasks for validation

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

- name: Validate Active Routing Protocol

assert:

that: napalm\_route\_to[item.route] |

selectattr('protocol','equalto',item.protocol) |

map(attribute='current\_active') | list | first == true

loop: "{{ route\_validation[inventory\_hostname] }}"

- name: Validate Correct Number of Next-Hops

assert:

that: napalm\_route\_to[item.route] |

selectattr('protocol','equalto',item.protocol) |

list | length == item.next\_hop | length

loop: "{{ route\_validation[inventory\_hostname] }}"

**How it works..**

NAPALM provide another Ansible module, napalm\_ping, that connects to the remote managed device and execute pings from the remote managed device towards a destination that we specify. Using this module we are able to validate the forwarding path between the managed devices and the specified destination.

In our example we create a new playbook and we specify the destination that we want to ping along with the maximum delay for our ping packets within the playbook itself using the **vars** parameter. Then we use the napalm\_ping module to connect to the MPLS PE devices in our topology to execute Ping from all these PE nodes towards the destination that we specified (in our case this is the loopback for our RR router). We store all this data in a variable called **rr\_ping**.

Below is a snippet of the output returned from napalm\_ping

"ping\_results": {

"success": {

"packet\_loss": 0,

"probes\_sent": 2,

"results": [

{

"ip\_address": "10.100.1.254",

"rtt": 2.808

},

{

"ip\_address": "10.100.1.254",

"rtt": 1.91

}

],

"rtt\_avg": 2.359,

"rtt\_max": 2.808,

"rtt\_min": 1.91,

"rtt\_stddev": 0.449

}

}

Finally, we use the assert module to validate and compare the results returned by NAPALM against the our requirements ( ping is successful, no packet loss and delay less than max\_delay)

**See Also..**

For more information and the other parameters supported by napalm\_ping please check the below URL

<https://napalm.readthedocs.io/en/latest/integrations/ansible/modules/napalm_ping/>

**Validating Network reachability using NAPALM**

In this recipe, we will outline how to utilize NAPALM and its Ansible modules to validate network reachability across the network. This validation performs ping from the managed devices to the destination that we specify in order to make sure that forwarding path across the network is working as expected.

**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..**

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

$ cat cat pb\_napalm\_ping.yml

---

- name: " Validation Traffic Forwarding with NAPALM"

hosts: pe

vars:

rr: 10.100.1.254

max\_delay: 5 # This is 5 msec

tasks:

- name: "P1T1: Ping Remote Destination using NAPALM"

napalm\_ping:

hostname: "{{ ansible\_host }}"

username: "{{ ansible\_user }}"

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

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

destination: "{{ rr }}"

count: 2

register: rr\_ping

* Update the playbook with the validation tasks as shown below

$ cat pb\_napalm\_ping.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: Validate Packet Loss is Zero and No Delay

assert:

that:

- rr\_ping.ping\_results.keys() | list | first == 'success'

- rr\_ping.ping\_results['success'].packet\_loss == 0

- rr\_ping.ping\_results['success'].rtt\_avg < max\_delay

**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;

}

}

**Network Validation with NAPALM**

In this recipe we will outline how we can validate the operational state of the network by defining the intended state of the network and letting NAPALM validate that the actual/operational state of the network matches our intended state. This is useful in network auditing and compliance reports for our network infrastructure.

**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..**

* Create a new folder called napalm\_validate and create a YAML file for each device we want to validate its state as shown below.

$ cat napalm\_validate/mxpe01.yml  
  
---  
- get\_interfaces\_ip:

ge-0/0/0.0:

ipv4:

10.1.1.3:

prefix\_length: 31

- get\_bgp\_neighbors:

global:

router\_id: 10.100.1.1

* Create a new playbook pb\_napalm\_validation.yml with the below contents

$ cat pb\_napalm\_validation.yml  
  
---

- name: " Validating Network State via NAPALM"

hosts: junos:&pe

tasks:  
 - name: "P1T1: Validation with NAPALM"

napalm\_validate:

hostname: "{{ ansible\_host }}"

username: "{{ ansible\_user }}"

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

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

validation\_file: "napalm\_validate/{{ inventory\_hostname}}.yml"

ignore\_errors: true

register: net\_validate

* Update the playbook to create folder to store compliance reports for each device as shown below

$ cat pb\_napalm\_validation.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: Create Compliance Report Folder

file: path=compliance\_folder state=directory

- name: Create Compliance Report File per Node

file: path=compliance\_folder/{{inventory\_hostname}}.txt state=absent

- name: Create Compliance Report

copy:

content: "{{ net\_validate.compliance\_report | to\_nice\_yaml }}"

dest: "compliance\_folder/{{ inventory\_hostname }}.txt"

**How it works..**

NAPALM provides another module for network validation which is the napalm\_validate module. This module is mainly used to perform auditing and compliance reports for the network infrastructure and the main goal is to provide an intended state for the network defined and declared in a YAML document formatted in a specific format following the same structure that the different NAPALM getters uses. In this YAML file we specify the NAPALM facts that we want to retrieve from the network and the expected output that we expect the network state to be like.

We supply these validation files to napalm\_validate module and NAPALM will connect to the devices, retrieve the facts specified in these validation files and compare the output retrieved from the network against the network stated declared in these validation files. Finally, it generates a compliance\_report object which has the result of the comparison and whether the network comply with these validation files or not. We also set the **ignore\_errors** parameter in order to continue with the other tasks in this playbook in case the device doesn’t comply, so we can capture this compliance problem in the compliance report that we will generate.

Finally, we save these output in a separate folder called compliance\_folder for each node and we copy the contents of the compliance\_report parameter and we formate it using to\_nice\_yaml filter.

A snippet for a correct compliance report generated for mxpe01 device is shown below

complies: true

get\_bgp\_neighbors:

complies: true

extra: []

missing: []

present:

global:

complies: true

nested: true

get\_interfaces\_ip:

complies: true

extra: []

missing: []

present:

ge-0/0/0.0:

complies: true

nested: true

skipped: []

**See Also..**

For Further information regarding NAPALM validation and the other options available for napalm\_validation please check the below URLs

<https://napalm.readthedocs.io/en/latest/integrations/ansible/modules/napalm_validate/index.html>  
<https://napalm.readthedocs.io/en/latest/validate/index.html>