Network Automation Cookbook

Proven and actionable recipes to automate and manage network devices using Ansible

Karim Okasha

Packt
www.packt.com

Network Automation Cookbook

Proven and actionable recipes to automate and manage network devices using Ansible

Karim Okasha

BIRMINGHAM - MUMBAI

Network Automation Cookbook

Commissioning Editor: Vijin Boricha
Acquisition Editor: Rohit Rajkumar
Content Development Editor: Ronn Kurien
Senior Editor: Richard Brookes-Bland
Technical Editor: Dinesh Pawar
Copy Editor: Safis Editing
Project Coordinator: Neil Dmello
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Nilesh Mohite

First published: April 2020

Production reference: 1170420

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78995-648-1

www.packt.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Karim Okasha is a network consultant with over 15 years of experience in the ICT industry. He is specialized in the design and operation of large telecom and service provider networks and has lots of experience in network automation. Karim has a bachelor's degree in telecommunications and holds several expert-level certifications, such as CCIE, JNCIE, and RHCE. He is currently working in Red Hat as a network automation consultant, helping large telecom and service providers to design and implement innovative network automation solutions. Prior to joining Red Hat, he worked for Saudi Telecom Company as well as Cisco and Orange S.A.

I would like to thank my wife and kids for providing me with the freedom and understanding needed to focus on this dream; without their support, this book wouldn't be possible.

I would like to thank the Packt Publishing team and my technical reviewers, for making my dream of writing this book a reality.

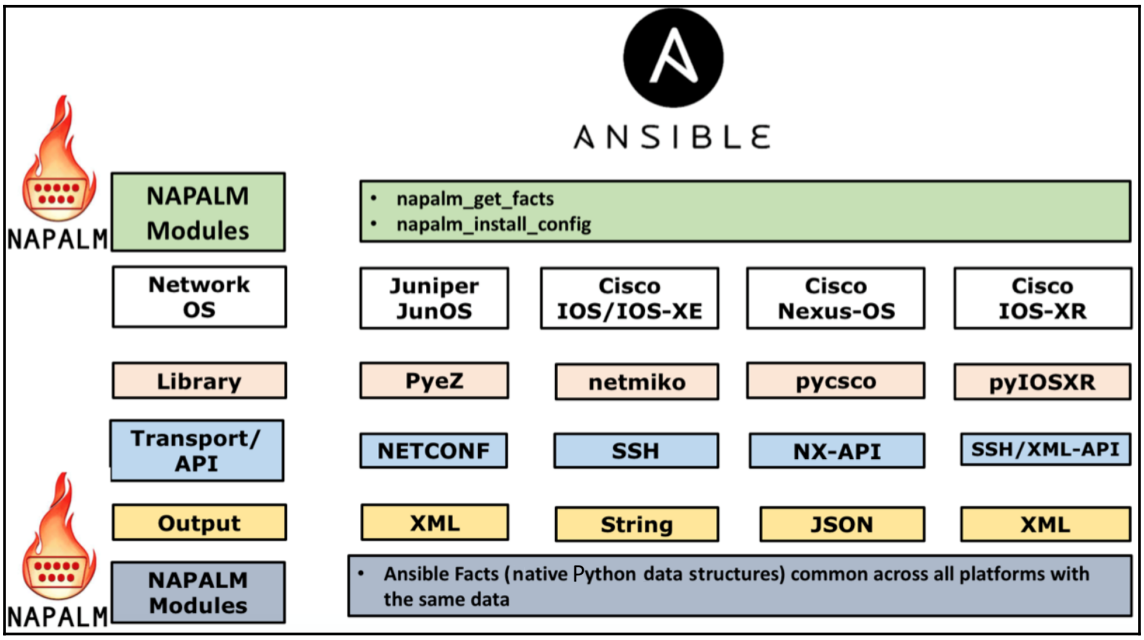
Finally, I would like to thank my mentor and best friend, Mohammed Mahmoud, for all his support and encouragement during all these years.

6
Administering a Multi-Vendor Network with NAPALM and Ansible

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

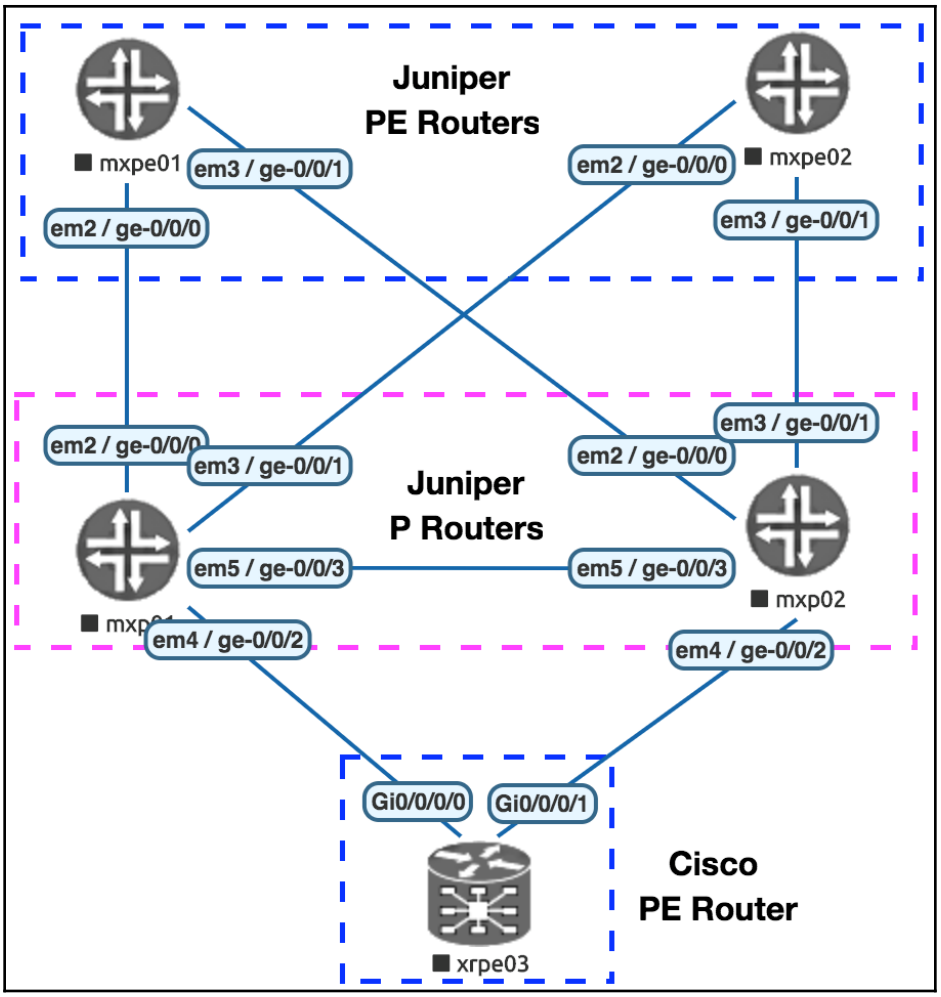
In previous chapters, we have seen how to interact with different network devices using Ansible. However, for each vendor OS, we had to use a different Ansible module to support that specific OS. Furthermore, we saw that the data returned from each vendor OS is completely different. Although writing a playbook for multi-vendor devices is still possible, it requires the use of multiple different modules, and we need to work with the different data structures returned by these devices. This is the main issue that NAPALM tries to address. NAPALM attempts to provide an abstracted and consistent API to interact with multiple vendor OSes, while the data returned by NAPALM from these different vendor OSes is normalized and consistent.

NAPALM interacts with each device according to the most common API supported by this node, and the API that is widely adopted by the community. The following diagram outlines how NAPALM interacts with the most common network devices, as well as the libraries used in NAPALM to interact with the APIs on these devices:



Since NAPALM tries to provide a consistent method to interact with network equipment, it supports a specific set of vendor devices. NAPALM also supports the most common tasks that are carried out on these devices, such as device configuration, retrieving the operational state for interfaces, **Border Gate Protocol** (**BGP**) and **Link Layer Discovery Protocol** (**LLDP**), and many others. For more information regarding the supported devices, as well as the supported methods when interacting with these devices, please check the following link: `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 will outline how to manage the configuration of these different vendor OSes, as well as how to retrieve the operational state from these devices. We will base our illustration on the following sample network diagram of a basic service provider network:



The following table outlines the devices in our sample topology and their respective management **Internet Protocols** (**IPs**):

Device	Role	Vendor	Management (MGMT) Port	MGMT IP
`mxp01`	P Router	Juniper vMX 14.1	`fxp0`	`172.20.1.2`
`mxp02`	P Router	Juniper vMX 14.1	`fxp0`	`172.20.1.3`
`mxpe01`	PE Router	Juniper vMX 14.1	`fxp0`	`172.20.1.4`
`mxpe01`	PE Router	Juniper vMX 17.1	`fxp0`	`172.20.1.5`
`xrpe03`	PE Router	Cisco XRv 6.1.2	`Mgmt0/0/CPU0/0`	`172.20.1.6`

The main recipes covered in this chapter are shown in the following list:

- Installing NAPALM and integrating with Ansible
- Building an Ansible network inventory
- Connecting and authenticating to network devices using Ansible
- Building the device configuration
- Deploying configuration on network devices using NAPALM
- Collecting device facts with NAPALM
- Validating network reachability using NAPALM
- Validating and auditing networks with NAPALM

Technical requirements

The code files for this chapter can be found here: `https://github.com/PacktPublishing/Network-Automation-Cookbook/tree/master/ch6_napalm`.

The following software will be required in this chapter:

- Ansible machine running CentOS 7
- Ansible 2.9
- Juniper **Virtual MX** (**vMX**) router running Junos OS 14.1R8 and Junos OS 17.1R1 release
- Cisco XRv router running IOS XR 6.1.2

Check out the following video to see the Code in Action:
`https://bit.ly/2Veox8j`

Installing NAPALM and integrating with Ansible

In this recipe, we outline how to install NAPALM and integrate it to work with Ansible. This task is mandatory since NAPALM Ansible modules are not part of the core modules that are shipped with Ansible by default. So, in order to start working with these modules, we need to install NAPALM and all of its Ansible modules. Then, we need to inform Ansible of where to find it and start working with the specific modules developed by the NAPALM team for Ansible.

Getting ready

Ansible and Python 3 need to be installed on the machine, along with the `python3-pip` package, which we will use to install NAPALM.

How to do it...

1. Install the `napalm-ansible` Python package, as shown in the following code snippet:

```
$ pip3 install napalm-ansible
```

2. Run the `napalm-ansible` command, as shown in the following code block:

```
$ napalm-ansible
```

3. 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/python3.6/site-
packages/napalm_ansible/modules
 action_plugins = /usr/local/lib/python3.6/site-
packages/napalm_ansible/plugins/action
```

For more details on Ansible's configuration file, visit `https://docs.ansible.com/ansible/latest/intro_configuration.html`.

4. Create a new folder called `ch6_napalm` and create the `ansible.cfg` file, updating it as shown in the following code block:

```
$ cat ansible.cfg
[defaults]
inventory=hosts
retry_files_enabled=False
gathering=explicit
host_key_checking=False
library = /usr/local/lib/python3.6/site-
packages/napalm_ansible/modules
action_plugins = /usr/local/lib/python3.6/site-
packages/napalm_ansible/plugins/action
```

How it works…

Since the NAPALM package and corresponding NAPALM Ansible modules are not part of the core modules shipped and installed by default with Ansible, we need to install it on the system in order to start working with the NAPALM Ansible modules. The NAPALM team has shipped a specific Python package to install NAPALM along with all the Ansible modules and all the dependencies, in order to start working with NAPALM from inside Ansible. This package is `napalm-ansible`. We will use the `pip3` program to install this package since we are using Python 3.

In order to tell Ansible where the Ansible module is installed, we need to enter the path for these modules into Ansible. The NAPALM team also provides 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. We execute the `napalm-ansible` command, which outputs the required configuration that we need to include in the `ansible.cfg` file so that Ansible can find the NAPALM modules that we will be using.

We update the `ansible.cfg` file with the output that we obtained from the `napalm-ansible` command. We then update the library and action plugin options, which tell Ansible to include these folders 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 an Ansible network inventory

In this recipe, we will outline how to build and structure our Ansible inventory to describe our sample service provider 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. In the case of NAPALM, we need to sort the different nodes in our network into the correct vendor types supported by NAPALM.

How to do it…

1. Inside the new folder (`ch6_napalm`), we create a `hosts` file with the following 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 follows:

- We created the `PE` group, which references all the **Multiprotocol Label Switching** (**MPLS**) **Provider Edge** (**PE**) nodes in our topology.
- We created the `P` group, which references all the MPLS **Provider** (**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 vendor or per OS is a best practice when working with NAPALM since we use these groups to specify the required parameters needed by NAPALM to identify the vendor of the remotely managed node, and how to establish network connectivity with this remote node. In the next recipe, we will outline how we will employ these groups (`junos` and `iosxr`), and which parameters we will include in order for NAPALM to establish a connection to the remotely managed nodes.

Connecting and authenticating to network devices using Ansible

In this recipe, we will outline how to connect to both Juniper and IOS-XR nodes using Ansible, 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…

1. On the Juniper devices, configure the username and password, as shown in the following code block:

```
system {
  login {
    user ansible {
      class super-user;
      authentication {
        encrypted-password "$1$mR940Z9C$ipX9sLKTRDeljQXvWFfJm1"; ##
ansible123
      }
    }
  }
}
```

2. On the Cisco IOS-XR devices, configure the username and password, as shown in the following code block:

```
!
  username ansible
  group root-system
  password 7 14161C180506262E757A60 # ansible123
!
```

3. Enable the **Network Configuration Protocol** (**NETCONF**) on the Juniper devices, as follows:

```
system {
  services {
    netconf {
      ssh {
        port 830;
      }
    }
  }
}
```

4. On the IOS-XR devices, we need to enable **Secure Shell** (**SSH**), as well as enable `xml-agent`, as follows:

```
!
xml agent tty
iteration off
!
xml agent
!
ssh server v2
ssh server vrf default
```

5. On the Ansible machine, create the `group_vars` directory in the `ch6_napalm` folder, and create the `junos.yml` and `iosxr.yml` files, as shown in the following code block:

```
$ cat group_vars/iosxr.yml
---
 ansible_network_os: junos
 ansible_connection: netconf

 $ cat group_vars/junos.yml
---
 ansible_network_os: iosxr
 ansible_connection: network_cli
```

6. Under the `group_vars` folder, create the `all.yml` file with the following login details:

```
$ cat group_vars/all.yml
ansible_user: ansible
 ansible_ssh_pass: ansible123
```

How it works…

NAPALM uses a specific transport API for each vendor equipment supported by NAPALM. It uses this API in order to connect to the device, so in our sample topology, we need NETCONF to be enabled on the Juniper devices. For Cisco IOS-XR devices, we need to enable SSH, as well as enabling the XML agent on the IOS-XR devices.

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 them ready for NAPALM to communicate with them.

Using the legacy `xml agent` on the IOS-XR devices in production is not recommended and needs to be evaluated as per the Cisco documentation. For further details, refer to `https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r5-3/sysman/command/reference/b-sysman-cr53xasr/b-sysman-cr53xasr_chapter_01010.html`.

On the Ansible machine, we set the `ansible_connection` parameter per each vendor (`netconf` for `juniper` and `network_cli` for `iosxr`), and we specify the `ansible_network_os` parameter to designate the vendor OS. All these parameters are defined under the `group_vars` hierarchy in `junos.yml` and `iosxr.yml`, corresponding to the groups that we defined in our inventory for grouping the devices on vendor OS basics. Finally, we specify the username and password via `ansible_user` and `ansible_ssh_pass` in the `all.yml` file, since we are using the same user to authenticate to both Juniper and Cisco devices.

To test and validate that, we can communicate with the devices from the Ansible control machine using the Ansible `ping` module, as shown in the following code block:

```
$ 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 the device configuration

NAPALM doesn't provide declarative modules to configure the various system parameters on the managed devices, such as interfaces' BGP, **Quality of Service** (**QoS**), and so on. However, it provides 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 for all our devices. This is the configuration that we will push to our devices using NAPALM, in the next recipe.

Getting ready

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

How to do it…

1. Create a `roles` folder, and inside this folder, create a new role called `build_router_config`, as follows:

```
$ mkdir roles && mkdir roles/build_router_config
```

2. Use the exact same contents (Jinja2 templates and tasks) for the `build_router_config` role that we developed for Juniper devices in `Chapter 3`, *Automating Juniper Devices in the Service Providers Using Ansible*, to generate the configuration for the devices. The directory layout should be as shown in the following code block:

```
$ tree roles/build_router_config/

roles/build_router_config/
  ├── tasks
  │   ├── build_config_dir.yml
  │   ├── build_device_config.yml
```

```
|   └──── main.yml
└──── templates
└──── junos
├──── bgp.j2
├──── intf.j2
├──── mgmt.j2
├──── mpls.j2
└──── ospf.j2
```

3. Create a new folder called `iosxr` under the `templates` folder and populate it with the Jinja2 templates for the different IOS-XR configuration sections, as shown in the following code block:

```
$ tree roles/build_router_config/templates/iosxr/

roles/build_router_config/templates/iosxr/
  ├──── bgp.j2
  ├──── intf.j2
  ├──── mgmt.j2
  ├──── mpls.j2
  └──── ospf.j2
```

4. Update the `group_vars/all.yml` file with the required data to describe our network topology, as shown in the following code block:

```
$ 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
```

5. Create a specific directory for each host in the `host_vars` directory, and in each directory, create the `bgp.yml` file with the following BGP peering content:

```
$ cat host_vars/xrpe03/bgp.yml
bgp_asn: 65400
bgp_peers:
  - local_as: 65400
    peer: 10.100.1.254
    remote_as: 65400
```

6. Create a new playbook called `pb_napalm_net_build.yml` that utilizes the `build_router_config` role in order to generate the device configuration, as shown in the following code block:

```
$ cat pb_napalm_net_build.yml
---
- name: " Generate and Deploy Configuration on All Devices"
  hosts: sp_core
  tasks:
    - name: Build Device Configuration
      import_role:
        name: build_router_config
      delegate_to: localhost
      tags: build
```

How it works…

In this recipe, our main goal is to create the device configuration that we will deploy on the devices in our sample topology. We are using the same Ansible role that we used to generate the configuration for Juniper devices in `Chapter 3`, *Automating Juniper Devices in the Service Providers Using Ansible*. The only addition to this role is that we are adding the required Jinja2 templates for IOS XR.

Here is a quick explanation of the steps, as a quick review:

- Modeling the network via Ansible variables

  We describe the different aspects of our network topology, such as **Peer-to-Peer** (**P2P**) interface, loopback interfaces, and **Open Shortest Path First** (**OSPF**) parameters under different data structures in the `group_vars/all.yml` file. For any host-specific data, we use the `host_vars` directory to populate all variables/parameters that are specific to a specific node, and, in our case, we use this approach for BGP data to outline `bgp_peers` variable for each node. This provides us with all the required data to populate the Jinja2 templates needed to generate the final configuration for each device in our sample network.

- Building the Jinja2 templates

  We place all our Jinja2 templates in the `templates` folder inside our role, and we segment our Jinja2 templates per the vendor OS, each in a separate folder. Next, we create a Jinja2 template for each section of the configuration. The following code snippet outlines the directory structure for the templates folder:

```
templates/
  ├── iosxr
  │   ├── bgp.j2
  │   ├── intf.j2
  │   ├── mgmt.j2
  │   ├── mpls.j2
  │   └── ospf.j2
  └── junos
  ├── bgp.j2
  ├── intf.j2
  ├── ospf.j2
  ├── mgmt.j2
  └── mpls.j2
```

For a detailed explanation of the different Jinja2 templates used in this recipe and how they use the defined Ansible variables to generate the final configuration, please refer to `Chapter 3` of this book, *Automating Juniper Devices in the Service Providers Using Ansible*, since we are using the exact same network topology and the same data structures for both JunOS and IOS-XR devices.

Running this playbook will generate the configuration for all the devices in our Ansible inventory in the `configs` folder, as shown in the following code block:

```
$ tree configs/
configs/
  ├── mxp01.cfg
  ├── mxp02.cfg
  ├── mxpe01.cfg
  ├── mxpe02.cfg
  └── xrpe03.cfg
```

Deploying configuration on network devices using NAPALM

In this recipe, we will outline how to push configurations on different vendor devices using Ansible and NAPALM. NAPALM provides a single Ansible module for configuration management, and this module allows us to use a single common method to push any configuration on any vendor equipment supported by NAPALM, greatly simplifying Ansible playbooks.

Getting ready

To follow along with this recipe, you will need to have an Ansible inventory already set up, with network reachability between the Ansible controller and the network devices established. The configuration that we will be pushing to the devices is the one we generated in the previous recipe.

How to do it…

1. Update the `pb_napalm_net_build.yml` playbook file, and add the tasks shown in the following code block:

```
$ 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 configurations to the network devices. It requires the configuration to be present in a text file. When it connects to the network device, it pushes 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. We also need to provide the other parameters, such as username/password, to log in and authenticate with the device.

The `napalm_install_config` module requires the following mandatory parameters in order to correctly log in 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`: This is the username and password to connect to the device. We need to supply the `ansible_user` and `ansible_ssh_pass` attributes.
- `dev_os`: This parameter provides the vendor OS name that NAPALM requires in order to choose the correct API and the correct library to communicate with the device. For this option, we provide the `ansible_network_os` parameter.
- The `napalm_install_config` module uses the following parameters to manage the configuration on remote devices:
    - `config_file`: This provides the path of the configuration file containing the device configuration that needs to be pushed to the managed device.

- `commit_changes`: This tells the device whether or not to commit the configuration. NAPALM provides a consistent method for configuration commits, even for devices that don't support it by default (for instance, Cisco IOS devices).
- `replace_config`: This parameter controls how to merge between the existing configuration on the device and the configuration in the `config_file` parameter. 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 with the configuration that we generate. This will cause 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 `deploy` tag, NAPALM will connect to the device and push the configuration. However, it will not commit the configuration on the remote device, since we have specified 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 in the following code snippet:

```
$ ansible-playbook pb_napalm_net_build.yml --tags deploy --e commit=yes
```

There's more...

The `napalm_install_config` module provides extra options to control how to manage the configuration on the remote devices, such as the configuration diff. With this option, we can collect the differences between the running configuration on the device and the configuration that we will push via NAPALM. This option can be enabled as follows:

- Create a folder called `config_diff` to store the configuration diff captured by NAPALM, as shown in the following code block:

```
$ cat group_vars/all.yml

< -- Output Omitted for brevity -->
config_diff_dir: ./config_diff

$ cat tasks/build_config_dir.yml

- name: "Create Config Diff Directory"
  file: path={{config_diff_dir}} state=directory
  run_once: yes
```

- Update the `pb_napalm_net_build.yml` playbook, as shown in the following code block:

```
$ cat pb_napalm_net_build.yml

---
- name: "Generate and Deploy Configuration on All Devices"
  hosts: sp_core
  tasks:

< -- Output Omitted for brevity -->

    - name: "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
```

Next, we create a new folder to house all the configuration diff files that we will generate for each device, and add the `diff_file` parameter to the `napalm_install_config` module. This will collect the configuration 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 are generated, as shown in the following code block:

```
$ 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 the operational state from network devices using the NAPALM fact-gathering Ansible module. This can be used to validate the network state across multi-vendor equipment since NAPALM Ansible's fact-gathering module returns a consistent data structure across all vendor OSes supported by NAPALM.

Getting ready

To follow along with this recipe, it is assumed that an Ansible inventory is already in place and network reachability between the Ansible controller and the network is already established. Finally, the network is configured as per the previous recipe.

How to do it…

1. Create an Ansible playbook named `pb_napalm_get_facts.yml` with the following content:

```
$ 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
```

2. Update the playbook with the following tasks to validate the data returned by the 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 Sessions 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 `napalm_get_facts` Ansible module 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. In this example, we are limiting the data retrieved to `bgp_neighbors`.

The `napalm_get_facts` module returns the data retrieved from the nodes as Ansible facts. This data can be retrieved from the `napalm_bgp_neighbors` variable, which stores all the NAPALM BGP facts retrieved from the device.

The following snippet outlines the output from `napalm_bgp_neighbors`, retrieved from a Junos OS device:

```
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 outlines the output from `napalm_bgp_neighbors`, retrieved from an IOS-XR device:

```
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 vendors is consistent between different network vendors. This simplifies parsing this data and allows us to run much simpler playbooks to validate the network state.

We use the data returned by NAPALM to compare and validate the operational state of the network against our network design, which we defined using Ansible variables such as `bgp_peers`. We use the `assert` module to validate multiple BGP information, such as the following:

- Correct number of BGP peers
- BGP router ID
- All BGP sessions are operational

We use the `when` statement in the different `assert` modules in scenarios in which we have a router in our topology that doesn't run BGP (`mxp02` is an example). Consequently, 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 supported and the level of facts supported by 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 the `napalm_get_facts` module, please check the following 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 following URL:
`https://napalm.readthedocs.io/en/latest/support/`.

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 pings from the managed devices to the destination that we specify, in order to make sure that the forwarding path across the network is working as expected.

Getting ready

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

How to do it…

1. Create a new playbook called `pb_napalm_ping.yml` with the following content:

```
$ cat pb_napalm_ping.yml

---
- name: " Validation Traffic Forwarding with NAPALM"
  hosts: junos:&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
```

2. Update the playbook with the validation tasks shown in the following code block:

```
$ 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…

NAPALM provides another Ansible module, `napalm_ping`, which connects to the remote managed device and executes pings from the remote managed device toward a destination that we specify. Using this module, we are able to validate the forwarding path between the managed devices and the specified destination.

This `napalm_ping` module does not currently support Cisco IOS-XR devices, which is why we only select all PE devices that are in the Junos OS group. In our playbook, we use the `junos:&pe` pattern in order to do this.

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 (only Junos OS ones) in our topology to execute `ping` from all these PE nodes toward the destination that we specified (in our case, this is the loopback for our **route reflector** (**RR**) router). We store all this data in a variable called `rr_ping`.

The following snippet shows 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 our requirements (ping is successful, no packet loss, and delay less than `max_delay`).

Validating and auditing networks 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, it is assumed that an Ansible inventory is already in place and network reachability between the Ansible controller and the network is established. Finally, the network is configured as per the previously outlined recipe.

How to do it…

1. Create a new folder called `napalm_validate` and create a YAML file for each device. We will validate its state, as shown in the following code block:

```
$ 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
```

2. Create a new `pb_napalm_validation.yml` playbook with the following content:

```
$ cat pb_napalm_validation.yml

---
- name: " Validating Network State via NAPALM"
  hosts: 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
```

3. Update the playbook to create a folder that will store the compliance reports for each device, as shown in the following code block:

```
$ cat pb_napalm_validation.yml

< -- Output Omitted for brevity -->

- name: Create Compliance Report Folder
  file: path=compliance_folder state=directory

- name: Clean Last Compliance Report
  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 generate compliance reports for the network infrastructure. The main idea is to declare the intended state of the network and define it in a YAML document. This YAML file has a specific format, following the same structure with which the different NAPALM facts are generated. In this YAML file, we specify the NAPALM facts that we want to retrieve from the network, along with the network's expected output.

We supply these validation files to the `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 state declared in these validation files.

Next, NAPALM generates a `compliance_report` object, which has the result of the comparison and whether the network complies 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 the output in a separate folder called `compliance_folder` for each node, copy the contents of the `compliance_report` parameter, and format it using the `to_nice_yaml` filter.

The code for a correct compliance report generated for a `mxpe01` device is shown in the following snippet:

```
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 on validating deployments and the other options available for `napalm_validate`, please check the following URLs:

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