For a long time, SNMP has played a crucial role in the monitoring of networks and IT infrastructure. Despite the introduction of streaming telemetry, SNMP continues to have an important role. For brand-new network equipment, streaming telemetry based on gRPC/gNMI is a much better solution. However, as a relatively new management protocol, gRPC/gNMI operates only in reasonably new software, and many legacy devices still require monitoring and rely heavily on SNMP.

SNMP was created to standardize information structures across different vendors. For example, SNMP has a standard ISO MIB tree, which is the same across all the vendors implementing it. This is quite convenient because it means you can collect the information in the same format (counter names, sizing, and so on) and using the same SNMP OID from your Cisco, Juniper, Nokia, and Cumulus devices. When this approach was created, it was a breakthrough. Even today, in the era of open-source products and collaboration, fully standardized YANG models (for example, IETF, OpenConfig) are not yet implemented across all major vendors.

Besides offering monitoring capabilities, SNMP also provides an opportunity to manage network elements. Actually, it represents the first attempt to make a network element programmable by just conveying the value of the specific parameter to a particular SNMP OID. If you think about it abstractly (not looking into a particular realization), that is what is happening today with NETCONF/RESTCONF, only via different channels.

It's essential to acknowledge that the configuration capabilities of SNMP are minimal compared to those of the CLI. In addition, there is no feature parity in terms of configuration available via the CLI and via SNMP, and the CLI is an absolute winner.

Programmability via the Linux Shell

Linux is the number-one operating system for building highly available infrastructure for the most demanding and resource-intensive applications around the world. For a long time, it was an operating system for servers; that's why Linux was not a subject of discussion in the networking arena, although each Linux distribution has a complete network stack, including routing and firewalling. Each of the most popular distributions, including Red Hat Enterprise Linux/CentOS, Ubuntu, and Debian, has a CLI, commonly referred to as the Linux shell. All the distributions are capable of running shell scripts natively, so Linux provides programmability out-of-the-box. When using a shell, you can define variables, use cycles and conditions, and perform other activities related to programming. Linux and scripting using Bash are covered extensively in Chapters 2, "Linux Fundamentals," 3, "Linux Networking and Security," and 4, "Linux Scripting." The only point that may be added here is that the Linux shell has proven to be a more programmable interface than the regular router/switch CLI, as discussed in Chapter 4.

Programmability via NETCONF

NETCONF operates on XML-wrapped data sent over SSH. For some time, the XML API was the top choice for communication between the different components of distributed applications.

In general, if you take a broader look at the IT/network ecosystem, a network device can be treated equally to any other part of the infrastructure supporting distributed applications, such as an HTTP server or a database service. Approaches and protocols to manage the network functions could be the same as between application components. This is exactly what is happening: All the protocols that were good for communication between traditional IT applications are coming to the network world. NETCONF (which is XML based) and RESTCONF (which is JSON/XML based) are perfect examples of this.

NETCONF (which is discussed in detail in Chapter 14, "NETCONF and RESTCONF") is one of the newer protocols in the network programmability arena. NETCONF has overcome the limitations of SNMP in terms of feature parity with the CLI. Depending on the vendor, the feature parity between the CLI and NETCONF could be as high as 100%. A significant factor in this success is YANG, which is covered in Chapter 13, "YANG." In the NETCONF/YANG framework, a network device is represented as a set of parameters with specific values that can be managed in individual or interdependent mode. NETCONF was the first protocol to use YANG, and so today, programmability in the network world is sometime equated to NETCONF/YANG.

Programmability via RESTCONF and REST APIs

RESTCONF is another major programmability-related protocol. RESTCONF is a RESTful API. Today, REST APIs are a de facto standard for communication between various applications, and almost all modern applications either support them or plan to. For instance, management of Docker, the infrastructure for containers, is realized over REST APIs. As another example, InfluxDB, one of the market-leading time-series databases (a crucial component for streaming telemetry), is also managed using REST APIs; InfluxDB communicates with other components of the telemetry stack, such as a telemetry collector or visualization dashboard over REST APIs. In the network world, it is possible to manage Cisco NSO or DNA Center using REST APIs on the northbound interface.

The popularity of REST APIs in general (and RESTCONF as a specific use case) is due to the fact that it is simple, works fast, and has CRUD (create, read, update, delete) capabilities to work with the data. REST APIs (and RESTCONF) are realized over HTTP transport, typically protected by TLS. Therefore, RESTCONF has a different application layer protocol (SSH) than NETCONF (HTTP). (RESTCONF is covered in detail in Chapter 14, and HTTP, REST, and TLS are covered in Chapter 7, "HTTP and REST," and Chapter 8, "Advanced HTTP.")

RESTCONF and NETCONF are similar in their use of YANG data models. In reality, a particular network element has a single YANG data model, which defines the data structure and the interdependencies of the parameters. Given that a network device may supports both NETCONF and RESTCONF interfaces, it doesn't matter how the data is transmitted: Transmission can occur over SSH/XML (NETCONF) or HTTPS/JSON/XML (RESTCONF).

However, to complete the picture, it should be noted that RESTCONF/YANG support is still in an emerging stage as of this writing. Across non-Cisco platforms, RESTCONF support is not common. For instance, Juniper Junos OS and Arista EOS support RESTCONF, and Nokia does not. Nevertheless, the general trend is toward adapting advanced and useful application concepts, and we will see more RESTCONF implementations in the next few years.

Programmability via gRPC/gNMI

Among all the protocols involved in working with YANG data models, gNMI is the most recently developed one. gNMI stands for gRPC Network Management Interface, and, as the name indicates, gNMI uses gRPC as the protocol for remote procedure calls based on HTTP/2 transport to communicate between the server and the network devices. Like RESTCONF, gNMI also has CRUD capabilities. Despite its novelty, gNMI has an extremely strong following and support in the community, and it is becoming adopted widely. Google is its main supporter and is also leading the development of both gNMI and OpenConfig YANG data models with two essential requirements in mind: low latency and scalability. As you can imagine, Google applies these two requirements to all its applications.

gNMI has a big advantage over NETCONF and RESTCONF in that its low-latency requirement and the way the data is packed for transmission make it very useful for streaming telemetry, where there is much information to be sent from each network function to the telemetry collector for processing. (See Chapter 15, "gRPC, Protobuf, and gNMI," for details.) All the major vendors, including Cisco, Juniper, and Nokia, have implemented gNMI (and therefore gRPC) as a primary protocol for streaming telemetry. There are other options available, such as TCP or NETCONF telemetry streaming, but they requires far more compute resources on the network devices to generate messages or produce more administrative traffic overhead due to inefficient serialization. Therefore, gNMI is actively used for telemetry. Due to its use for telemetry collection and network management, gNMI is becoming very popular.

gNMI adoption for the management of the network devices is expected to continue to increase. Having several management protocols—such as NETCONF, RESTCONF, and gNMI—running all together is not necessarily the best usage of a device's resources. Given that all CRUD functions can be realized using a single protocol, it might be that in future we see unification of all processes to gNMI or RESTCONF.

Implementation Examples

Now that you have read about the available APIs, this section guides you through some real implementations with vendors other than Cisco. Some of these implementations might seem quite familiar to you, and others may not. Nevertheless, the following sections will help you follow the development of programmability in the network world.

Converting the Traditional CLI to a Programmable One

The main workhorse in the process of turning the CLI into a programmable interface is the management host. It can be an SDN controller or any other host running some application that is capable of managing the network function. In Chapter 20, "Looking Ahead," you will learn how easily you can program virtually everything in your network, even with the CLI. Ansible enables you to implement programmability easily, but even without it, you can reach some level of programmability. Earlier in this book, you learned about Linux scripting and Python programming, and you can use that knowledge to start programming network elements.

Let's say that you need to create an interface on an Arista switch. Example 18-1 shows the syntax for doing so.

Note

The IP address used in this example is arbitrary, and you can use whatever IP address you need.

Example 18-1 *Creating the Loopback Interface in Arista EOS*

```
EOS1#show run int lo678

interface Loopback678

   ip address 192.168.192.168/32

!

end
```

In this example, it is important to understand the sequence of the commands you enter in order to get the interface up and running as well as the variables and the fixed text. The variables in Example 18-1 are the interface name (Loopback678 in this case) and the IP address (192.168.192.168/32). There may be another variable that you can't see in Example 18-1: the network device itself, where the interface is supposed to be configured. It can be either the hostname, if the management host knows how to reach it, or the IPv4/IPv6 address.

If you know the sequence of the commands and their syntax, you can create a simple Bash script that can create the interface for you. Example 18-2 shows an example of a possible script.

Example 18-2 *Bash Script for Creating the Interface in Arista EOS*

```
$ cat create_interface.sh

#!/bin/bash


INTERFACE_NAME=${1}

INTERFACE_IPV4=${2}

NETWORK_FUNCTION=${3}


echo "Interface with the name \"${INTERFACE_NAME}\" and IP \"${INTERFACE_IPV4}\" is to

be created on \"${NETWORK_FUNCTION}\""


ssh aaa@${NETWORK_FUNCTION} << EOF

enable

configure terminal

interface ${INTERFACE_NAME}

ip address ${INTERFACE_IPV4}

end

show ip interface ${INTERFACE_NAME}

EOF
```

Keep in mind that this script needs to be created on a management host rather than on the network element itself. The management host could be your laptop or any jump server in your network that can reach the target network function over SSH.

This script reads a sequence of three arguments, where the first one (**${1}**) contains the name of the interface, the second one (**${2}**) contains the IPv4 address of the interface, and the third one (**${3}**) contains the hostname of the management IP address of the target network device. In this specific case, the position of the arguments is important; however, you could extend the script by adding the specific identifier as a part of the argument to tell Bash about the argument type, regardless of its position.

When the scripts receives all the arguments, it prints them to STDOUT for your information and then connects to the network device from the third argument by using non-interactive SSH mode and passes the configuration lines contained between the tags **<<EOF** and **EOF** created using the first two arguments. Example 18-3 demonstrates the execution of the Bash script from Example 18-2.

Example 18-3 *Creating an Interface in Arista EOS Using the Bash Script from Example 18-2*

```
$ ./create_interface.sh Loopback123 172.16.0.0/32 EOS1
```

```
Interface with the name "Loopback123" and IP "172.16.0.0/32" is to be created on "EOS1"

Pseudo-terminal will not be allocated because stdin is not a terminal.

Password:

Loopback123 is up, line protocol is up (connected)

  Internet address is 172.16.0.0/32

  Broadcast address is 255.255.255.255

  IPv6 Interface Forwarding : None

  Proxy-ARP is disabled

  Local Proxy-ARP is disabled

  Gratuitous ARP is ignored

  IP MTU 65535 bytes
```

In Example 18-3, you can see that the interface is successfully created using the Bash script from Example 18-2. The last command in the script creates the output of the **show ip interface** command for the created interface.

As discussed earlier, you could add many enhancements to this script, such as the one already mentioned for the sequence of the arguments or a more sophisticated configuration using loops and conditions. Two factors are crucial for such automation:

- You must know how to configure network elements with the CLI.
- You must know how to create the scripts.

The latter is particularly important because even if a script is working with one of the network vendors, it may not work with another one. One of the reasons is that the shell script expects to get the response in a certain way. For example, it might wait for the Password: pattern in the output before it starts entering the password. In addition, it may expect other matches, such as a > or $, before it starts sending the commands in the script. You would need to create proper updates to your script for each managed vendor.

Now say that you want to turn the script from Example 18-2 into a multivendor one. If you know the CLI and the programming/scripting, you can easily do that. Example 18-4 shows how to extend the initial Bash script so that it's capable of configuring both Arista EOS and Cumulus Linux switches.

Example 18-4 *Adding Multivendor Capabilities to the Bash Script from Example 18-2*

```
$ cat create_interface.sh

#!/bin/bash


INTERFACE_NAME=${1}

INTERFACE_IPV4=${2}

NETWORK_FUNCTION=${3}

VENDOR=${4}


echo "Interface with the name \"${INTERFACE_NAME}\" and IP \"${INTERFACE_IPV4}\" is to

be created on \"${NETWORK_FUNCTION}\""


case ${VENDOR} in

arista*)

ssh aaa@${NETWORK_FUNCTION} << EOF

enable

configure terminal

interface ${INTERFACE_NAME}

ip address ${INTERFACE_IPV4}

end

show ip interface ${INTERFACE_NAME}

EOF

;;

cumulus*)

ssh cumulus@${NETWORK_FUNCTION} << EOF

net add interface ${INTERFACE_NAME} ip address ${INTERFACE_IPV4}

net commit

ip addr show dev ${INTERFACE_NAME}

EOF

;;

esac
```

You should already know the shell syntax used in this example. A fourth variable (**${4}**), which contains the vendor name, is used with conditional **case** syntax that defines the proper CLI commands for each vendor. (See Chapter 4 for more details on **case**.)

If you execute this script against a switch running Cumulus Linux, you see output similar to the output in Example 18-5.

Example 18-5 *Executing the Bash Script with Multivendor Capabilities*

```
$ ./create_interface.sh swp1 169.254.0.0/31 192.168.141.156 cumulus

Interface with the name "swp1" and IP "169.254.0.0/31" is to be created on

"192.168.141.156"

Pseudo-terminal will not be allocated because stdin is not a terminal.

cumulus@192.168.141.156's password:


Welcome to Cumulus VX (TM)


Cumulus VX (TM) is a community supported virtual appliance designed for

experiencing, testing and prototyping Cumulus Networks' latest technology.

For any questions or technical support, visit our community site at:

http://community.cumulusnetworks.com


The registered trademark Linux (R) is used pursuant to a sublicense from LMI,

the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide

basis.

--- /etc/network/interfaces   2019-06-10 13:24:38.928437459 +0000

+++ /run/nclu/ifupdown2/interfaces.tmp   2019-06-10 13:33:29.702733758 +0000

@@ -7,16 +7,17 @@

 auto lo

 iface lo inet loopback


 # The primary network interface

 auto eth0

 iface eth0 inet dhcp

     vrf mgmt


 auto swp1

 iface swp1

+    address 169.254.0.0/31


 auto mgmt

 iface mgmt

     address 127.0.0.1/8

     vrf-table auto





net add/del commands since the last "net commit"

==================================================


User     Timestamp                   Command

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

cumulus  2019-06-10 13:33:29.680811  net add interface swp1 ip address 169.254.0.0/31

3: swp1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group

default qlen 1000

    link/ether 00:50:56:32:44:29 brd ff:ff:ff:ff:ff:ff

    inet 169.254.0.0/31 scope global swp1

       valid_lft forever preferred_lft forever
```

```
    inet6 fe80::250:56ff:fe32:4429/64 scope link

       valid_lft forever preferred_lft forever
```

As these examples illustrate, multivendor programmability is not very difficult. These examples illustrate another essential concept: abstraction. These examples include the same set of parameters for both Arista and Cumulus switches, correctly deployed in each network operating system. In addition, you have seen the so-called adaptor level, which translates abstract parameters into the particular syntax of the target network operating system.

Implementing similar programmability with Python or Ansible would be much easier because these solutions include relevant libraries to manage connectivity to different vendors. However, this section helps you see that programmability can be accomplished even without those tools.

Classical Linux-Based Programmability

The next step in the programmability of the various network devices is the management of Linux if it's the core operating system. As mentioned at the beginning of the chapter, as networks are moving toward including white box switches, Linux is becoming a network operating system. For example, at a high level abstraction, Cumulus took the Debian Linux distribution and created a product that includes a unique CLI. In the vast majority of cases, networks rely on the open-source products, such as Debian Linux or IP routing protocol suite called FRR.

A bit earlier in this chapter, you saw how to create an interface in Cumulus Linux by using its CLI (refer to Example 18-5). Cumulus Linux is a Linux distro, so all the concepts and tools explained in Chapters 2 through 4 apply to it. This means that interface configuration using a *programmable* CLI can also be reproduced using the individual files. The interfaces configuration in Cumulus Linux (and Debian in general) is stored in the file /etc/network/interfaces and looks as shown Example 18-6.

Example 18-6 *File with the Interface Configuration in Linux*

```
cumulus@cumulus:mgmt-vrf:~$ cat /etc/network/interfaces

# This file describes the network interfaces available on your system

# and how to activate them. For more information, see interfaces(5).


source /etc/network/interfaces.d/*.intf


# The loopback network interface

auto lo

iface lo inet loopback


# The primary network interface

auto eth0

iface eth0 inet dhcp

    vrf mgmt


auto swp1

iface swp1

    address 169.254.0.0/31


auto mgmt

iface mgmt

    address 127.0.0.1/8

    vrf-table auto
```

By default, only the interfaces that have non-standard configurations are listed here because the Cumulus switch used in this example has more interfaces than are shown in the file /etc/network/interfaces. This can be verified by using standard Linux command shown in Example 18-7.

Example 18-7 *Output of the Interface Configuration in Linux*

```
cumulus@cumulus:mgmt-vrf:~$ ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group

default

    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master mgmt state

UP mode DEFAULT group default qlen 1000

    link/ether 00:50:56:30:a2:99 brd ff:ff:ff:ff:ff:ff

3: swp1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode

DEFAULT group default qlen 1000

    link/ether 00:50:56:32:44:29 brd ff:ff:ff:ff:ff:ff

4: swp2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group

default qlen 1000
```

```
    link/ether 00:50:56:2b:27:db brd ff:ff:ff:ff:ff:ff

5: swp3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group

default qlen 1000

    link/ether 00:50:56:3a:14:d8 brd ff:ff:ff:ff:ff:ff

6: swp4: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group

default qlen 1000

    link/ether 00:50:56:29:f8:1b brd ff:ff:ff:ff:ff:ff

7: swp5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group

default qlen 1000

    link/ether 00:50:56:33:1a:43 brd ff:ff:ff:ff:ff:ff

8: mgmt: <NOARP,MASTER,UP,LOWER_UP> mtu 65536 qdisc pfifo_fast state UP mode DEFAULT

group default qlen 1000

    link/ether b6:b7:20:53:a6:dd brd ff:ff:ff:ff:ff:ff
```

In Example 18-7, you can see the names of the ports available. All the ports called swp* are used for data plane forwarding. To add an IP address to any of these ports without using the CLI, you need to add the corresponding lines in the file /etc/network/interfaces and reapply the configuration by using the **ifreload -a** command. This sequence of actions forms a step-by-step procedure that you could implement directly on a network device. To do this, you would need to connect to it and use a Python or Bash script from your management host. Example 18-8 shows a Bash script that performs the file handling.

Example 18-8 *Bash Script for the Interface Configuration in Linux*

```
$ cat manage_interaces.sh

#!/bin/bash


INTERFACE_NAME=${1}

INTERFACE_IPV4=${2}

NETWORK_FUNCTION=${3}

VENDOR=${4}


echo "Interface with the name \"${INTERFACE_NAME}\" and IP \"${INTERFACE_IPV4}\" is to

be created on \"${NETWORK_FUNCTION}\""


case ${VENDOR} in

cumulus*)

ssh cumulus@${NETWORK_FUNCTION} "sudo chown cumulus:cumulus /etc/network/interfaces"

scp cumulus@${NETWORK_FUNCTION}:/etc/network/interfaces .

echo "auto ${INTERFACE_NAME}

iface ${INTERFACE_NAME}

    address ${INTERFACE_IPV4}" >> interfaces

scp interfaces cumulus@${NETWORK_FUNCTION}:/etc/network/interfaces

ssh cumulus@${NETWORK_FUNCTION} "sudo chown root:root /etc/network/interfaces;sudo

ifreload -a;ip addr show dev ${INTERFACE_NAME}"

;;

esac
```

The input parameters and general logic of the script in this example are the same as in Example 18-4. The difference between these examples is in the set of actions performed. This is what happens in Example 18-8:

1. The owner of the configuration file with the interface configuration is set to a value equal to the username, which is used for SSH access.

2. The configuration file is copied locally to the management host over SCP. This action allows you to get the existing set of interfaces and validate or analyze them if needed.

3. The configuration of the new interface is appended to this configuration file. This step ensures that the configuration file is accurate and complete.

4. The updated file is uploaded back to the network element so that the new validated set of interfaces is put on the device.

5. Once the file is uploaded, its owner is changed back to the root, the interfaces are reloaded, and the status of the changed interface is displayed.

Example 18-9 shows the execution of the Bash script from Example 18-8.

Example 18-9 *Programmable Management of the Linux Files*

```
$ ./manage_interaces.sh swp2 169.254.0.2/31 192.168.141.156 cumulus
```

```
Interface with the name "swp2" and IP "169.254.0.2/31" is to be created on

"192.168.141.156"

cumulus@192.168.141.156's password:

cumulus@192.168.141.156's password:

interfaces                                                    100%  437   602.4KB/s

00:00

cumulus@192.168.141.156's password:

interfaces                                                    100%  486    38.1KB/s

00:00

cumulus@192.168.141.156's password:

4: swp2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group

default qlen 1000

    link/ether 00:50:56:2b:27:db brd ff:ff:ff:ff:ff:ff

    inet 169.254.0.2/31 scope global swp2

       valid_lft forever preferred_lft forever

    inet6 fe80::250:56ff:fe2b:27db/64 scope link tentative

       valid_lft forever preferred_lft forever
```

The result shown in Example 18-9 is the desired result, as the interface configured here has the correct IP address. What is not desired—and this a major drawback of using a shell—is that you need to enter the password each time you interact with the remote host on SSH/SCP. Even with this simple script, you would have to enter the password four times.

The easiest way to avoid entering a password multiple times is to use Python or Ansible, which are able to store the credentials and use them when needed. However, if you still want to use Bash, you can use SSH with RSA keys instead of passwords. To implement it, you need to generate a pair of private/public key on your management host and send the public key to the target managed host so the management host can establish an SSH tunnel to the managed host without the password. Figure 18-2 illustrates the SSH architecture with RSA keys.

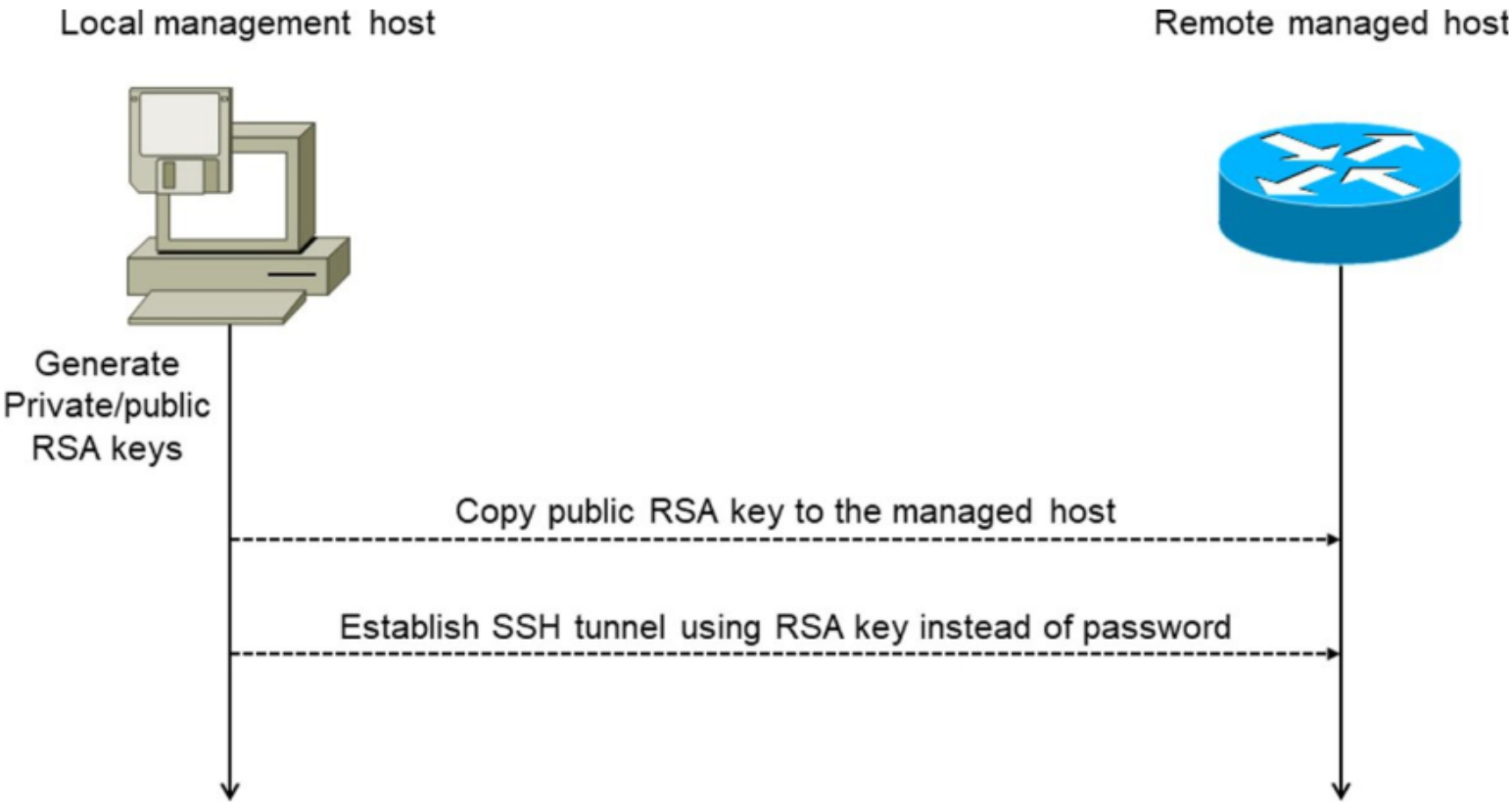

Figure 18-2 *SSH with RSA Keys*

Example 18-10 shows how you can deploy SSH access with RSA keys instead of a password.

Example 18-10 *Deploying SSH Access with RSA Keys Instead of a Password*

```
# Generate the SSH RSA Key Pair on the local management host

$ ssh-keygen

Generating public/private rsa key pair.

Enter file in which to save the key (/home/aaa/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/aaa/.ssh/id_rsa.

Your public key has been saved in /home/aaa/.ssh/id_rsa.pub.
```

```
The key fingerprint is:
SHA256:dspRSyuwkcqiRBLHgItQaabOHR0c7Wgdx9IhBlKT+HY aaa@sand7.karneliuk.com
The key's randomart image is:
+---[RSA 2048]----+
|++oo+==oo..      |
|oo=..+o=.+       |
|+*  o O + o      |
|*  o B E o o     |
|o.o * o S +      |
|.+ o   o =       |
|.       o        |
|                 |
|                 |
+----[SHA256]-----+

# Create the folder for SSH keys on the remote managed host
$ ssh cumulus@192.168.141.156 "mkdir /home/cumulus/.ssh/"

# Copy the public RSA key to the remote managed host
$ scp /home/aaa/.ssh/id_rsa.pub
cumulus@192.168.141.156:/home/cumulus/.ssh/authorized_keys
cumulus@192.168.141.156's password:
id_rsa.pub                                            100%  405   462.7KB/s
00:00

# Test the SSH access using the RSA keys without password to the remote managed host
$ ssh cumulus@192.168.141.156

Welcome to Cumulus VX (TM)

Cumulus VX (TM) is a community supported virtual appliance designed for
experiencing, testing and prototyping Cumulus Networks' latest technology.
For any questions or technical support, visit our community site at:
http://community.cumulusnetworks.com

The registered trademark Linux (R) is used pursuant to a sublicense from LMI,
the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide
basis.
Last login: Mon Jun 10 14:19:27 2019 from 192.168.141.144
cumulus@cumulus:mgmt-vrf:~$
```

The comments in Example 18-10 help you understand each step. As you can see, you need to provide the password for the SSH command when you create the folder for SSH keys on the remote host, as well as when you copy the key file there over SCP. However, you don't need it anymore when you perform the SSH connectivity once more at the end of the output.

After applying the keys, if you now redo the execution of the configuration script from Example 18-9, the configuration process is truly automated, as shown in Example 18-11.

Example 18-11 *Programmable Management of Linux Files with SSH Using RSA Keys*

```
$ ./manage_interaces.sh swp3 169.254.0.4/31 192.168.141.156 cumulus
Interface with the name "swp3" and IP "169.254.0.4/31" is to be created on
"192.168.141.156"
interfaces                                            100%  486   476.9KB/s
00:00
interfaces                                            100%  534   644.7KB/s
00:00
5: swp3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
```

```
    link/ether 00:50:56:3a:14:d8 brd ff:ff:ff:ff:ff:ff

    inet 169.254.0.4/31 scope global swp3

       valid_lft forever preferred_lft forever

    inet6 fe80::250:56ff:fe3a:14d8/64 scope link tentative

       valid_lft forever preferred_lft forever
```

You might argue that the examples with the interface configuration are straightforward. The point would be fair enough. However, the core idea of any programmability is to split the whole configuration process into small and easy steps that can be templated and parametrized and then deployed in a replicable manner.

There is no right or wrong way to do automation. Different contexts require different solutions. The next section looks at a newer way to manage the network functions that might be best in some situations: by using NETCONF/YANG.

Managing Network Devices with NETCONF/YANG

Together with YANG data models, the NETCONF protocol enables you to manage network devices in a programmable way. The main difference between the NETCONF/YANG approach and the traditional CLI configuration approach is that the data structure is more consistent with NETCONF/YANG. When you use NETCONF/YANG, you define the particular node that contains the data. This operation is precise, so you don't need to take care of the sequence of the commands, as you would need to do with CLI templates.

In the examples so far in this chapter, you have learned how to manage network elements via the CLI by using shell scripts. These CLI commands are sent using non-interactive SSH sessions. A NETCONF session, on the other hand, requires extensive message exchange, so it is impossible to use shell scripts for management of network devices via NETCONF.

Let's continue our example of creating an interface with an associated IP address, but now let's look at NETCONF/YANG and another vendor. This time, let's consider the Nokia SR OS running on a Nokia SR 7750 router. Currently, Nokia supports a vendor-proprietary YANG module and a subset of the OpenConfig modules. Nokia officially published the YANG modules in October 2019, long after other vendors published their modules. Without published YANG modules, the life of a developer is difficult. Before modules are published, a developer must perform reverse engineering, which involves configuring something in the network element and then extracting the configuration over NETCONF to get the YANG model for a specific context. Example 18-12 shows exchange of NETCONF hello messages between the management host and the Nokia router.

Example 18-12 *NETCONF Hello Exchange*

```
$ ssh  admin@nokia_router -p 830 -s netconf



admin@secgw-5.viplabs.de's password:

# NETCONF HELLO sent by the remote managed host

<?xml version="1.0" encoding="UTF-8"?>

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

    <capabilities>

        <capability>urn:ietf:params:netconf:base:1.0</capability>

        <capability>urn:ietf:params:netconf:base:1.1</capability>

        <capability>urn:ietf:params:netconf:capability:writable-

running:1.0</capability>

        <capability>urn:ietf:params:netconf:capability:notification:1.0</capability>

        ! The output is truncated for brevity

        <capability>urn:alcatel-lucent.com:sros:ns:yang:conf-rl3?module=alu-conf-

r13&amp;revision=2019-02-13</capability>

        !

    </capabilities>

    <session-id>100</session-id>

</hello>

]]>]]>


# NETCONF HELLO sent by the local management host

<?xml version="1.0" encoding="UTF-8"?>

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

 <capabilities>

  <capability>urn:ietf:params:netconf:base:1.0</capability>

 </capabilities>

</hello>

]]>]]>
```

As you learned earlier, the character sequence]]>]]> indicates the end of the NETCONF messages (see RFC 6241. This example involves manual interaction with the remote managed host over NETCONF. Both Python and Ansible have specific modules that handle such interactions, and you only need to focus on the content of your messages. Each message begins with # and an explanation of the intent of the message.

When the exchange of the NETCONF hello messages is finished, you can start applying the configuration, as illustrated in Example 18-13.

Example 18-13 *Configuring the Loopback Interface in the Nokia SR OS Using NETCONF*

```
# NETCONF EDIT-CONFIG message with the loopback details

<?xml version="1.0" encoding="UTF-8"?>

<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

<edit-config>

<target>

<running/>

</target>

<config>

<configure xmlns="urn:alcatel-lucent.com:sros:ns:yang:conf-r13">

 <router>

  <interface>

   <interface-name>test_loopback</interface-name>

   <address>

    <ip-address-mask>192.168.192.168/32</ip-address-mask>

   </address>

   <loopback>true</loopback>

   <shutdown>false</shutdown>

  </interface>

 </router>

</configure>

</config>

</edit-config>

</rpc>

]]>]]>


# NETCONF OK confirmation from the remote managed host

<?xml version="1.0" encoding="UTF-8"?>

<rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

    <ok/>

</rpc-reply>

]]>]]>
```

In the case of NETCONF, it's straightforward to separate the variables from the rest part of the XML message. The variables are the values inside the <> framing; these keys are the configuration nodes. The variables in Example 18-13 are the same as the variables in the previous examples: the interface name and the IPv4 addresses. In addition, in Example 18-13 you can see two more values that were not used before:

• The administrative state of the interface defines whether it's up or down. Theoretically, this value could have been defined earlier, but it was omitted for brevity. In Nokia SR OS, the newly created interface is disabled by default, however, so you need to bring it up manually.

• The interface loopback is or is not identified. Nokia SR OS handles interface naming and configuration differently than Cisco. With Nokia SR OS, you can create a Layer 3 interface with a name you like and then configure it as a loopback or map a physical port to it.

To verify that the interface is configured properly, you can check the output of the CLI from the router, as shown in Example 18-14.

Example 18-14 *Verifying the Loopback Interface Configuration on the Nokia Router*

```
*A:nokia_router# show router interface

===============================================================================

Interface Table (Router: Base)

===============================================================================

Interface-Name                   Adm       Opr(v4/v6)  Mode    Port/SapId

   IP-Address                                                   PfxState

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

! The output is truncated for brevity

test_loopback                    Up        Up/Down     Network loopback

   192.168.192.168/32                                           n/a

! The output is truncated for brevity

-------------------------------------------------------------------------------
```

```
Interfaces : 4

==============================================================================
```

After verifying the configuration, the next step is to create a script, using Ansible or Python, to manage the network functions with the variables. Example 18-15 provides the Python script. (Ansible is covered in Chapter 19.)

Example 18-15 *Python Script to Create the Interface in Nokia SR OS*

```
$ cat nokia_netconf.py

import sys, warnings

warnings.simplefilter("ignore", DeprecationWarning)

from ncclient import manager


# Variables

NOKIA_USERNAME = 'admin'

NOKIA_PASSWORD = 'admin'

NOKIA_PORT     = 830


# User functions

def npf(host, ip_address, if_name):

    xml_conf = """<config>

                    <configure xmlns="urn:alcatel-lucent.com:sros:ns:yang:conf-rl3">

                      <router>

                        <interface>

                          <interface-name>%s</interface-name>

                          <address>

                            <ip-address-mask>%s</ip-address-mask>

                          </address>

                          <loopback>true</loopback>

                          <shutdown>false</shutdown>

                        </interface>

                      </router>

                    </configure>

                  </config>""" % (if_name, ip_address)


    conn = manager.connect(host=host,

                           port=NOKIA_PORT,

                           username=NOKIA_USERNAME,

                           password=NOKIA_PASSWORD,

                           timeout=10,

                           device_params={'name': 'alu'},

                           hostkey_verify=False)


    conn.edit_config(target='running', config=xml_conf)


    conn.close_session()


# User functions

if __name__ == '__main__':

    npf(sys.argv[1], sys.argv[2], sys.argv[3])
```

The syntax of the Python is quite different from that of Bash, but the structure is similar. As you learned in Chapter 5, "Python Fundamentals," at the beginning of the snippet, you define the modules you are going to use, as well as some variables (username, password, and port). Then the function configures Nokia SR OS over NETCONF using the Nokia YANG data model. The template is the same as in Example 18-14, but this time there are two variables, **if_name** and **ip_address**, placed in the template marked with the **%s** symbol. To connect with the network function over NETCONF, the specific Python module **ncclient** (https://pypi.org/project/ncclient/) is used. This module is specially developed by the community to manage devices over NETCONF, and it includes all the relevant handlers. The function of this module in this example is to connect to the network and send the appropriate configuration to a particular datastore. If you compare Examples 18-14 and 18-15, you will see many similarities.

It is worth pointing out that parameters such as the hostname of the network function, the name of the interface, and its IPv4 address are provided as arguments during the launch of the script, as shown in Example 18-16.

Example 18-16 *Executing and Checking the Script*

```
$ sudo python nokia_netconf.py secgw-5.viplabs.de 192.168.192.169/32 npf_lo

$




*A: nokia_router# show router interface

===============================================================================

Interface Table (Router: Base)

===============================================================================

Interface-Name                   Adm       Opr(v4/v6)  Mode    Port/SapId

   IP-Address                                                   PfxState

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

npf_lo                           Up        Up/Down     Network loopback

   192.168.192.169/32                                           n/a

! The output is truncated for brevity

test_loopback                    Up        Up/Down     Network loopback

   192.168.192.168/32                                           n/a

! The output is truncated for brevity

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

Interfaces : 5

===============================================================================
```

If the execution of the Python script is successful, you get no notification. However, in the event of failure, as you learned in Chapter 14, you get notifications that help you understand what went wrong.

The script in Example 18-16 is straightforward, but it gives you valuable insight. If your network functions support NETCONF/YANG, you can manage the network as described here by modifying the YANG data structure in the Python script.

Managing Network Devices with RESTCONF/YANG

Recall that RESTCONF uses HTTPS as a transport protocol, with data encoded directly in JSON format; this makes RESTCONF a perfect protocol for all kinds of programmability.

There are several points you need to consider when you plan to use RESTCONF:

• RESTCONF requires SSL encryption, so you need to create either self-signed or CA-signed SSL certificates on your target network function to be able to connect to it.

• HTTPS-based applications don't use credentials to access the network devices directly; instead, they use Base64-encoded username/password pairs. Therefore, you need to transform your credentials into this format.

• RESTCONF uses HTTP transport, so you need to become comfortable with HTTP request types.

This section guides you through these considerations, using Arista EOS to configure a loopback interface.

Example 18-17 shows how to create SSL certificates and enable RESTCONF.

Example 18-17 *Preparing for RESTCONF/YANG Usage*

```
# Generate certificate/key pair

EOS1#security pki key generate rsa 4096 RESTCONF_KEY

EOS1#security pki certificate generate self-signed RESTCONF_CERT key RESTCONF_KEY

parameters common-name "rest_test.acme.com"

certificate:RESTCONF_CERT generated




# Enable RESTCONF with created cert/key pair (only relevant output is provided)

EOS1#show run | section management

management security

   ssl profile RESTCONF_SEC_PROFILE

      certificate RESTCONF_CERT key RESTCONF_KEY

!

management api restconf

   transport https default

      ssl profile RESTCONF_SEC_PROFILE
```

```
# Verification the RESTCONF works correctly

EOS1#show management api restconf

Enabled:            Yes

Server:             running on port 6020, in default VRF

SSL Profile:        RESTCONF_SEC_PROFILE

QoS DSCP:           none



# Create credentials for remote access

EOS1#show run | grep 'username'

username aaa privilege 15 secret aaa
```

In the PKI architecture, you always need a certificate/key pair. Therefore, in Example 18-17, you need to create this as a first step. The second step is to associate the generated certificate/key pair with an SSL security profile, which in the third step is associated with the RESTCONF configuration. You can also specify a port that RESTCONF processes listen to. If you don't do this (as shown in Example 18-17), the default port value, TCP port 6020, is used. The last thing you need to do is to create the credentials that will be used with RESTCONF.

Now that the network devices are prepared for RESTCONF operation, you need to understand how to authenticate your REST requests to a network device. The REST API doesn't support the user credentials per se, as you learned in Chapter 14. However, it has the concept of the Authorization headers, which are part of each message sent to the network device. Authentication can be achieved in different ways (for example, using API tokens), and one of the easiest ways is to use a basic authorization type, such as the string *username:password* encoded in Base64 format, with this value then added to the authorization header. Figure 18-3 shows a simple way to convert credentials into Base64 format using the free online tool available at www.base64encode.org.

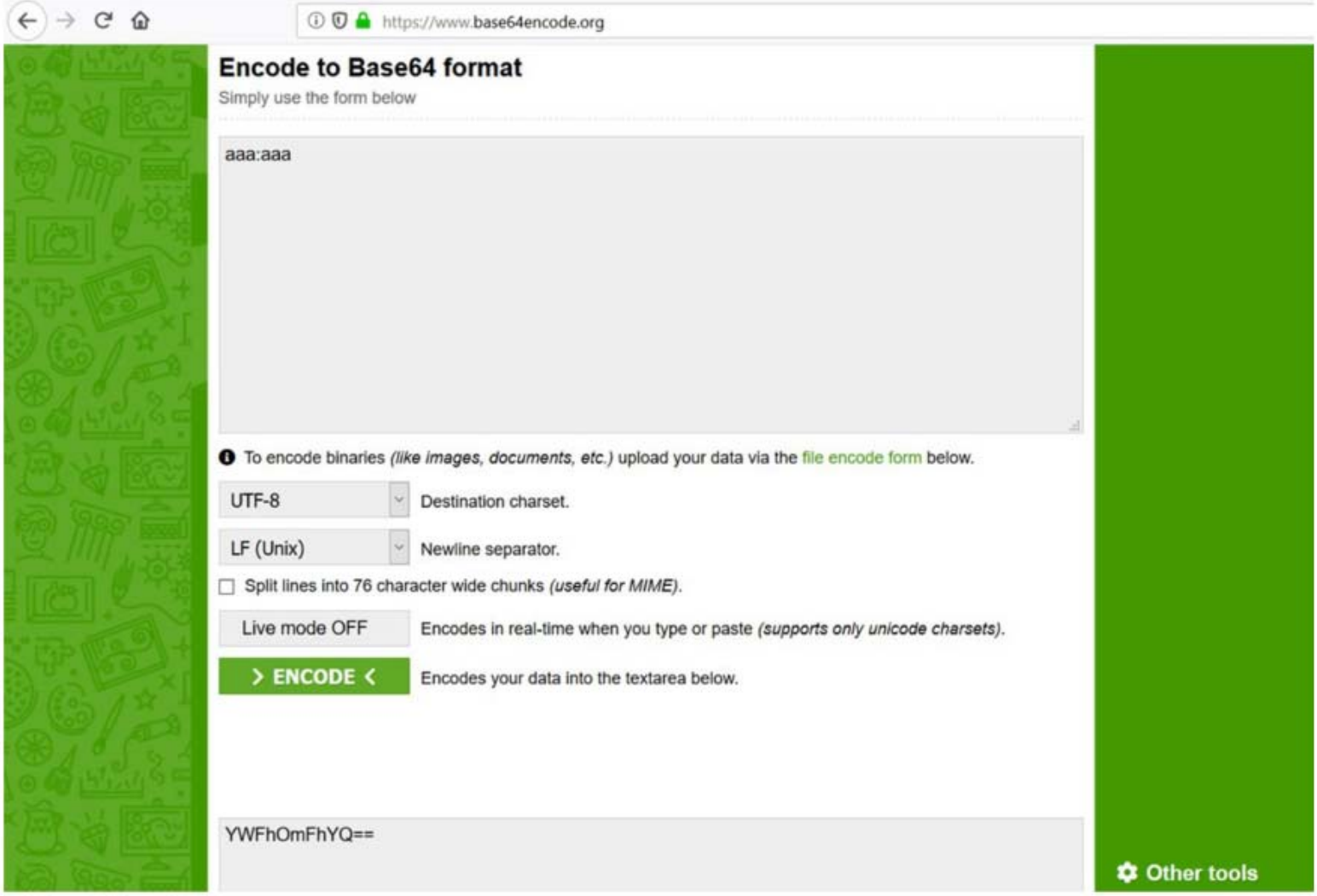

Figure 18-3 *Converting a username:password Tuple to Base64 Format*

As shown in this figure, you put your string precisely in the *username:password* form configured on the device earlier (refer to Example 18-17). Then you just click the Encode button, and at the bottom of the screen, you see the encoded value, which you can then use in the Authorization header in RESTCONF message.

The final step before you can get or set the configuration over RESTCONF is to determine the address of the device to be configured (basically, the URI you need to access). As you learned in Chapter 14, according to RFC 8040, the URI of the node is typically an address in the format https://*hostname*/restconf/data/*yang_module*:*container*.

For information about supported YANG modules for a particular vendor, you can typically look at the GitHub pages or release notes for the version of the network operating system that is being used (refer to Chapter 13). The current example uses Arista, and Figure 18-4 shows the GitHub page where you can find the needed information.

https://github.com/aristanetworks/yang/tree/master/EOS-4.21.3F/openconfig/public/release/models

Why GitHub? Enterprise Explore Marketplace Pricing Search Sign in Sign up

aristanetworks / yang

Watch 27 Star 13 Fork 2

Code Issues 0 Pull requests 0 Security Insights

Branch: master yang / EOS-4.21.3F / openconfig / public / release / models / Create new file Find file History

Aparna Add supported OpenConfig models to EOS-4.21.3F Latest commit f8730d7 on 21 Jan

..		
acl	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
aft	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
bfd	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
bgp	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
catalog	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
interfaces	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
isis	Add supported OpenConfig models to EOS-4.21.3F	5 months ago
lacp	Add supported OpenConfig models to EOS-4.21.3F	5 months ago

Figure 18-4 *Arista YANG Modules*

As indicated in this figure, Arista supports OpenConfig YANG modules. OpenConfig is a core YANG data model that Arista uses in its product. OpenConfig YANG data models implemented in Arista EOS have various augmentationsdeviations, but there are no vendor-proprietary YANG modules.

In Chapter 13, you learned how to find the name of a YANG module and the name of a container. Using that knowledge and what you have learned about RESTCONF authorization in this chapter, you can create a RESTCONF request like the one shown in Example 18-18.

Example 18-18 *RESTCONF GET Request Using an Authorization Header*

```
$ curl -X GET https://EOS1:6020/restconf/data/openconfig-interfaces:interfaces --header
"Authorization: Basic YWFhOmFhYQ==" --insecure | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2321    0  2321    0     0   9070      0 --:--:-- --:--:-- --:--:--  9101
{
    "openconfig-interfaces:interface": [{
! The output is truncated for brevity
        "config": {
            "arista-intf-augments:load-interval": 300,
            "description": "",
            "enabled": true,
            "loopback-mode": true,
            "name": "Loopback0",
            "openconfig-vlan:tpid": "TPID_0X8100",
            "type": "softwareLoopback"
        },
! The output is truncated for brevity
        "name": "Loopback0",
        "xstate": {
            "enabled": true,
            "loopback-mode": false,
            "openconfig-vlan:tpid": "TPID_0X8100"
        },
        "subinterfaces": {
            "subinterface": [
                {
                    "config": {
                        "enabled": true,
```

```
                    "index": 0
                },
                "index": 0,
                "openconfig-if-ip:ipv4": {
                    "addresses": {
                        "address": [
                            {
                                "config": {
                                    "ip": "10.0.0.22",
                                    "prefix-length": 32
                                },
                                "ip": "10.0.0.22",
                                "state": {
                                    "ip": "10.0.0.22",
                                    "prefix-length": 32
                                }
                            }
                        ]
                    },
                    "config": {
                        "dhcp-client": false,
                        "enabled": true,
                        "mtu": 1500
                    },
                    "proxy-arp": {
                        "config": {
                            "mode": "DISABLE"
                        },
                        "state": {
                            "mode": "DISABLE"
                        }
                    },
                    "state": {
                        "dhcp-client": false,
                        "enabled": true,
                        "mtu": 1500
                    },
                    "unnumbered": {
                        "config": {
                            "enabled": false
                        },
                        "state": {
                            "enabled": false
                        }
                    }
                }
! The output is truncated for brevity
            }
        }
    }
}]
}
```


Because RESTCONF works over HTTP, you can use the handy Linux tool cURL as shown in this example. The **-X** key sets the HTTP message

type to **GET**, which means you collect the information already configured on the network function. Then, using the key **--header** (or **-H**), you provide the authorization data, including the credentials encoded in Base64 format. You use the **--insecure** key to avoid the certificate check, which you typically want to do with self-signed certificates. You can also see in this example that the output of the **curl** command is the input for the Python module json.tool. This makes the output format easy to read; without this command, the JSON output of **curl** is just a single string. Despite the fact that this string has all the correct JSON framing, it's difficult to read.

Instead of using cURL, you can use any other tool that allows you to work with a REST API. One of the most popular tools in this area is Postman, which provides a flexible way to generate the REST API commands and receive responses. (Postman is beyond the scope of this book.)

In addition to collecting data by using RESTCONF, it's also possible to create a new configuration, as discussed in Chapter 14. Although it's theoretically possible to create a shell script that uses cURL to provision a network device, this is not necessarily the best way of managing network devices over RESTCONF. Example 18-19 provides some ideas about how to create a Python script to manage the network function with a REST API.

Example 18-19 *Python Script to Manage Network Elements over RESTCONF*

```
$ cat restconf_put.py

import sys

import requests

from requests.auth import HTTPBasicAuth


# Variables

NF_USERNAME = 'aaa'

NF_PASSWORD = 'aaa'

NF_RESTPORT = 6020


# User functions

def set_interfaces(NF_HOSTNAME, NEW_INTERFACE_NAME, NEW_INTERFACE_IP):

    """

    Setting the interfaces in RESTCONF

    """

    templated_stuff = """{

                            "openconfig-interfaces:name": "%s",

                            "openconfig-interfaces:config": {

                                "enabled": true,

                                "name": "%s",

                                "loopback-mode": true,

                                "type": "softwareLoopback"

                            },

                            "openconfig-interfaces:subinterfaces": {

                                "subinterface": [

                                    {

                                        "index": 0,

                                        "config": {

                                            "enabled": true,

                                            "index": 0

                                        },

                                        "openconfig-if-ip:ipv4": {

                                            "addresses": {

                                                "address": [

                                                    {

                                                        "ip": "%s",

                                                        "config": {

                                                            "ip": "%s",

                                                            "prefix-length": %s

                                                        }

                                                    }

                                                ]

                                            }

                                        }
```

```
                                    }

                                ]

                            }

                        }""" % (NEW_INTERFACE_NAME, NEW_INTERFACE_NAME,

NEW_INTERFACE_IP.split('/')[0], NEW_INTERFACE_IP.split('/')[0],

NEW_INTERFACE_IP.split('/')[1])


    url = "https://%s:%s/restconf/data/openconfig-interfaces:interfaces/interface=%s" %

(NF_HOSTNAME, NF_RESTPORT, NEW_INTERFACE_NAME)

    set_rest = requests.put(url, auth=HTTPBasicAuth(NF_USERNAME, NF_PASSWORD),

verify=False, data=templated_stuff)


    print (set_rest)


# Main body


if __name__ == "__main__":

    set_interfaces(sys.argv[1], sys.argv[2], sys.argv[3])
```

This script is quite similar in structure to Example 18-15. At the beginning of the script, some modules are imported, and they add the function necessary for this script. The module sys, which was also used earlier, makes it possible to send variables as arguments in the CLI, and the module requests is in charge of interacting with the HTTP API used in RESTCONF. The requests module has a built-in ability to convert your credentials into Base64 encoding, so you don't need to do this conversion manually.

When the modules are imported, variables such as the credentials and RESTCONF port are provided. You can also pass these variables by using the CLI if you like. If you have the same credentials for all your devices, by storing them in the Python file, you can avoid entering them each time, and you can still provide the hostname of the target managed network device.

After the block with variables, you see the user-defined functions. This is a convenient way to structure the code because the user functions help make the code reusable. Although this structure is not needed in this small script, applying it is a good habit. Within the user-defined function **set_interfaces**, you can see the template for the interface followed by the variables. The template contains JSON data that is later used as a payload for a **PUT** message. The tricky part with the payload is that you need to provide the IP address and the subnet mask separately. Earlier in this chapter, you saw the IP address and the subnet mask together in a single line; in the OpenConfig YANG data model for interfaces, they need to appear separately. To achieve that, you use the function **split('/')**, which divides a single variable into multiple parts, using the separator provided in single quotes. When the split is complete, you can call the parts of the variable by using an index **[x]** starting from 0 for the leftmost split value.

When the template is filled in with variables, the payload for the REST API call is ready, and you can make the call by using the requests module. The type of the request is specified as **requests.put**, and you can change this request to switch **PUT** to **POST**, **PATCH**, or **GET**, depending on your needs. Within the REST API request, you need to provide the URI, which is also a template filled in with variables, the authorization data, and the payload (in this case, **data**). Because a self-signed certificate is used, you should disable the certificate check by using the key/value pair **verify=false**. The user-defined function ends by printing the output of the execution result.

The main body the script calls the user-defined function and provides to it the variables that it receives from the CLI. Example 18-20 shows how the Python script is executed.

Example 18-20 *Execution of a Python Script to Configure the Network Function over RESTCONF*

```
$ python restconf_put.py EOS1 Loopback456 192.168.192.168/32

/home/aaa/book/non_cisco_prog/python_restconf/venv/lib/python3.7/site-

packages/urllib3/connectionpool.py:851: InsecureRequestWarning: Unverified HTTPS

request is being made. Adding certificate verification is strongly advised. See:

https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings

  InsecureRequestWarning)

<Response [200]>
```

This example launches the Python script the same way it is launched for NETCONF (refer to Example 18-15). In output shown here, you can see that the certificate isn't verified. This is okay because you manually disabled this check in Example 18-19. However, in a production network, you should use PKI and centralized certificates to avoid security issues. The second part of the output shows the HTTP message code, which in this case is 200, which means OK. Now, by using the REST API **GET** call provided in Example 18-18, you verify the status of the newly created interface.

Summary

This chapter provides an overview of the main APIs for configuring network functions with the shell. It also provides examples of using Python to manage non-Cisco operating systems. The following are some of the specifics covered in this chapter:

• Programmability has become important in the networking industry, and many vendors have become involved with programmability.

• To begin automating your network, it is important to determine which APIs your vendor supports.

• The most popular programmable interfaces today are NETCONF, RESTCONF, and gRPC. However, you can use the CLI for programmability by using scripts.

• To successfully apply automation, you need to understand what variable will vary across different network devices' management. Based on

that understanding, you can develop an API.

• The most popular and influential scripting language is Python. Nevertheless, you can use Bash, Perl, or any other scripting language that suits a particular solution.

• NETCONF, RESTCONF, and gRPC are built around a YANG data model, and they are basically just different transport and encoding protocols. You should check a vendor's supported YANG modules before you start developing automation.

• Sometimes modules aren't published. In such a case, you need to either ask your vendor for them or to do reverse engineering to extract the configuration from the network function over NETCONF/RESTCONF and analyze the YANG module.

Chapter 19. Ansible

The chapters so far in this book cover various components of the network programmability stack and how those components work together to provide a usable set of tools for network automation. This chapter covers yet another one of those tools: Ansible.

Ansible is an application that makes use of several of the components of the network programmability stack to abstract network automation tasks. As a matter of fact, the applications of Ansible extend well beyond network automation and into other domains, such as compute and application automation. The popularity of Ansible is due to the simplicity of the tool as well as the not-so-steep learning curve. Learning Ansible does not require previous knowledge of Python or any other programming language. It also does not require a deep understanding of any of the programmability protocols such as NETCONF or RESTCONF. To get started and be productive with Ansible, you don't even need to know what an API is!

By the end of this chapter, you should have a good working knowledge of Ansible and be able to write playbooks to automate everyday tasks.

Ansible Basics

Many examples related to network automation (and network programming as an inherited use case) covered so far in this book might seem complex. They require you to know some programming languages, such as Python, at a sufficiently deep level. In addition, you need to understand HTTP/REST in some detail. And then, on top of that, you need to know how to work with the different encodings as well as send and receive HTTP requests—using Python (or your language of choice). This can be a time-consuming and exhausting entry point into network automation. By this point in the book, you should have already mastered the appropriate knowledge.

But what if there is an option to approach network automation and programming in a more straightforward manner? In other words, what if there is a faster, simpler entry point into network automation?

At some point in time, the IT industry started looking for ways to simplify automation activities. It's crystal clear that the primary goal of any automation framework is to reduce the time spent on repetitive activities and to make the saved time available for something more valuable. The highest efficiency is achieved when automating tasks that must be performed across several (or several hundred or even several thousand) devices. Ansible aims to provide such efficiency.

Ansible is a relatively new tool, created in 2012 to perform simultaneous configuration of servers. Since then, it has evolved enormously, and today, perhaps, it's the first tool people think about when about it comes to IT automation in general or network automaton in particular. Several SDN controllers (including Cisco Crosswork and Juniper Contrail) use Ansible to perform configuration of network functions.

Ansible allows you to run ad hoc commands or predefined sequences of commands called *playbooks* across multiple devices simultaneously. In the context of network automation, Ansible can be heavily used for automated configuration provisioning, information collection, automated software upgrades, verification, and other activities. The functionality is practically limited only by your imagination.

Ansible, which is now owned by Red Hat, is an open-source project that is distributed under the GNU license. This means that Ansible is available for everybody to use and that everybody can contribute to its development to enrich the functionality further.

How Ansible Works

Ansible is an agentless automation tool, which means that nothing needs to be installed on the managed host besides SSH (or NETCONF, as you will see later in this chapter). If a host is capable of running Python, then Python should be installed as well. In any case, you need to install Ansible itself on the managing host, and Example 19-1 shows how you can do it on an RPM-based distro such as CentOS.

Example 19-1 *Installation of Ansible on a Managing Host*

```
$ sudo yum install -y ansible
```

When Ansible is installed on the managing host, it can immediately start managing hosts. The Figure 19-1 illustrates the call flow for a single command in Ansible.

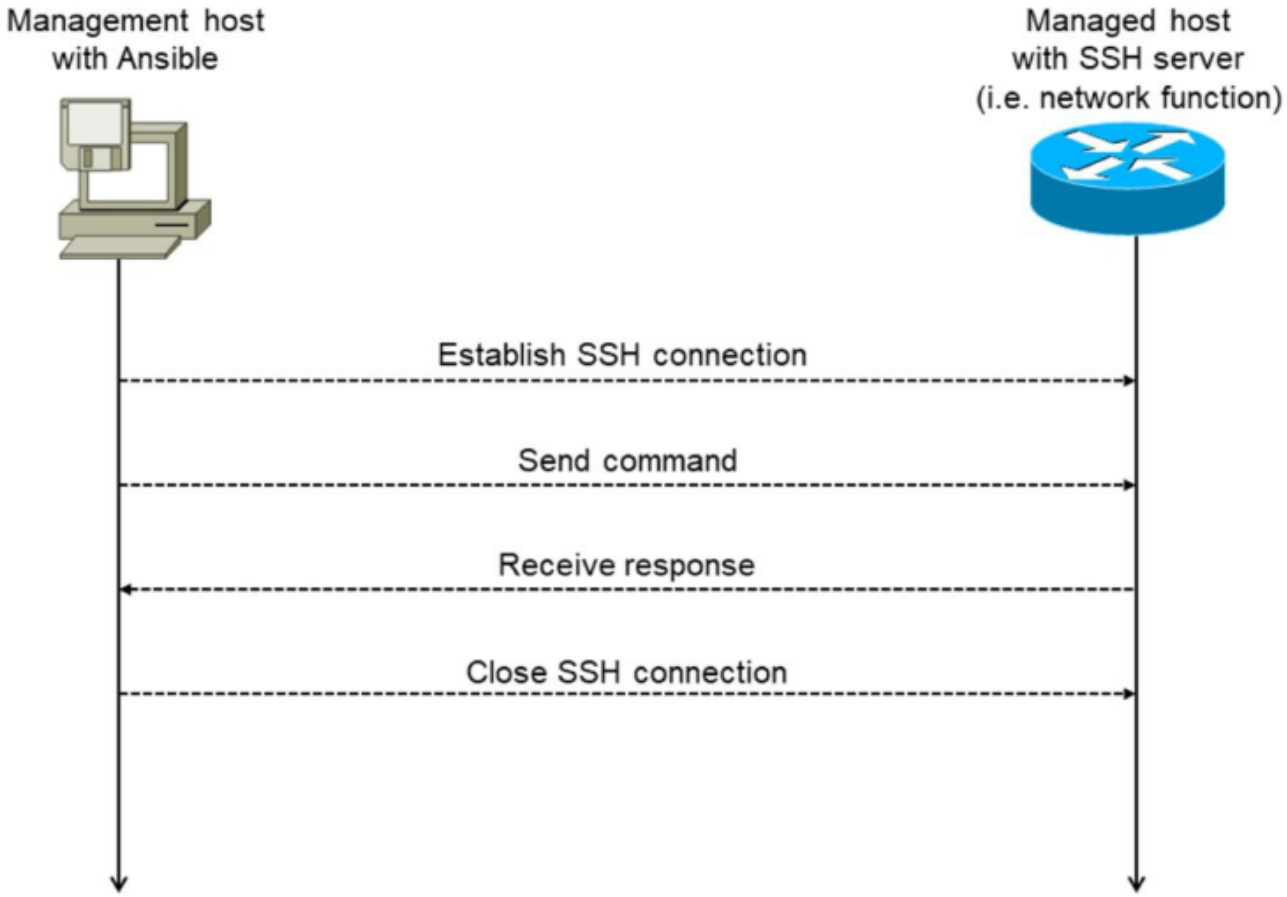

Figure 19-1 *Ansible Call Flowfor a Single Command*

Figure 19-1 shows the following sequence of actions:

1. The managing host with Ansible establishes an SSH connection with the managed host.

2. The managing host sends the command(s) to execute on the managed host. For example, it could be configuration or **show** commands in the context of network functions.

3. The managing host receives the output of the command execution from the managed host. For a **show** command, this is the output of that command (meaning some operational data), whereas for a configuration command, this is empty output for a success and an error message for a failure.

4. The managing host terminates the SSH connection with the managed host.

It is important to understand where Ansible stores the configuration data that is necessary for its operation. Example 19-2 shows the use of the command **ansible –version** to find the most important locations where Ansible stores information on the system.

Example 19-2 *Reviewing the Ansible Default Configuration Parameters using Ansible Version Command*

```
$ ansible --version
ansible 2.9.11
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/aaa/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Jul 13 2018, 13:06:57) [GCC 4.8.5 20150623 (Red Hat
4.8.5-28)]
```

In Example 19-2, the version (2.9.11 in this example) refers to the version of the Ansible installed. The first digit identifies the major software version—in this case, Ansible 2.0. The second digit refers to the major update (in this case, 9), and the third digit (in this case, 11) refers to the minor update. The major version typically has some modifications, such as introduction of new modules or plugins, whereas the minor version typically contains the bug fixes or security updates.

The default configuration file is /etc/ansible/ansible.cfg. It contains all parameters relevant to Ansible functioning, such as SSH connection parameters, paths to Ansible and Python modules, and the default username. In the vast majority of automation use cases, you won't change these default parameters.

The Ansible Python module location points to the folder that contains Ansible modules. You will learn about modules later in this chapter. For now, you can think of an Ansible module as a Python program performing certain activities (for example, copying files in Linux, performing configurations on Cisco NX-OS switches). The important point for now is that modules are all installed automatically, and you don't need to worry about getting them but can just use them.

Near the end of Example 19-2, the executable location points to the folder where Ansible is installed, and you can see the Python version that is in use, which in some cases may affect the execution of Ansible modules.

In the configuration file mentioned in the Example 19-2, there is a default link to the inventory, as shown in Example 19-3.

Example 19-3 *Default Path to the Ansible Inventory*

```
$ cat /etc/ansible/ansible.cfg | grep 'inventory'

#inventory      = /etc/ansible/hosts
```

The inventory is vital to the functioning of Ansible. It is the file or set of the files where hostnames of the managed hosts (and possibly some other parameters) are listed and grouped in specific categories, as illustrated in Example 19-4. The inventory is the file that is used each time Ansible launched.

Example 19-4 *Simple Ansible Inventory File*

```
$ cat /etc/ansible/hosts

[linux]

localhost



[ios]

CSR1

CSR2



[iosxr]

XR3

XR4



[nexus]

NX1

NX2
```

It is possible to run Ansible automation commands or playbooks against a single host (such as XR3), a group (such as iosxr), or all hosts together. You will learn more about this later in this chapter.

The file shown in Example 19-4 lists only groups and hostnames. You need to make sure that your managing host can reach all the listed hostnames. You can achieve this either by enabling DNS resolution or updating your Linux hosts file, as shown in Example 19-5.

Example 19-5 *The Linux Hosts Files with Entries for Ansible Hosts*

```
$ cat /etc/hosts

127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4

::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.1.101 CSR1

192.168.1.102 CSR2

192.168.1.111 XR3

192.168.1.112 XR4

192.168.1.121 NX1

192.168.1.122 NX2
```

In some cases, you may be unable to modify the /etc/hosts file (for example, if you don't have the appropriate permissions in the system). There is an alternative way to provide details about IP addresses: by putting the IP addresses directly in the Ansible inventory file. To do so, you use the specific variable name **ansible_ssh_host**, as shown in Example 19-6.

Example 19-6 *Ansible Inventory with IP Addresses*

```
$ cat /etc/ansible/hosts

[linux]

localhost



[ios]

CSR1 ansible_ssh_host=192.168.1.101

CSR2 ansible_ssh_host=192.168.1.102

!

! Further output is truncated for brevity
```

Note

It is possible to define other variables in the inventory file, as you will learn later in this chapter.

Now it's time to put all the pieces together and look at a simple example of Ansible operation. Example 19-7 shows the execution of an Ansible ad hoc command.

Example 19-7 *Sample Ansible Ad Hoc Command*

```
$ ansible iosxr --user=cisco --ask-pass --connection=network_cli --module-
name=iosxr_command --args="commands='show version | include uptime'"
SSH password:

XR3 | SUCCESS => {
    "changed": false,
    "stdout": [
        "XR3 uptime is 32 weeks, 2 days, 11 hours, 28 minutes"
    ],
    "stdout_lines": [
        [
            "XR3 uptime is 32 weeks, 2 days, 11 hours, 28 minutes"
        ]
    ]
}
XR4 | SUCCESS => {
    "changed": false,
    "stdout": [
        "XR4 uptime is 20 weeks, 4 days, 5 hours, 43 minutes"
    ],
    "stdout_lines": [
        [
            "XR4 uptime is 20 weeks, 4 days, 5 hours, 43 minutes"
        ]
    ]
}
```

By typing **ansible** at the Linux shell, you can execute an Ansible ad hoc command. With an ad hoc command, no script is created; rather, the required command is provided in {*arguments*} to the Ansible application.

Note

It is also possible to use sequenced commands called playbooks, as you will learn later in this chapter, in the section "The World of Ansible Modules."

The first attribute in Example 19-7, **iosxr**, is the name of the group from the Ansible inventory. It defines the nodes that the managing host will try to reach for the execution of the command. The command itself is defined by the two attributes **--module-name=iosxr_command** and **--args="commands='show version | include uptime'"**. The first of these attributes defines which Ansible module to use, and the second one passes the arguments required for module execution.

Additional arguments can be used as well, depending on the circumstances. For example, in Example 19-7, the argument **--user=cisco** defines the username for the SSH session, and the argument **--ask-pass** instructs Ansible to ask for the password for the SSH session. The last argument in Example 19-7, **--connection=network_cli**, defines which Python plug-in is used for establishing connectivity to the managed hosts. (You will learn more about such plug-ins later in this chapter, in the section "Variables and Facts.")

Ansible's built-in help lists and describes all the available keywords, as shown in Example 19-8.

Example 19-8 *Built-in Help in Ansible*

```
$ ansible --help
Usage: ansible <host-pattern> [options]

Define and run a single task 'playbook' against a set of hosts

Options:
  -a MODULE_ARGS, --args=MODULE_ARGS
                        module arguments
  --ask-vault-pass      ask for vault password
  -B SECONDS, --background=SECONDS
                        run asynchronously, failing after X seconds
                        (default=N/A)
  -C, --check           don't make any changes; instead, try to predict some
                        of the changes that may occur
```

```
  -D, --diff            when changing (small) files and templates, show the
                        differences in those files; works great with --check
  -e EXTRA_VARS, --extra-vars=EXTRA_VARS
                        set additional variables as key=value or YAML/JSON, if
                        filename prepend with @
  -f FORKS, --forks=FORKS
                        specify number of parallel processes to use
                        (default=5)
  -h, --help            show this help message and exit
  -i INVENTORY, --inventory=INVENTORY, --inventory-file=INVENTORY
                        specify inventory host path or comma separated host
                        list. --inventory-file is deprecated
  -l SUBSET, --limit=SUBSET
                        further limit selected hosts to an additional pattern
  --list-hosts          outputs a list of matching hosts; does not execute
                        anything else


! The output is truncated for brevity
```

Let's talk for another moment about Example 19-7. After the execution of the Ansible ad hoc command, there should be some output. For each host from the group {iosxr}, there is output, which is similar to the output you would get from locally executing the CLI the command **show version | include uptime** on a Cisco IOS XR device. Two variables contain the output: **stdout** contains unformatted output, and **stdout.lines** contains output split into lines based on a standard Linux newline character. Example 19-7 does not show **stdout.lines** because there is only one line, but this variable is beneficial in extensive output.

Ad Hoc Commands and Playbooks

As mentioned earlier in this chapter, it is possible to run ad hoc Ansible commands, and it is also possible to run playbooks, which contain sequences of actions. To better understand the difference between these two approaches, in this section we solve the following tasks:

- Verify the reachability of the network functions
- Check the software version

There are plenty of ways you could complete these tasks. For example, you could first try to ping the network device (which is effectively the solution for the first task) and then connect to the network element to collect the necessary information. Alternatively, you could complete both tasks by using a single data collection operation: Run the **show version** command on a remote network device, and you know that the network is functioning if it responds to the **show version** command sent over SSH. Example 19-9 shows this approach.

Example 19-9 *An Ansible Ad Hoc Command Using the **iosxr_command** Module*

```
$ ansible iosxr --user=cisco --ask-pass --connection=network_cli --module-
name=iosxr_command --args="commands='show version | include Cisco IOS XR'"
SSH password:


XR3 | SUCCESS => {
    "changed": false,
    "stdout": [
        "Cisco IOS XR Software, Version 6.1.4[Default]"
    ],
    "stdout_lines": [
        [
            "Cisco IOS XR Software, Version 6.1.4[Default]"
        ]
    ]
}
XR4 | SUCCESS => {
    "changed": false,
    "stdout": [
        "Cisco IOS XR Software, Version 6.1.4[Default]"
    ],
    "stdout_lines": [
        [
```

```
            "Cisco IOS XR Software, Version 6.1.4[Default]"
        ]
    ]
}
```

Instead of using ad hoc command, you can create an Ansible playbook relaying the same module as shown in Example 19-10.

Example 19-10 *An Ansible Playbook Using the* ***iosxr_command*** *and* ***debug*** *Modules*

```
$ cat npf_iosxr_show_version.yml
---
- hosts: iosxr
  connection: network_cli

  tasks:
      - name: Collect version from Cisco IOS XR
        iosxr_command:
            commands:
                - show version | include Cisco IOS XR
        register: show_output

      - name: Show results
        debug:
            msg: "{{ show_output.stdout }}"
...



$ ansible-playbook npf_iosxr_show_version.yml --user=cisco --ask-pass
SSH password:

PLAY [iosxr]
*******************************************************************************

TASK [Collect version from Cisco IOS XR]
***************************************************
ok: [XR3]
ok: [XR4]

TASK [Show results]
**********************************************************************
ok: [XR3] => {
    "msg": [
        "Cisco IOS XR Software, Version 6.1.4[Default]"
    ]
}
ok: [XR4] => {
    "msg": [
        "Cisco IOS XR Software, Version 6.1.4[Default]"
    ]
}

PLAY RECAP
********************************************************************************
XR3                        : ok=2    changed=0    unreachable=0    failed=0
XR4                        : ok=2    changed=0    unreachable=0    failed=0
```

The first significant difference between these two approaches is that a playbook is a file where you compose the logic of your automation. It's written using the YAML language, which you learned about in Chapter 12, "YAML." The information in YAML format has leading --- and ending

... characters. A playbook starts with some essential statements regarding the general execution process. For instance, in Example 19-10, the **hosts** statement defines the nodes against which the playbook will be executed; in this case, the scope is **iosxr**. You can see in Example 19-9 that this information is also provided in the ad hoc command but in a slightly different format. Next, the playbook in Example 19-10 shows important information contained in the **connection** field, which defines the connectivity plug-in that should be used to connect to nodes; the ad hoc command in Example 19-9 provides the same information.

In Example 19-10, the **tasks** list shows activities to be performed. In this list, it is important to use a - symbol followed by a space before each task. Each of the tasks starts with the keyword **name**. This field is not mandatory, but it's highly recommended as it helps the reader understand the purpose of each step. The next keyword is mandatory, as it calls the particular Ansible module, such as **iosxr_command** or **debug**. Every module has some specific keys, as each has a unique set of input and output parameters.

The last important point about Example 19-10 is that it introduces the concept of variables. By default, Ansible playbooks do not print to **stdout** the result of the execution of a tasks; they print only status (OK or NOT OK), and this is entirely okay because playbooks are typically used for solving complex tasks. In this case, the output needs to be printed to the CLI, so the **debug** module is used. However, in order for the **debug** module to show something, you need to save the result of the previous module execution to some variable. This is accomplished by using the key **register**, which saves all the output of the module to a variable (**show_output** in this example).

Note

The name of the variable is an arbitrary value.

After the output is saved to the variable, it can be used anywhere else in the playbook. As you can see in Example 19-10, it is called in the second task, using the **{{ show_output.stdout }}** construction. In Ansible, all variables are called by using the **{{** *variable* **}}** format. In this case, another key in the variable is defined to filter the output further. How this variable is structured depends on the module, and you can find this information in the documentation at the official Ansible website (https://docs.ansible.com).

After a playbook is created, the next step is to execute it by using the **ansible-playbook** command. After a playbook is launched, you can follow its execution. The **name** keyword in the playbook makes it evident which tasks are successful and which ones are not. Without names, you have to count the number of the tasks to identify the status of each task. If a task is performed successfully, you see ok: [*hostname*] in the output. There are four possible results of task execution:

• **ok**: The task is performed successfully.

• **changed**: The task is performed successfully, and some changes are made on the remote node (which is common for configuration modules).

• **failed**: The execution of the task is unsuccessful.

• **unreachable**: The remote node isn't reachable from the managing host.

In Example 19-10, you can see at the end of the playbook execution a PLAY RECAP section, which provides a summary of all the task status information. The underlying idea with playbook execution is that the same task is performed for all defined nodes in a task-by-task fashion. If some task fails or if a node is unreachable, that host is excluded from the list of nodes for further execution in the current execution.

Note

It is possible to change the standard behavior, as you will learn later in this chapter.

If you need to perform an elementary check or configuration, so that a single Ansible module is enough, you may want to use Ansible ad hoc commands. In more complicated cases, you need to use playbooks. For example, if you need to collect information about the software versions of Cisco IOS XR routers and save it somewhere using Ansible, you need to use playbooks and two operations: collect and save (refer to Example 19-10). The number of tasks you can include in a single playbook is virtually unlimited, and you can optimize each operational process, even those that are very complex.

The World of Ansible Modules

Earlier in this chapter, you saw examples of two Ansible modules, and now it's time to talk about them in more detail. Ansible is built on modules, and they are one of its strongest features. Modules are the workhorses that make Ansible such a powerful tool. In a nutshell, a *module* is a Python program with a clear structure and proper documentation that is executed within Ansible to perform a particular task. Figure 19-2 illustrates a generic Ansible module.

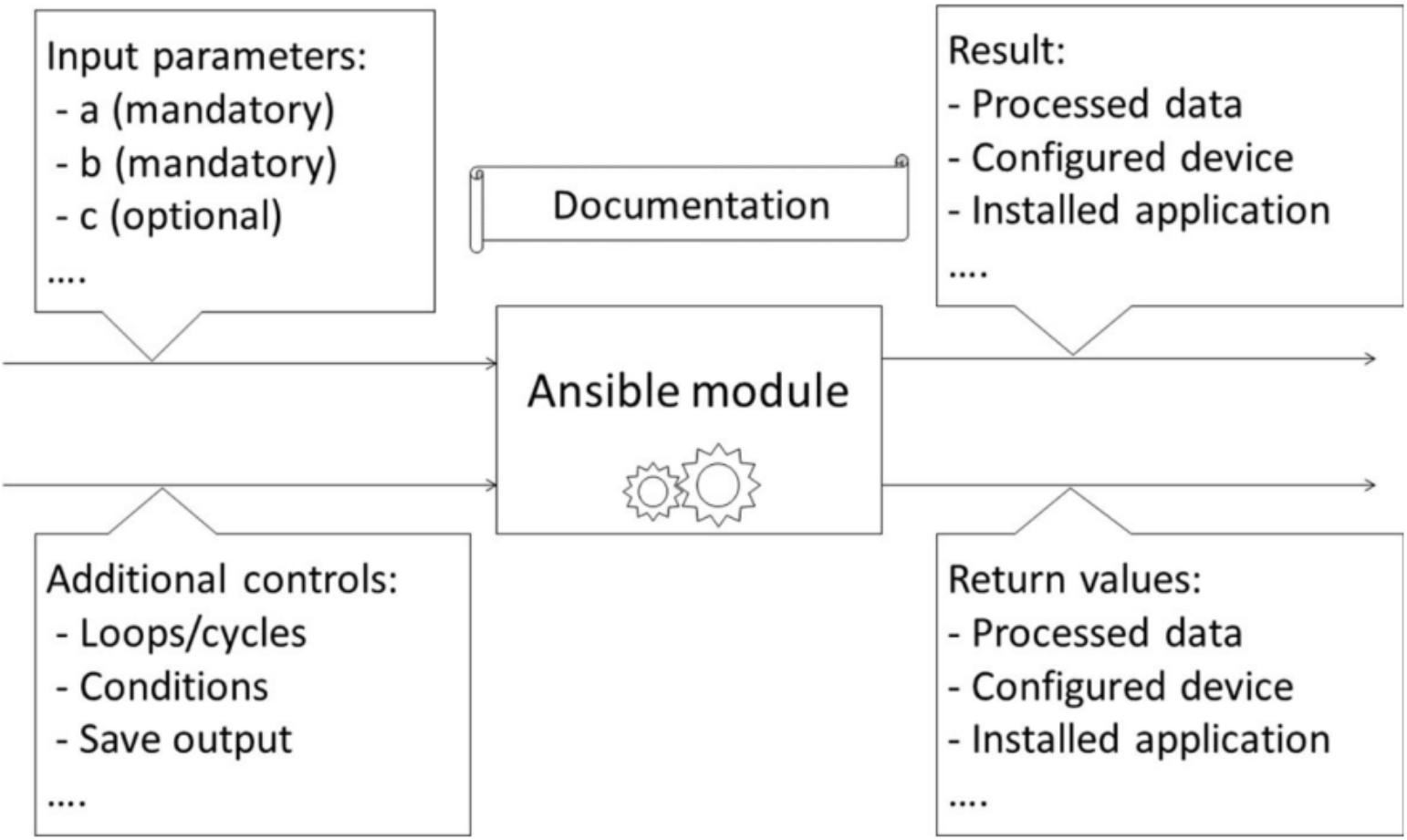

Figure 19-2 *Structure of a Generic Ansible Module*

Each Ansible module has input parameters (potentially many of them) that are necessary for its functioning. For example, earlier in this chapter, the module **iosxr_command** was used with the single defined parameter **commands**. Table 19-1, which is from the official Ansible documentation for the **iosxr_command** module, lists all the possible parameters for this module.

Table 19-1 *Input Parameters for the Ansible Module* ***iosxr_command***

Parameter	Choices/Defaults	Comments
commands (required)	—	Lists the commands to send to the remote IOS XR device over the configured provider. The resulting output from the command is returned. If the ***wait_for*** argument is provided, the module is not returned until the condition is satisfied or the number of retries has expired.
Interval	Default: **1**	Configures the interval, in seconds, to wait between retries of the command. If the command does not pass the specified conditions, the **Interval** parameter indicates how long to wait before trying the command again.
match	Choices: **any** **all**	The ***match*** argument is used in conjunction with the ***wait_for*** argument to specify the match policy. Valid values are **`all`** and **`any`**. If the value is set to **`all`**, then all conditionals in the **wait_for** argument must be satisfied. If the value is set to **`any`**, then only one of the values must be satisfied.
retries	Default: **10**	Specifies the number of times a command should be retried before it is considered failed. The command is run on the target device every retry and evaluated against the ***wait_for*** conditions.
wait_for	—	Lists the conditions to evaluate against the output of the command. The task waits for each condition to be true before moving forward. If the conditional is not true within the configured number of retries, the task fails.

If a parameter that is required (for example, **commands**) doesn't have a default value, it must be explicitly configured, as shown in Example 19-10. If a parameter has a default value, you can leave it as it is; in most cases, this is what you do. If a parameter isn't marked as required and doesn't have a default value, it means it's optional and can change the behavior of the module execution, if provided.

Additional controls influence the process of module execution. None of these controls are mandatory, but you can add them to enrich the logic of an automation. For instance, Example 19-10 includes the keyword **register**, which instructs the module to save the output (whatever the module prints to **stdout**) to some variable. Table 19-2 shows some of the additional controls that are most widely used in network programming. Later in this chapter, in the section "Extending Ansible Capabilities," you will see how these controls are used in some real-life use cases.

Table 19-2 *Control Functions for Modules*

Command	Description
register	Used to save the values of a task execution to some variable for further processing within the Ansible playbook.
loop (with_items)	Used to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached.
when	Used to execute a task in the event that a certain condition happens (typically to compare variables with values).
ignore_errors	Used to continue the execution of a playbook for all nodes, even if the execution of certain task fails.

The key outcome of the execution of any Ansible module is the result achieved by the execution of the task. Different modules have different purposes and therefore different results. The following are a few possible module results:

• **iosxr_command**: Gets output from the requested **show** command.

• **debug**: Prints the content of some variable to verify the correct execution of previous modules.

• **copy**: Copies a Linux file from the source to the destination path.

• **docker_container**: Creates, deletes, or modifies a Docker container.

• **netconf_config**: Configures a network function using NETCONF protocol.

If you look back at Figure 19-2, you can see that return values are an important part of a module. Return values are highly dependent on the type of module. For example, if the result of a module should include some information (such as after issuing a **show** command on a Cisco IOS XR router), this result is be stored in **stdout** as a whole and **stdout_lines** per line, as explained earlier in this chapter. If a module isn't expected to have any output, both of those keys will be missing. However, there are two keys that exist for every module: **changed** and **failed**. Both of these keys are of Boolean type, which means they can have true or false values only. These keys help you build nonlinear logic in a playbook. You might use them, for example, if you need to execute some tasks in the event that the previous task succeeds and other tasks in the event that it fails.

By now, you should have a good understanding of the structure and operation of Ansible modules in general, as well as the modules **iosxr_command** and **debug** in particular. There are several hundred Ansible modules, and each performs specific tasks. You can find them on the official Ansible website (https://docs.ansible.com), where they are split into multiple categories for ease of searching (see the Figure 19-3).

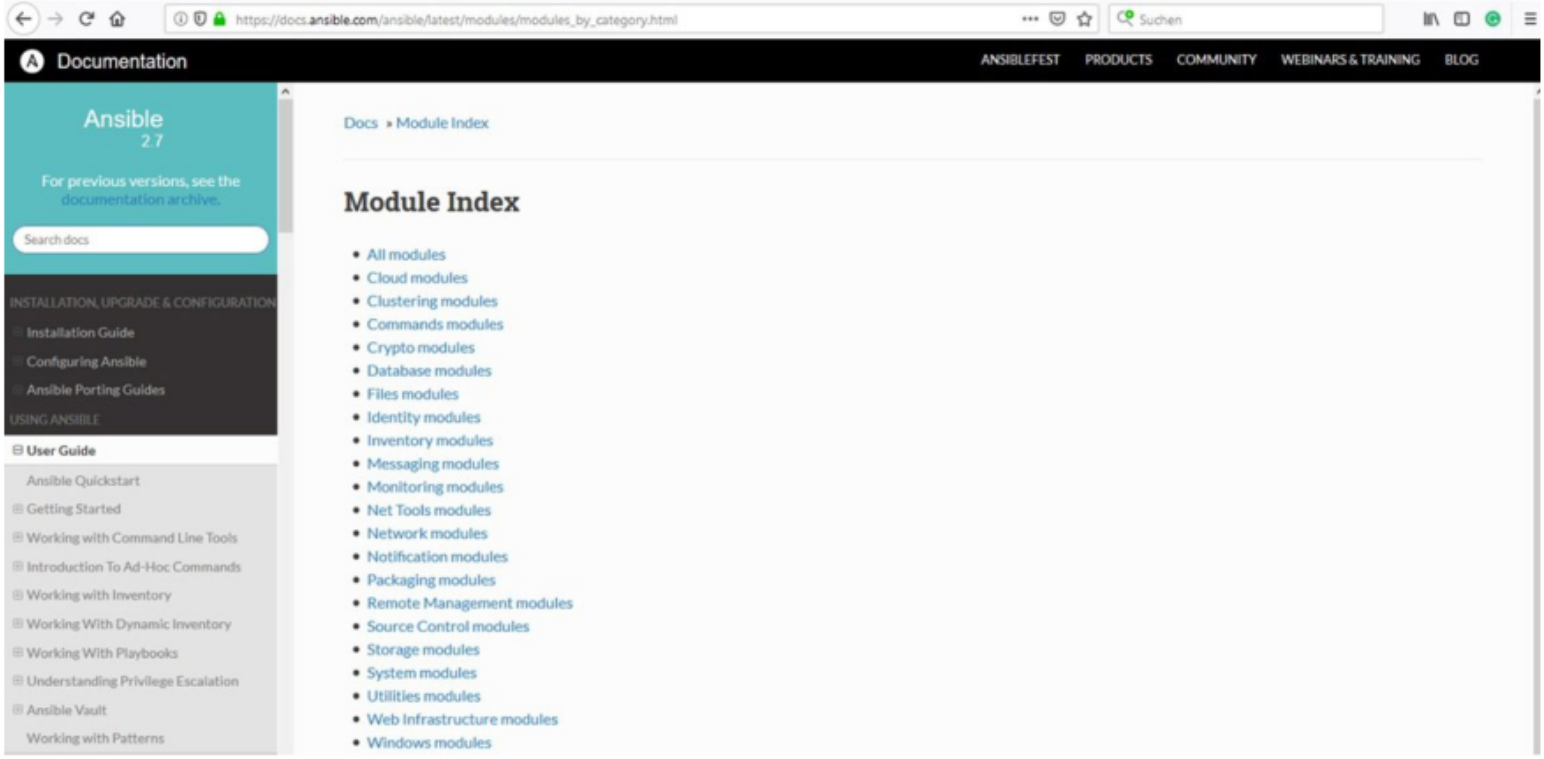

Figure 19-3 *Available Ansible Module Categories*

The modules most widely used for network automation and programming are in the categories network modules, file modules, and utility modules:

Network modules: Modules in this category deal with the network stack in Ansible, including Linux and different vendors and products. Examples include modules for Cisco IOS, IOS XR, NX-OS, ASA, and ACI.

File modules: This category contains modules related to file operations in Linux, such as creating, deleting, copying, and templating.

Utility modules: This category contains modules responsible for the general operation of an Ansible playbook.

Several modules from these categories are covered later in this chapter, in conjunction with specific scenarios.

Extending Ansible Capabilities

As you have learned in this chapter, Ansible modules are building blocks for network automation, but you need some glue to hold these blocks together to assemble a robust network automation solution.

Connection Plugins

Recall from the earlier examples of Ansible ad hoc commands and playbooks that information about the connection type is provided in the argument **–connection=network_cli** or **connection**: **network_cli**. **network_cli** is a connection plug-in that Ansible uses to establish connectivity with a managed node. There are many additional connection plug-ins, and Table 19-3 summarizes the ones that are essential for network automation. From this table, you can see that different connection plug-ins are used in different types of interactions with network functions. The documentation for each module indicates which connectivity plug-in the module uses.

Table 19-3 *Connection Plug-ins*

Plug-in	Protocol	Requires	Persistent?	When Used
network_cli	CLI over SSH	**network_os** setting	Yes	With the **iosxr_command** module for SSH communication
netconf	XML over SSH	**network_os** setting	Yes	With NETCONF modules
httpapi	API over HTTP/HTTPS	**network_os** setting	Yes	With connectivity between a network or control function established over HTTP/HTTPS, as with RESTCONF or GRPC
napalm	NAPALM drivers over SSH	**network_os** setting	Yes	This connection plugin provides connectivity to network devices using the NAPALM network device abstraction library.
local	Depends on additional parameters, if defined	Additional parameters, if defined	No	To process information locally on a Linux host where Ansible is installed (no longer recommended for network function management)

An important point regarding connection plug-ins requires further clarifications. The **persistence** parameter describes how the session between the remote network device and local node is treated. If persistence has been configured, then a single SSH or HTTP/HTTPS session (depending on the connection plug-in) is established for the duration of the whole playbook execution. If persistence has not been configured, for every task in an Ansible playbook, the session has to be reestablished, and this requires the user to enter login and password information if they weren't previously saved. Even if they were saved, the operation for building and closing an SSH session requires additional computation resources. In contrast, SSH or HTTPS sessions are persistent for the duration of the playbook execution.

Another important point is that for some connection plugins, the **ansible_network_os** variable should be defined within the playbook/ad hoc command or in some variables or facts. (You will learn about this shortly.) Sometimes you don't need to define the **ansible_network_os** variable, but typically it is a good idea to define it because doing so ensures that the proper escape characters or prompt matches are used for the particular platform. In the context of this book, for Cisco devices, the well-defined types are **ios** for Cisco IOS XE, **iosxr** for Cisco IOS XR, and **nexus** for Cisco NX-OS.

Variables and Facts

Variables are some of the most critical elements of any programming or scripting logic. Defining variables is the only way you can change input parameters for some actions performed by Ansible playbooks without changing the playbooks themselves. Ansible uses Jinja2 format to define variables. (You will learn about Jinja2 later in this chapter, in the context of the templates.)

Earlier in this chapter, in Example 19-10, you saw a variable used to save the output of the execution of some commands; you might want to do this, for example, if the logic of an automation is built on the results of the previous tasks. This is one of the ways you can assign and use variables, but much more often you use variables to assign parameters that are used later in Ansible playbook tasks for configuration or information collection activities on network functions. Example 19-11 shows the most common way to define variables and assign values to

them.

Example 19-11 *Defining Variables in an Ansible Playbook*

```
$ cat npf_example_simple_vars.yml
---
- hosts: linux
  gather_facts: no
  connection: local

  vars:
      interface:
          name: gig0/0/0/0
          mtu: 1500
          ip: 10.0.0.1/24
      bandwidth: 1000000
      description: Interface to Internet

  tasks:
      - name: CHECKING VARIABLES
        debug:
            msg: Interface {{ interface.name }} has IP address {{ interface.ip }}, MTU
{{ interface.mtu }}, BW {{ bandwidth }} and described as '{{ description }}'
...



$ ansible-playbook npf_example_simple_vars.yml --limit=localhost


PLAY [linux]
*******************************************************************************


TASK [CHECKING VARIABLES]
*******************************************************************
ok: [localhost] => {
    "msg": "Interface gig0/0/0/0 has IP address 10.0.0.1/24, MTU 1500, BW 1000000 and
described as 'Interface to Internet'"
}


PLAY RECAP
********************************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0
```

The variables are defined in the section **vars** as ordinary objects, and when they are called later on in the tasks, they are formatted in the form **{{** *variable* **}}**. In the **vars** part, the variables can be grouped in a hierarchy (for example, **{{ interface.name }}**, **{{ interface.mtu }}**, and **{{ interface.ip }}**). There is no limit to the hierarchy depth, so you should use common sense when creating a hierarchy. Alternatively, they can be treated as independent elements, (for example, **{{ bandwidth }}** and **{{ description }}**). These two ways to define variables (structured and flat) are shown in Example 19-11 to familiarize you both options. In real life, you are likely to the one format that best satisfies your requirements.

Another important component of a playbook is the **gather_facts** key, which is set to the value **no**. This setting determines whether the playbook will collect facts from the destination host before running the playbook's tasks. The facts are additional information about the managed host that could help in the execution of the playbook if you make it vendor or system agnostic. Ansible originally came from the Linux management world, where the different Linux distributions have different packages (for example, **apt-get** to install packages in Ubuntu/Debian, **yum** in CentOS/RedHat up to version 8, **dnf** in CentOS 8). Knowing which Linux distro will be managed by a playbook can help you identify the proper command. Keep in mind that this mechanism for Linux operating systems and is not applicable to network operating systems such as Cisco IOS XE and IOS XR and Juniper Junos OS.

Finally, in Example 19-11, you can see that the argument **--limit** has the value **localhost**. You use the **--limit** argument when you need to limit the scope of the playbook execution (for example, for only a specific group or a specific host). For the value of this argument, you need to provide the name of the host (or multiple hosts, separated by commas) from the inventory file.

By default, you don't define variable types because the interpreter understands the types, based on the context. There are five the main variable types, as shown in Table 19-4.

Table 19-4 *Ansible Variable Types*

Type	Examples	Description
Number	**1**, **23.4**, …	Any numerical value, including floats and negative values
Boolean	**true**, **false**	Binary logic, when there only one of two results is possible
String	**abc**, **interface**, **a3**, …	Any type of information
List	*value1*, *value2*	Multiple entries of any other type, which are siblings to each other
Dictionary	*nested_key*: *nested_value*	Any nested key/value pairs (There is no limit for nesting.)

Defining variables within an Ansible playbook is possible, but it is not necessarily the best way to use variables. Typically you want to use as much automation as possible for massive-scale operations that repeat the same activities several times or several hundred times. For example, say that you need to configure a new data center fabric with IP addresses and external BGP sessions to build a leaf/spine fabric. The set of actions will be the same for each network device that is to be configured, but the network devices have unique IP addresses and BGP peers. It's straightforward to implement such logic in Ansible by using some built-in variables and importing per node variables from an external file, as shown in Example 19-12.

Example 19-12 *Importing Variables from an External File*

```
$ ls -l | grep 'npf'

-rw-------. 1 aaa aaa 418 Dec  1 14:08 npf_example_external_vars.yml

-rw-------. 1 aaa aaa 103 Dec  1 14:09 npf_NX1_variables.yml

-rw-------. 1 aaa aaa 103 Dec  1 14:09 npf_NX2_variables.yml



$ cat npf_NX1_variables.yml

---

interface:

    name: Eth1

    mtu: 1500

    ip: 10.0.0.1/24

    description: Connection to NX2

...



$ cat npf_NX2_variables.yml

---

interface:

    name: Eth2

    mtu: 1500

    ip: 10.0.0.2/24

    description: Connection to NX1

...



$ cat npf_example_external_vars.yml

---

- hosts: nexus

  gather_facts: no

  connection: local


  tasks:

      - name: IMPORTING VARS

        include_vars:

            file: npf_{{ inventory_hostname }}_variables.yml


      - name: CHECKING VARIABLES

        debug:

            msg: Interface {{ interface.name }} has IP address {{ interface.ip }} and

described as '{{ interface.description }}'

...
```

```
$ ansible-playbook npf_example_external_vars.yml

PLAY [nexus]
*******************************************************************************

TASK [IMPORTING VARS]
*********************************************************************
ok: [NX1]
ok: [NX2]

TASK [CHECKING VARIABLES]
********************************************************************
ok: [NX1] => {
    "msg": "Interface Eth1 has IP address 10.0.0.1/24 and described as 'Connection to
NX2'"
}
ok: [NX2] => {
    "msg": "Interface Eth2 has IP address 10.0.0.2/24 and described as 'Connection to
NX1'"
}

PLAY RECAP
********************************************************************************
NX1                        : ok=2    changed=0    unreachable=0    failed=0
NX2                        : ok=2    changed=0    unreachable=0    failed=0
}
```

The first variable in Example 19-12 is the predefined variable **{{ inventory_hostname }}**. This variable is always defined when the playbook is launched and is equal to the value contained in the inventory file associated with the hostname of the network function (refer to Example 19-4). You don't need to define it manually anywhere, but you use it a lot when building automation. As shown in Example 19-12, this variable is used to construct the proper name of the file with variables that should be imported.

Example 19-12 also introduces the module **include_vars**. As indicated by its name, this module reads the variables from an external file so that you can use it in your main playbook. The **file** parameter defines the path file, and both relative and absolute paths can be provided. In the case of a relative path, the lookup is done from the perspective of the folder from which you execute the Ansible playbook. For each network function, the playbook is executed in the appropriate context, so even though the same variable names appear in the playbook, there are no problems, and the proper values are used.

So far, you have learned two ways to define variables: in dedicated files or directly in a playbook. There is one more way to predefine variables, as shown in the Example 19-13, and it relies on the inventory file.

Example 19-13 *Defining Variables in the Inventory File*

```
$ cat /etc/ansible/hosts
[linux]
localhost

[ios]
CSR1
CSR2

[iosxr]
XR3
XR4

[nexus]
NX1 interface=Eth1 ip=10.0.0.1/24
NX2 interface=Eth2 ip=10.0.0.2/24
```

```
[nexus:vars]
ansible_network_os=nexus
ansible_user=aaa
ansible_ssh_pass=aaa
```

```
$ cat npf_example_inventory_vars.yml
---
- hosts: nexus
  gather_facts: yes
  connection: local

  tasks:
      - name: CHECKING VARIABLES
        debug:
            msg: Device {{ inventory_hostname }} is running OS '{{ ansible_network_os
}}' and has interface {{ interface }} with IP address {{ ip }}
...
```

```
$ ansible-playbook npf_example_inventory_vars.yml

PLAY [nexus]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [NX2]
ok: [NX1]

TASK [CHECKING VARIABLES]
********************************************************************
ok: [NX1] => {
    "msg": "Device NX1 is running OS 'nexus' and has interface Eth1 with IP address
10.0.0.1/24"
}
ok: [NX2] => {
    "msg": "Device NX2 is running OS 'nexus' and has interface Eth2 with IP address
10.0.0.2/24"
}

PLAY RECAP
********************************************************************************
NX1                        : ok=2    changed=0    unreachable=0    failed=0
NX2                        : ok=2    changed=0    unreachable=0    failed=0
```

Two classes of variables are used in Example 19-13:

• **group_vars**: These variables are relevant for all the devices in a specific group. In Example 19-13, they are stored in **[nexus:vars]**. This is quite a convenient way to store credentials or other general information. You can see in Example 19-13 that all the variables start with **ansible_**. **group_vars** variable names are typically predefined and have some default values, as different modules use them. You need to manually change those default values to allow modules to operate according to your needs.

• **host_vars**: These variables are relevant only for a particular host. In Example 19-13, these values are stored in the same line with the host itself in *key=value* format and divided by spaces.

There is one more way to define variables, and this method is very important for building nonlinear logic in an application. With this dynamic option, the value of a variable is defined during the execution of the Ansible playbook. Say that you want to check the version of Cisco IOS XR software running on your nodes, compare it to the target release, and automatically update it (or at least notify the operation engineers) in the event that the original software version deviates from the target. Example 19-14 shows how to set up the variable value from the output of a

command.

Example 19-14 *Setting Variables Dynamically*

```
$ cat npf_iosxr_check_version.yml
---
- hosts: iosxr
  gather_facts: yes
  connection: network_cli

  tasks:
      - name: Collect version from Cisco IOS XR
        iosxr_command:
            commands:
                - show version | include Cisco IOS XR
        register: show_output
        ignore_errors: yes

      - name: debug
        debug:
            msg: The full output is {{ show_output.stdout }}

      - name: Create variable with SW value
        set_fact:
            cisco_sw: "{{ show_output.stdout | string | regex_replace('^.*Version
(.+)\\[.*$', '\\1')}}"

      - name: debug
        debug:
            msg: Device {{ inventory_hostname }} is running Cisco IOS XR {{ cisco_sw }}
...


$ ansible-playbook npf_iosxr_check_version.yml

PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR1]
ok: [XR2]

TASK [Collect version from Cisco IOS XR]
*****************************************************
ok: [XR1]
ok: [XR2]

TASK [debug]
*******************************************************************************
ok: [XR2] => {
    "msg": "The full output is [u'Cisco IOS XR Software, Version 6.1.4[Default]']"
}
ok: [XR1] => {
    "msg": "The full output is [u'Cisco IOS XR Software, Version 6.1.4[Default]']"
```

```
}

TASK [Create variable with SW value]
*******************************************************
ok: [XR2]
ok: [XR1]

TASK [debug]
*******************************************************************************
ok: [XR2] => {
    "msg": "Device XR2 is running Cisco IOS XR 6.1.4"
}
ok: [XR1] => {
    "msg": "Device SR1 is running Cisco IOS XR 6.1.4"
}

PLAY RECAP
*******************************************************************************
XR1                        : ok=5    changed=0    unreachable=0    failed=0
XR2                        : ok=5    changed=0    unreachable=0    failed=0
```

In Example 19-14, the variable is created using the module **set_fact**, which is used as a task opposite the **register** instruction, which saves the output of the execution of other modules. Within a module, you can dynamically define any number of variables you need in the form *variable*:*key*. In Example 19-14, the variable **cisco_sw** is created, and some value is assigned to it. The value is extracted from the output of the **show** command saved in the variable **show_output.stdout**. After the name of the variable, you see the rather complex construction **| string | regex_replace('^.*Version (.+)\\[.*$', '\\1')**. This construction applies two filters to the initial variable. We will look more closely at filters in the next section, but for now, you need to understand only how this particular construction works. The filter **string** is applied to an object and converts hat object to a string, no matter how the variable was formatted before. The second filter, **regex_replace**, uses regular expressions to find interesting information and modify the original text. It uses standard Linux regular expressions like the ones discussed in Chapter 4, "Linux Scripting." It uses the syntax **regex_replace('***input_value***', '***output_value***')**, much like the **sed***///* function in Linux.

In this section, you have learned several ways to define variables, depending on your needs:

• Using the **register** instruction to save the result of the task execution

• Using the **vars** section in a playbook

• Using the **set_fact** module

• Using the inventory file

There are no good and bad ways to define variables; the method used depends on the context.

Filters

In general, *filters* are used to manipulate data within an expression. Ansible uses both its own filters and Jinja2 filters. As with Ansible modules, the number of filters available is rather significant, and filters can perform a tremendous number of actions.

Example 19-14 shows a filter applied to a variable's value using the **{{** *variable* **|** *filter* **}}** syntax, where the number of filters is not limited. Each new filter is added as **|** *filter*. The filters are applied sequentially one after another, so you need to pay attention to the order of the filters.

There are so many filters available that it can be difficult to foresee which ones you will use in your automation tasks. To get an idea of what filters can do for you, this section focuses on two filters that are commonly used in networking. The first of these filters allows you to generate random MAC address, and the second one provides a very flexible way of working with IP addresses. However, before you can create an Ansible playbook that uses a filter, you need to install the filter. Example 19-15 shows how to install the two filters we examine in this section.

Example 19-15 *Installing the ipaddr library$ sudo yum install -y python-netaddr*

```
! The output is truncated for brevity
Resolving Dependencies
--> Running transaction check
---> Package python-netaddr.noarch 0:0.7.5-9.el7 will be installed
--> Finished Dependency Resolution
! The output is truncated for brevity
Installed:
  python-netaddr.noarch 0:0.7.5-9.el7
Complete!
```

```
$ sudo pip install ipaddr
```

```
Collecting ipaddr

  Downloading https://files.pythonhosted.org/packages/9d/a7/1b39a16cb90dfe491f57e1cab3103a15d4e8dd9a1

50872744f531b1106c1/ipaddr-2.2.0.tar.gz

Installing collected packages: ipaddr

  Running setup.py install for ipaddr ... done

Successfully installed ipaddr-2.2.
```

After these filters have been installed, you can start using them with IP addresses. You can use the **ipaddr** and **random_mac** filters in any virtual environment where MAC addresses of hosts must deviate from each other and, therefore, can be random, and IP addresses must be consistent within the address allocation schema. Example 19-16 shows these two filters being used in such scenario.

Example 19-16 *Using the **ipaddr** and **random_mac** Filters*

```
$ cat npf_example_simple_vars_and_filters.yml

---

- hosts: nexus

  gather_facts: yes

  connection: local


  vars:

      interface:

          name_prefix: Eth

          mac_prefix: 00:00:5E

          ip_prefix: 10.0.0.0/24


  tasks:

      - name: CHECKING VARIABLES

        debug:

            msg: |

                interface {{ interface.name_prefix }}1

                    mac {{ interface.mac_prefix | random_mac }}

                    ip address {{ interface.ip_prefix | ipaddr('1')}}

...



$ ansible-playbook npf_example_simple_vars_and_filters.yml


PLAY [nexus]

*******************************************************************************


TASK [Gathering Facts]

*********************************************************************

ok: [NX2]

ok: [NX1]


TASK [CHECKING VARIABLES]

*******************************************************************

ok: [NX1] => {

    "msg": "interface Eth1\n    mac 00:00:5e:20:ce:a6\n    ip address 10.0.0.1/24\n"

}

ok: [NX2] => {

    "msg": "interface Eth1\n    mac 00:00:5e:25:13:42\n    ip address 10.0.0.1/24\n"

}


PLAY RECAP

*******************************************************************************

NX1                        : ok=2    changed=0    unreachable=0    failed=0
```

```
NX2                        : ok=2    changed=0    unreachable=0    failed=0
```

Before we discuss these two filters, let's return for a moment to the **debug** module. In Example 19-16, the **msg** argument of the **debug** module is provided in a different format, where it starts with a pipe **|** and then the text is provided as a multi-line object. This approach is the basis for template-based configuration, which is a cornerstone of massive-scale operations. (Templates are discussed in detail later in this chapter.)

The first filter in Example 19-16, **random_mac**, is used to generate the random host part of a MAC address with a given fixed OUI. As you can see, the MAC addresses for nodes NX1 and NX2 are entirely different. The second filter in this example, **ipaddr**, is applied with the argument 1, which in this context means it takes the second IP address from the initial subnet 10.0.0.0/24, which is why you see 10.0.0.1/20. Why does it take the second IP address? As you might know, in programming languages, the index of the first element in an array is always 0, so the argument 1 tells the filter to take from the array the element with index 1—that is, the second element. At this point, you might also wonder what use this filter is if both network functions NX1 and NX2 have the same IP address. You will learn more about this later in this chapter, in the section "Loops."

Conditionals

Conditionals make it possible to create if-then logic in automation, giving you the ability to react to input information with higher intelligence. In Ansible, conditionals allow you to create useful and flexible automation.

Ansible conditionals follow the standard logic of comparing one value with another one. Typically at least one value in the comparison comes from a variable because it does not usually make sense to compare one fixed value with another one. In the discussion of Example 19-14, we talked about checking the software version of the Cisco IOS XR router, but we did not look at how to compare the value collected with some other values. Example 19-17 shows how to do that.

Example 19-17 *Using a Conditional Statement in Ansible, Part 1*

```
$ cat npf_iosxr_check_version.yml
---
- hosts: iosxr
  gather_facts: yes
  connection: network_cli

  vars:
      target_cisco_sw: 6.5.1

  tasks:
      - name: Collect version from Cisco IOS XR
        iosxr_command:
            commands:
                - show version | include Cisco IOS XR
        register: show_output
        ignore_errors: yes

      - name: Create variable with SW value
        set_fact:
            cisco_sw: "{{ show_output.stdout | string | regex_replace('^.*Version
(.+)\\[.*$', '\\1')}}"

      - name: debug
        debug:
            msg: SW upgrade is needed! Device {{ inventory_hostname }} is running Cisco
IOS XR {{ cisco_sw }}, target release is {{ target_cisco_sw }}
        when: cisco_sw != target_cisco_sw

      - name: debug
        debug:
            msg: SW is actual, no actions are needed! Device {{ inventory_hostname }}
is running Cisco IOS XR {{ cisco_sw }}, target release is {{ target_cisco_sw }}
        when: cisco_sw == target_cisco_sw
...

$ ansible-playbook npf_iosxr_check_version.yml
```

```
PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR1]
ok: [XR2]

TASK [Collect version from Cisco IOS XR]
****************************************************
ok: [XR1]
ok: [XR2]

TASK [Create variable with SW value]
*******************************************************
ok: [XR2]
ok: [XR1]

TASK [debug]
*******************************************************************************
ok: [XR2] => {
    "msg": "SW update is needed! Device XR2 is running Cisco IOS XR 6.1.4, target
release is 6.5.1"
}
ok: [XR1] => {
    "msg": "SW update is needed! Device XR1 is running Cisco IOS XR 6.1.4, target
release is 6.5.1"
}

TASK [debug]
*******************************************************************************
skipping: [XR2]
skipping: [XR1]

PLAY RECAP
********************************************************************************
XR2        : ok=4    changed=0    unreachable=0    failed=0
XR1        : ok=4    changed=0    unreachable=0    failed=0
```

At the beginning of Example 19-17, you see the variable **target_cisco_sw** defined with the value **6.5.1**, which is a string. Then the **set_fact** module is used to extract the Cisco IOS XR software version from the output of the **show** command. At the end of the automation script, based on the condition **when**, either one action or another is taken: Either a software update is needed, or it isn't. The comparison is made using syntax that is standard in many programming languages, where **==** means equal and **!=** means not equal.

As you can see in Example 19-17, the extracted value **6.1.4** is not equal to the predefined value **6.5.1**, so you are informed that an update is necessary. The task in the playbook (which isn't played in this case due to the condition) is **skipping**. Now, if the value of **target_cisco_sw** is changed to **6.1.4**, you see a different action occur, as shown in Example 19-18.

Example 19-18 *Using a Conditional Statement in Ansible, Part 2*

```
$ cat npf_iosxr_check_version.yml | grep 'target_cisco_sw:'
      target_cisco_sw: 6.1.4



$ ansible-playbook npf_iosxr_check_version.yml
! The output is truncated for brevity
TASK [debug]
*******************************************************************************
skipping: [XR1]
```

```
skipping: [XR2]


TASK [debug]
*******************************************************************************
ok: [XR1] => {
    "msg": "SW is actual, no actions are needed! Device XR1 is running Cisco IOS XR
6.1.4, target release is 6.1.4"
}
ok: [XR2] => {
    "msg": "SW is actual, no actions are needed! Device XR2 is running Cisco IOS XR
6.1.4, target release is 6.1.4"
}


PLAY RECAP
*******************************************************************************
XR2        : ok=4    changed=0    unreachable=0    failed=0
XR1        : ok=4    changed=0    unreachable=0    failed=0
```

When you are working with numbers, you can perform the Jinja2 comparison operations listed in Table 19-5.

Table 19-5 *Jinja2 Comparison Operations That Are Used in Ansible*

Operator	Action
==	Compares two objects for equality
!=	Compares two objects for inequality
>	True if the left side is greater than the right side
>=	True if the left side is greater than or equal to the right side
<	True if the left side is less than the right side
<=	True if the left side is less than or equal to the right side

In addition to that, in Ansible automation, you will often use two handy conditional operators that are not technically math operators. The first of these operators is used to perform a test, such as to verify whether some variable exists, as shown in Example 19-19.

Example 19-19 *Using a Conditional Statement to Check Whether a Variable Exists*

```
$ cat npf_iosxr_check_version.yml
---
- hosts: iosxr
  gather_facts: yes
  connection: network_cli


#  vars:
#      target_cisco_sw: 6.1.4


  tasks:
      - name: Check input variables for consistency
        fail:
            msg: There is no target SW defined
        when: target_cisco_sw is not defined


      - name: Collect version from Cisco IOS XR
        iosxr_command:
            commands:
                - show version | include Cisco IOS XR
        register: show_output
        ignore_errors: yes


      - name: Create variable with SW value
        set_fact:
            cisco_sw: "{{ show_output.stdout | string | regex_replace('^.*Version
```

```
(.+)\\[.*$', '\\1')}}"


      - name: debug

        debug:

            msg: SW update is needed! Device {{ inventory_hostname }} is running Cisco
IOS XR {{ cisco_sw }}, target release is {{ target_cisco_sw }}

        when: cisco_sw != target_cisco_sw


      - name: debug

        debug:

            msg: SW is actual, no actions are needed! Device {{ inventory_hostname }}
is running Cisco IOS XR {{ cisco_sw }}, target release is {{ target_cisco_sw }}

        when: cisco_sw == target_cisco_sw

...



$ ansible-playbook npf_iosxr_check_version.yml


PLAY [iosxr]
*******************************************************************************


TASK [Gathering Facts]
***********************************************************************

ok: [XR1]

ok: [XR2]


TASK [Check input variables for consistency]
************************************************

fatal: [XR1]: FAILED! => {"changed": false, "msg": "There is no target SW defined"}

fatal: [XR2]: FAILED! => {"changed": false, "msg": "There is no target SW defined"}

        to retry, use: --limit @/home/karneliuka/de-
secgw/ansible/npf_iosxr_check_version.retry


PLAY RECAP
********************************************************************************

XR1        : ok=1    changed=0    unreachable=0    failed=1

XR2        : ok=1    changed=0    unreachable=0    failed=1
```

Example 19-19 shows some lines of the Ansible playbook commented out with the **#** character. Comments allow you to leave notes in code for others or to temporarily deactivate some parts of your code for debugging. Example 19-19 also shows the module **fail**, which stops the execution of the playbook. Because **fail** it stops a playbook, you use it only with some conditions. For example, in Example 19-19, the playbook should fail if the variable **target_cisco_sw** doesn't exist or if the way it is reflected in the code is not defined. The operator **is**, together with the predefined argument **defined**, performs this test by looking at whether such a variable is defined somewhere earlier in the playbook, is imported from another playbook, or comes from group/host variables. Example 19-19 provides an obvious example in which the variable is defined in the same file, and you can easily spot a mistake. However, if the variables are imported from another task, or even if they are dynamically set using the **set_fact** module, based on the output of the execution of some previous parts of the code, the check plays a significant role in saving resources and aiding in the development of the code. Moreover, even in Example 19-18, if you don't do a check, the playbook collects the information from the remote nodes before it fails because the last task can't be finished. Collecting the information from all the remote nodes takes time and effort, and it's best to fail earlier to save that time.

Another important operator for conditions in Jinja2 used in Ansible, the **in** operator, allows you to perform a search in the text. In specific scenarios like our earlier example of software upgrades, this operator allows you to reduce the number of tasks, as shown in Example 19-20.

Example 19-20 *Using a Conditional Statement to Check Whether a Variable Contains Some Substring*

```
$ cat npf_iosxr_check_version.yml

---

- hosts: iosxr

  gather_facts: yes

  connection: network_cli
```

```
  vars:
      target_cisco_sw: 6.1.4

  tasks:
      - name: Collect version from Cisco IOS XR
        iosxr_command:
            commands:
                - show version | include Cisco IOS XR
        register: show_output
        ignore_errors: yes

      - name: debug
        debug:
            msg: SW update is needed! Device {{ inventory_hostname }} is not running SW
{{ target_cisco_sw }}
        when: target_cisco_sw not in show_output.stdout | string

      - name: debug
        debug:
            msg: SW is actual, no actions are needed! Device {{ inventory_hostname }}
is running SW {{ target_cisco_sw }}
        when: target_cisco_sw in show_output.stdout | string
...
```

```
$ ansible-playbook npf_iosxr_check_version.yml

PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR1]
ok: [XR2]

TASK [Collect version from Cisco IOS XR]
****************************************************
ok: [XR2]
ok: [XR1]

TASK [debug]
*******************************************************************************
skipping: [XR1]
skipping: [XR2]

TASK [debug]
*******************************************************************************
ok: [XR1] => {
    "msg": "SW is actual, no actions are needed! Device XR1 is running SW 6.1.4"
}
ok: [XR2] => {
    "msg": "SW is actual, no actions are needed! Device XR2 is running SW 6.1.4"
}
```

```
PLAY RECAP
*********************************************************************************
XR1                        : ok=3    changed=0    unreachable=0    failed=0
XR2                        : ok=3    changed=0    unreachable=0    failed=0
```

The conditional operator **in** is used to find a match, and **not in** indicates that there is not a match. In a nutshell, this operator takes the value of the variable on the left side from the keyword **in** (in this case, **target_cisco_sw**) and tries to find a substring match at the beginning, middle, or end of the variable on the right side (in this case, **show_output.stdout**). If there is a match, then the result of the condition **in** is true; if there is no match, the condition **in** is false. (**not in** works the opposite way.) The filter **string** can be used for proper text processing of the output of the **show** command.

So far, you have learned about different operators used in conditional statements as well as how to apply conditionals to the tasks in Ansible playbooks. The final topic related to conditionals that we need to discuss is related to scenarios where you need to apply several conditionals simultaneously when triggering a single action—for example, to make sure that all conditions are fulfilled together (logical AND) or only partially (logical OR). Example 19-21 shows how to use these conditionals.

Example 19-21 *Using and/or Logic to Combine Multiple Conditionals*

```
$ cat npf_iosxr_check_version.yml
---
- hosts: iosxr
  gather_facts: yes
  connection: network_cli

  vars:
      target_cisco_sw: 6.1.4

  tasks:
      - name: Collect version from Cisco IOS XR
        iosxr_command:
            commands:
                - show version | include Cisco IOS XR
        register: show_output
        ignore_errors: yes

      - name: debug
        debug:
            msg: SW upgrade is needed! Device {{ inventory_hostname }} is not running
SW {{ target_cisco_sw }}
        when: target_cisco_sw not in show_output.stdout | string or 'Cisco' not in
show_output.stdout | string

      - name: debug
        debug:
            msg: SW is actual, no actions are needed! Device {{ inventory_hostname }}
is running SW {{ target_cisco_sw }}
        when: target_cisco_sw in show_output.stdout | string and 'Cisco' in
show_output.stdout | string
...



$ ansible-playbook npf_iosxr_check_version.yml


PLAY [iosxr]
*********************************************************************************


TASK [Gathering Facts]
*********************************************************************
ok: [XR1]
ok: [XR2]
```

```
TASK [Collect version from Cisco IOS XR]
****************************************************
ok: [XR2]
ok: [XR1]

TASK [debug]
*******************************************************************************
skipping: [XR1]
skipping: [XR2]

TASK [debug]
*******************************************************************************
ok: [XR1] => {
    "msg": "SW is actual, no actions are needed! Device XR1 is running SW 6.1.4"
}
ok: [XR2] => {
    "msg": "SW is actual, no actions are needed! Device XR2 is running SW 6.1.4"
}

PLAY RECAP
********************************************************************************
XR1                        : ok=3    changed=0    unreachable=0    failed=0
XR2                        : ok=3    changed=0    unreachable=0    failed=0
```

To execute a task in the event that two or more conditions are fulfilled together, you need to use the keyword **and** to combine the conditions. This is equivalent to configuring routing policies in Cisco IOS with the **match-all** condition. If you want to execute a task in the event that at least one of all the provided conditions are fulfilled, you need to use the keyword **or** to combine the conditions. This is equivalent to using a **match-any** condition in routing policies in Cisco IOS (or NX-OS or IOS XR).

Example 19-21, note in the **when** section that all the variables are provided without any special syntax, like **{{** *variable* **}}**. To distinguish the variables from values (for example, any static text you might need to use), the values (static text) are framed by quotes, in the format *'value'*. In Example 19-21, you can see the value **'Cisco'** results from a search in the variable **show_output.stdout** processed with the filter **string**.

Loops

A *loop* enables you to repeat some actions with slight changes. Loops are crucial in creating templates for configuring network functions and for performing actions on remote devices. Ansible enables you to create loops with a rich set of parameters.

In Example 19-16, you saw an example of how to create a configuration for an interface of a Cisco NX-OS switch device. In that example, you could see that there were some limitations, such as the same IP addresses on the two nodes. Example 19-22 shows how to solve the duplicate IP address problem.

Example 19-22 *Using a Loop and Conditions Together*

```
$ cat npf_example_simple_vars_and_filters.yml
---
- hosts: nexus
  gather_facts: yes
  connection: local

  vars:
      interface:
          name_prefix: Eth
          mac_prefix: 00:00:5E
          ip_prefix: 10.0.0.0/8

  tasks:
      - name: CHECKING VARIABLES
        debug:
            msg: |
                interface {{ interface.name_prefix }}{{ item }}
```

```
                mac {{ interface.mac_prefix | random_mac }}
                ip address {{ interface.ip_prefix | ipaddr(1) }}
        loop:
            - 1/1
            - 2/1
            - 1/7
            - 1/18
        when: inventory_hostname == 'NX1'

      - name: CHECKING VARIABLES
        debug:
            msg: |
                interface {{ interface.name_prefix }}{{ item }}
                mac {{ interface.mac_prefix | random_mac }}
                ip address {{ interface.ip_prefix | ipaddr(2) }}
        loop:
            - 1/3
            - 1/7
            - 2/18
            - 2/23
        when: inventory_hostname == 'NX2'
...
```

```
$ ansible-playbook npf_example_simple_vars_and_filters.yml

PLAY [nexus]
*******************************************************************************

TASK [Gathering Facts]
***********************************************************************
ok: [NX2]
ok: [NX1]

TASK [CHECKING VARIABLES]
*******************************************************************
ok: [NX1] => (item=1/1) => {
    "msg": "interface Eth1/1\n    mac 00:00:5e:75:8f:23\n    ip address 10.0.0.1/8\n"
}
ok: [NX1] => (item=2/1) => {
    "msg": "interface Eth2/1\n    mac 00:00:5e:10:4d:f9\n    ip address 10.0.0.1/8\n"
}
ok: [NX1] => (item=1/7) => {
    "msg": "interface Eth1/7\n    mac 00:00:5e:11:3f:b6\n    ip address 10.0.0.1/8\n"
}
skipping: [NX2] => (item=1/1)
ok: [NX1] => (item=1/18) => {
    "msg": "interface Eth1/18\n    mac 00:00:5e:b9:56:59\n    ip address 10.0.0.1/8\n"
}
skipping: [NX2] => (item=2/1)
skipping: [NX2] => (item=1/7)
skipping: [NX2] => (item=1/18)
skipping: [NX2]
```

```
TASK [CHECKING VARIABLES]
*********************************************************************
skipping: [NX1] => (item=1/3)
skipping: [NX1] => (item=1/7)
skipping: [NX1] => (item=2/18)
skipping: [NX1] => (item=2/23)
skipping: [NX1]
ok: [NX2] => (item=1/3) => {
    "msg": "interface Eth1/3\n    mac 00:00:5e:88:75:e2\n    ip address 10.0.0.2/8\n"
}
ok: [NX2] => (item=1/7) => {
    "msg": "interface Eth1/7\n    mac 00:00:5e:bd:b2:05\n    ip address 10.0.0.2/8\n"
}
ok: [NX2] => (item=2/18) => {
    "msg": "interface Eth2/18\n    mac 00:00:5e:54:74:5b\n    ip address 10.0.0.2/8\n"
}
ok: [NX2] => (item=2/23) => {
    "msg": "interface Eth2/23\n    mac 00:00:5e:fd:84:8d\n    ip address 10.0.0.2/8\n"
}


PLAY RECAP
*********************************************************************************
NX1                        : ok=2    changed=0    unreachable=0    failed=0
NX2                        : ok=2    changed=0    unreachable=0    failed=0
```

In Example 19-22, the keyword **loop** introduces a loop after the **task** description. The loop contains a list of variables, and the task is performed sequentially for each item in the list. Actually, **{{** *item* **}}** is also the name of a built-in variable, which takes the value from the active entry in the list. You can see how the conditions play a role here: The list contains different items, depending on the value of **{{** *inventory_hostname* **}}**. This brings us to another critical point: The list of entries in the loop is just a list of variables, which can be defined as any other variables in the playbook or somewhere externally, as shown in the Example 19-23.

Example 19-23 *Using a Loop and Conditions Together with External Variables*

```
$ cat npf_NX1_variables.yml
---
list_of_interfaces:
    - 1/1
    - 2/1
    - 1/7
    - 1/18
...



$ cat npf_NX2_variables.yml
---
list_of_interfaces:
    - 1/3
    - 1/7
    - 2/18
    - 2/23
...



$ cat npf_example_loops_external_vars.yml
---
- hosts: nexus
  gather_facts: yes
```

```
  connection: local

  vars:
      interface:
          name_prefix: Eth
          mac_prefix: 00:00:5E
          ip_prefix: 10.0.0.0/8

  tasks:
      - name: IMPORTING VARS
        include_vars:
            file: npf_{{ inventory_hostname }}_variables.yml

      - name: CHECKING VARIABLES
        debug:
            msg: |
                interface {{ interface.name_prefix }}{{ item }}
                    mac {{ interface.mac_prefix | random_mac }}
                    ip address {{ interface.ip_prefix | ipaddr(1) }}
        loop: "{{ list_of_interfaces }}"
        when: inventory_hostname == 'NX1'

      - name: CHECKING VARIABLES
        debug:
            msg: |
                interface {{ interface.name_prefix }}{{ item }}
                    mac {{ interface.mac_prefix | random_mac }}
                    ip address {{ interface.ip_prefix | ipaddr(2) }}
        loop: "{{ list_of_interfaces }}"
        when: inventory_hostname == 'NX2'
...
```

```
$ ansible-playbook npf_example_loops_external_vars.yml

PLAY [nexus]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [NX1]
ok: [NX2]

TASK [IMPORTING VARS]
**********************************************************************
ok: [NX1]
ok: [NX2]

TASK [CHECKING VARIABLES]
*********************************************************************
ok: [NX1] => (item=1/1) => {
    "msg": "interface Eth1/1\n    mac 00:00:5e:18:91:6e\n    ip address 10.0.0.1/8\n"
}
ok: [NX1] => (item=2/1) => {
```

```
    "msg": "interface Eth2/1\n    mac 00:00:5e:23:64:3f\n    ip address 10.0.0.1/8\n"
}
ok: [NX1] => (item=1/7) => {
    "msg": "interface Eth1/7\n    mac 00:00:5e:44:03:0f\n    ip address 10.0.0.1/8\n"
}
ok: [NX1] => (item=1/18) => {
    "msg": "interface Eth1/18\n    mac 00:00:5e:14:64:72\n    ip address 10.0.0.1/8\n"
}
skipping: [NX2] => (item=1/3)
skipping: [NX2] => (item=1/7)
skipping: [NX2] => (item=2/18)
skipping: [NX2] => (item=2/23)
skipping: [NX2]

TASK [CHECKING VARIABLES]
*********************************************************************
skipping: [NX1] => (item=1/1)
skipping: [NX1] => (item=2/1)
skipping: [NX1] => (item=1/7)
skipping: [NX1] => (item=1/18)
skipping: [NX1]
ok: [NX2] => (item=1/3) => {
    "msg": "interface Eth1/3\n    mac 00:00:5e:13:cc:a9\n    ip address 10.0.0.2/8\n"
}
ok: [NX2] => (item=1/7) => {
    "msg": "interface Eth1/7\n    mac 00:00:5e:22:62:82\n    ip address 10.0.0.2/8\n"
}
ok: [NX2] => (item=2/18) => {
    "msg": "interface Eth2/18\n    mac 00:00:5e:24:7f:20\n    ip address 10.0.0.2/8\n"
}
ok: [NX2] => (item=2/23) => {
    "msg": "interface Eth2/23\n    mac 00:00:5e:10:e1:32\n    ip address 10.0.0.2/8\n"
}

PLAY RECAP
*********************************************************************************
NX1                        : ok=3    changed=0    unreachable=0    failed=0
NX2                        : ok=3    changed=0    unreachable=0    failed=0
```

Example 19-23 continues to use the keyword **loop**, but instead of providing a list of items directly, it calls the variable name **list_of_interfaces**, as defined in the (imported) list with variables.

We need to look at two more aspects of loops. The first aspect is that a list can contain much more information than a single entry, which means you can enrich the configuration of the interface. The second aspect is that there is a built-in counter that calculates the index of the current item. You can use this counter in any scenario you need. Take a look at the structure of the variables in Example 19-24.

Example 19-24 *Using a Loop Index and Complex List Variables*

```
$ cat npf_NX1_variables.yml
---
list_of_interfaces:
    - id: 1/1
      descr: Connectivity to Internet
    - id: 2/1
      descr: Connectivity to NX1
    - id: 1/7
      descr: Connectivity to XR1
    - id: 1/18
```

```
      descr: Connectivity to XR2
...
```

```
$ cat npf_NX2_variables.yml
---
list_of_interfaces:
    - id: 1/3
      descr: Connectivity to Internet
    - id: 1/7
      descr: Connectivity to NX1
    - id: 2/18
      descr: Connectivity to XR1
    - id: 2/23
      descr: Connectivity to XR2
```

Unlike in Example 19-23, in Example 19-24, the interface variables for each network element are now in a list containing dictionaries rather than pure values. Using the same approach as in Example 19-23, those per-device variables can be imported in the playbook during execution, as shown Example 19-25. This creates more powerful automation for configuration of the interfaces for Cisco NX-OS.

Example 19-25 *A Playbook Demonstrating the Use of a Loop Index and Complex List Variables*

```
$ cat npf_example_loops_external_vars.yml
---
- hosts: nexus
  gather_facts: yes
  connection: local

  vars:
      interface:
          name_prefix: Eth
          mac_prefix: 00:00:5E
          ip_prefix: 10.0.0.0/8

  tasks:
      - name: IMPORTING VARS
        include_vars:
            file: npf_{{ inventory_hostname }}_variables.yml

      - name: CHECKING VARIABLES
        debug:
            msg: |
                interface {{ interface.name_prefix }}{{ item.id }}
                    mac {{ interface.mac_prefix | random_mac }}
                    description {{ item.descr }}
                    ip address {{ interface.ip_prefix | ipsubnet(24, loop_index) |
ipaddr(1) }}
        loop: "{{ list_of_interfaces }}"
        loop_control:
            index_var: loop_index
        when: inventory_hostname == 'NX1'

      - name: CHECKING VARIABLES
        debug:
            msg: |
                interface {{ interface.name_prefix }}{{ item.id }}
                    mac {{ interface.mac_prefix | random_mac }}
```

```
                    description {{ item.descr }}

                    ip address {{ interface.ip_prefix | ipsubnet(24, loop_index) |

ipaddr(2) }}

        loop: "{{ list_of_interfaces }}"

        loop_control:

            index_var: loop_index

        when: inventory_hostname == 'NX2'

...
```

In Example 19-25, you still call the loop by using the keyword **loop** followed by the variable **{{ list_of_interfaces }}**. However, because the list has different key/value pairs, you need to call the proper value. You do this by using **{{** *item.key* **}}** syntax (for example, **{{ item.id }}** and **{{ item.descr }}** in Example 19-25). Every item should have the same set of the keys in order to work properly. You can create any further hierarchy if needed.

Another important new detail in Example 19-25 is the keyword **loop_control**, which defines different parameters relevant to the loop, including the parameter **index_var**, which, as the name suggests, is responsible for calculating the index. To use it, you need to assign some variable to it (such as **loop_index** in Example 19-25). Then this variable is used with filter **ipsubnet** to split the original subnet 10.0.0.0/8 into smaller subnets with the /24 prefix length, and the subnet number is defined based on the value of the variable **loop_index**. Example 19-26 shows that now each interface has an IP address within the subnet as well as a different subnet for each interface.

Example 19-26 *Executing the Playbook Containing a Loop Index and Complex List Variables*

```
$ ansible-playbook npf_example_loops_external_vars.yml --inventory=hosts


PLAY [nexus]

*******************************************************************************


TASK [Gathering Facts]

*********************************************************************

ok: [NX2]

ok: [NX1]


TASK [IMPORTING VARS]

*********************************************************************

ok: [NX1]

ok: [NX2]


TASK [CHECKING VARIABLES]

*********************************************************************

skipping: [NX2] => (item={u'id': u'1/3', u'descr': u'Connectivity to Internet'})

skipping: [NX2] => (item={u'id': u'1/7', u'descr': u'Connectivity to NX1'})

skipping: [NX2] => (item={u'id': u'2/18', u'descr': u'Connectivity to XR1'})

skipping: [NX2] => (item={u'id': u'2/23', u'descr': u'Connectivity to XR2'})

skipping: [NX2]

ok: [NX1] => (item={u'id': u'1/1', u'descr': u'Connectivity to Internet'}) => {

    "msg": "interface Eth1/1\n    mac 00:00:5e:ba:2f:2c\n    description Connectivity

to Internet\n    ip address 10.0.0.1/24\n"

}

ok: [NX1] => (item={u'id': u'2/1', u'descr': u'Connectivity to NX1'}) => {

    "msg": "interface Eth2/1\n    mac 00:00:5e:57:4e:6d\n    description Connectivity

to NX1\n    ip address 10.0.1.1/24\n"

}

ok: [NX1] => (item={u'id': u'1/7', u'descr': u'Connectivity to XR1'}) => {

    "msg": "interface Eth1/7\n    mac 00:00:5e:ea:fe:cf\n    description Connectivity

to XR1\n    ip address 10.0.2.1/24\n"

}

ok: [NX1] => (item={u'id': u'1/18', u'descr': u'Connectivity to XR2'}) => {

    "msg": "interface Eth1/18\n    mac 00:00:5e:18:a9:09\n    description Connectivity

to XR2\n    ip address 10.0.3.1/24\n"

}
```

```
TASK [CHECKING VARIABLES]
*******************************************************************
skipping: [NX1] => (item={u'id': u'1/1', u'descr': u'Connectivity to Internet'})
skipping: [NX1] => (item={u'id': u'2/1', u'descr': u'Connectivity to NX1'})
skipping: [NX1] => (item={u'id': u'1/7', u'descr': u'Connectivity to XR1'})
skipping: [NX1] => (item={u'id': u'1/18', u'descr': u'Connectivity to XR2'})
skipping: [NX1]
ok: [NX2] => (item={u'id': u'1/3', u'descr': u'Connectivity to Internet'}) => {
    "msg": "interface Eth1/3\n    mac 00:00:5e:20:06:4e\n    description Connectivity
to Internet\n    ip address 10.0.0.2/24\n"
}
ok: [NX2] => (item={u'id': u'1/7', u'descr': u'Connectivity to NX1'}) => {
    "msg": "interface Eth1/7\n    mac 00:00:5e:ba:b8:8e\n    description Connectivity
to NX1\n    ip address 10.0.1.2/24\n"
}
ok: [NX2] => (item={u'id': u'2/18', u'descr': u'Connectivity to XR1'}) => {
    "msg": "interface Eth2/18\n    mac 00:00:5e:1d:78:30\n    description Connectivity
to XR1\n    ip address 10.0.2.2/24\n"
}
ok: [NX2] => (item={u'id': u'2/23', u'descr': u'Connectivity to XR2'}) => {
    "msg": "interface Eth2/23\n    mac 00:00:5e:17:08:f3\n    description Connectivity
to XR2\n    ip address 10.0.3.2/24\n"
}


PLAY RECAP
*******************************************************************************
NX1                        : ok=3    changed=0    unreachable=0    failed=0
NX2                        : ok=3    changed=0    unreachable=0    failed=0
```

You should now be able to create rather advanced playbooks. So far in this chapter, you have learned how to connect to network devices by using connectivity plug-ins, how to define variables statically or dynamically, and how to use conditions and loops. However, before we look at more advanced cases with Cisco network devices, we need to examine one more crucial mass automation topic: Jinja2 templates.

Jinja2 Templates

Jinja2 is a modern and designer-friendly templating language for Python. Ansible is based on Python, and it heavily uses Jinja2. Actually, you have already seen Jinja2 used a lot. The syntax **{{** *variable* **}}** is used to define a variable in Jinja2, and **{{** *variable* **|** *filter* **}}** is also Jinja2 syntax. However, you need to learn more about Jinja2 and related modules in order to unleash the full power of automation with Ansible.

The Need for Templates

To automate any massive-scale operations, you need templates. Basically, a *template* is a predefined string (or multi-line string) that has fixed parts and variables that change in value based on what is passed to the template.

Cloud services, including AWS, Google Cloud, and THG Hosting, heavily use templates for allocating and deleting users' computing resources. Templates are also used with network devices, as typically the vast majority of the parameters in a configuration are the same across all the network functions in a network, and only a small number of them vary. Just think about the standard router configuration in an enterprise network. Let's say you are running some router protocol, like OSPF or EIGRP. It is likely that timers and areas will have the same parameters, but router IDs and interface numbers will be different. You might think that an enterprise network has a lot of specific configuration for each device, such as aggregations for EIGRP and multiple areas for OSPF, which translates to different flavors of templates. However, even such deviations can be covered using conditions and loops, as you will learn in this section. Another example that shows the power of templating is high-scale data centers and service provider networks, which have very similar configurations on a huge number of network elements.

Figure 19-4 illustrates template-based network configuration.

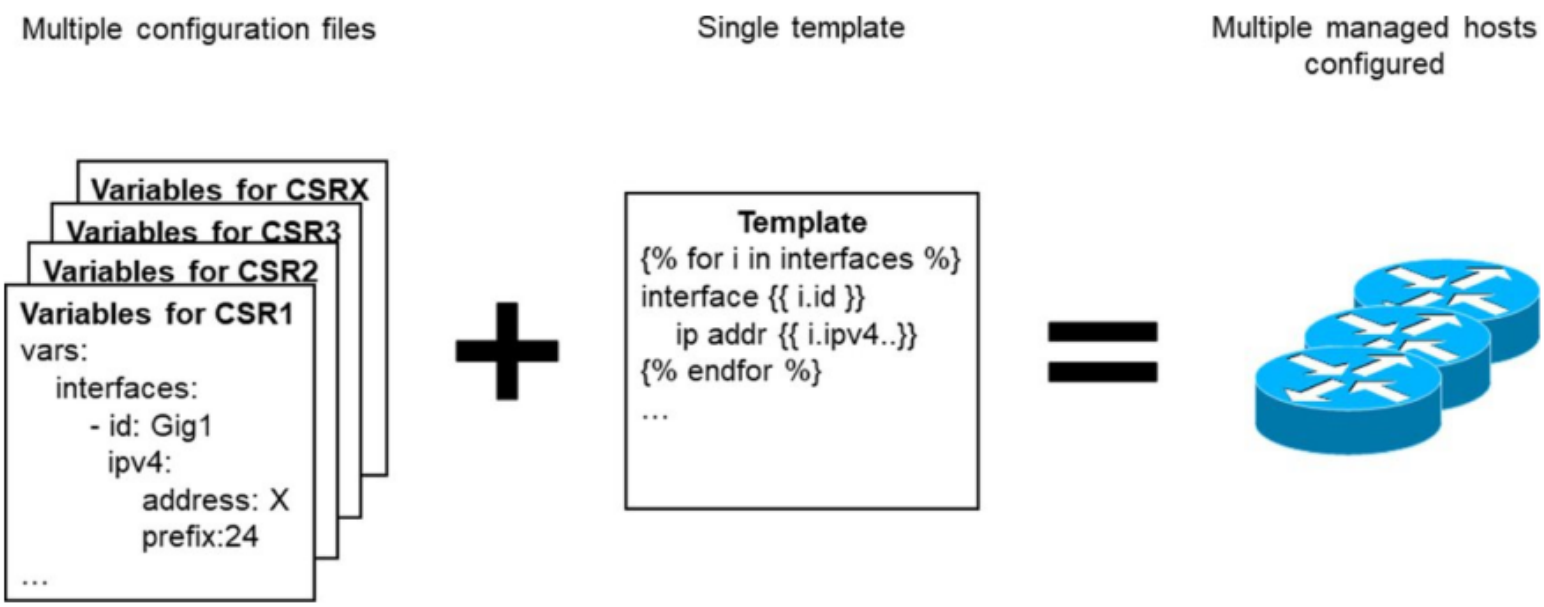

Figure 19-4 *Template-Based Approach for Network Automation*

With Ansible, there are three significant ways to interact with templates, and you are likely to use a mixture of them, depending on the particular use case or the Ansible module you use.

To see the effect of using Ansible templates, let's consider a practical example: adding new entries in /etc/hosts on Linux hosts to ease their management via hostnames. As you have already seen in this chapter, one option with Jinja2 templates is inline templates (refer to Example 19-25). An inline template usually starts with a pipe (|) as a value for some key, such as the key **msg** in the module **debug**, and then it is continued from the new string. Example 19-27 shows the structure of an inline template.

Example 19-27 *Using an Inline Jinja2 Template*

```
$ cat npf_example_inline_template.yml
---
- hosts: linux
  gather_facts: no
  connection: local
  become: yes

  tasks:
      - name: ADDING INFO TO /etc/hosts
        blockinfile:
            dest: /etc/hosts
            marker: "#{MARK} ANSIBLE MANAGED BLOCK {{ item.hostname }}"
            block: |
                {{ item.ip }}   {{ item.hostname }}
        loop:
            - { hostname: XR1, ip: 192.168.1.11 }
            - { hostname: XR2, ip: 192.168.1.12 }
            - { hostname: NX1, ip: 192.168.1.21 }
            - { hostname: NX2, ip: 192.168.1.22 }
...


$ ansible-playbook npf_example_inline_template.yml --ask-become-pass
SUDO password:

PLAY [linux]
*******************************************************************************

TASK [ADDING INFO TO /etc/hosts]
***********************************************************
changed: [localhost] => (item={u'ip': u'192.168.1.11', u'hostname': u'XR1'})
changed: [localhost] => (item={u'ip': u'192.168.1.12', u'hostname': u'XR2'})
```

```
changed: [localhost] => (item={u'ip': u'192.168.1.21', u'hostname': u'NX1'})

changed: [localhost] => (item={u'ip': u'192.168.1.22', u'hostname': u'NX2'})


PLAY RECAP
*********************************************************************************
localhost                  : ok=1    changed=1    unreachable=0    failed=0



$ cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
#BEGIN ANSIBLE MANAGED BLOCK XR1
192.168.1.11   XR1
#END ANSIBLE MANAGED BLOCK XR1
#BEGIN ANSIBLE MANAGED BLOCK XR2
192.168.1.12   XR2
#END ANSIBLE MANAGED BLOCK XR2
#BEGIN ANSIBLE MANAGED BLOCK NX1
192.168.1.21   NX1
#END ANSIBLE MANAGED BLOCK NX1
#BEGIN ANSIBLE MANAGED BLOCK NX2
192.168.1.22   NX2
#END ANSIBLE MANAGED BLOCK NX2
```

Did you notice the new entry **become: yes** at the beginning of Example 19-27? This entry forces the task to be executed from **sudo** mode in Linux; this is necessary because only to user root or another using in **sudo** mode can change the /etc/hosts file.

Now let's take a look at the content on Example 19-27. The Ansible module **blockinfile** adds some text (single-line or multi-line text) from the key **block** to the file defined in **dest**. To make it evident that Ansible has created a specific configuration, it adds the value contained in **marker** before the text insertion and after it. **marker** has a default value, and you need to be careful with it to avoid quickly deleting something you don't mean to delete. The value of **marker** is changed so that it starts and ends with the hostname of the device upon creating the entries in /etc/hosts, so that if the IP address of any host changes, it can be updated. The beginning and the end of the entry are defined by keywords **BEGIN** and **END**, which are automatically placed on the specific variable **{ mark }**.

In Example 19-27, you can also see that several variables are listed in a loop statement, in a single line separated by a comma and framed by curly brackets (**{}**). This syntax differs from the syntax used in Example 19-25, where the key/value pairs are written once per line, but the meaning is the same. In Example 19-27, the template is created directly in the playbook, where it is used; this might be suboptimal with a long template having tens or hundreds of lines. There are alternatives to this approach, as described later in this section.

Another option is to put the Jinja2 template in a separate file and call it by using a plug-in, as shown in Example 19-28.

Example 19-28 *Using a Jinja2 Template from an External File and the **lookup** Plug-in*

```
$ cat npf_example_external_template.j2
{{ item.ip }}   {{ item.hostname }}



$ cat npf_example_external_template.yml
---
- hosts: linux
  gather_facts: no
  connection: local
  become: yes

  tasks:
      - name: ADDING INFO TO /etc/hosts
        blockinfile:
            dest: /etc/hosts
            marker: "#{mark} ANSIBLE MANAGED BLOCK {{ item.hostname }}"
            block: "{{ lookup('template', 'npf_example_external_template.j2') }}"
        loop:
            - { hostname: XR1, ip: 192.168.1.11 }
            - { hostname: XR2, ip: 192.168.1.12 }
```

```
            - { hostname: NX1, ip: 192.168.1.21 }
            - { hostname: NX2, ip: 192.168.1.22 }
...
```

```
$ ansible-playbook npf_example_inline_template.yml --ask-become-pass
SUDO password:

PLAY [linux]
*******************************************************************************

TASK [ADDING INFO TO /etc/hosts]
***********************************************************
changed: [localhost] => (item={u'ip': u'192.168.1.11', u'hostname': u'XR1'})
changed: [localhost] => (item={u'ip': u'192.168.1.12', u'hostname': u'XR2'})
changed: [localhost] => (item={u'ip': u'192.168.1.21', u'hostname': u'NX1'})
changed: [localhost] => (item={u'ip': u'192.168.1.22', u'hostname': u'NX2'})

PLAY RECAP
*******************************************************************************
localhost                  : ok=1    changed=1    unreachable=0    failed=0
```

```
$ cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
#BEGIN ANSIBLE MANAGED BLOCK XR1
192.168.1.11   XR1
#END ANSIBLE MANAGED BLOCK XR1
#BEGIN ANSIBLE MANAGED BLOCK XR2
192.168.1.12   XR2
#END ANSIBLE MANAGED BLOCK XR2
#BEGIN ANSIBLE MANAGED BLOCK NX1
192.168.1.21   NX1
#END ANSIBLE MANAGED BLOCK NX1
#BEGIN ANSIBLE MANAGED BLOCK NX2
192.168.1.22   NX2
#END ANSIBLE MANAGED BLOCK NX2
```

A Jinja2 template is stored in a file with a *.j2 extension; this means you can start directly with any text or variable, and no specific syntax is needed to define that it's a template. You can call a Jinja2 template from an Ansible playbook by using the module **lookup**; Example 19-28 shows a template being called with **{{ lookup('template', 'npf_example_external_template.j2') }}**. The path to a template can be either relative or absolute. In the example provided, the path is relative as the template is located in the same folder as the playbook.

You can instead use absolute path (for example, **lookup('template', '/home/user/templates/npf_example_external_template.j2')**) if that is more beneficial with your playbook. The main advantage of using an absolute path is that when templates are stored in a separate file, in the playbook you can focus on the logic of the execution, and in the separate template file you focus only on the quality of the template. In addition, this approach is advantageous with large templates.

A third option is to use a specific Ansible module to fill in the template with proper data and render it in the finale file, as shown in Example 19-29.

Example 19-29 *Using a Specific Ansible Module for Templating*

```
$ cat npf_example_external_template.j2
{{ item.ip }}   {{ item.hostname }}
```

```
$ cat npf_example_template_module.yml
---
- hosts: linux
```

```
  gather_facts: no
  connection: local

  tasks:
      - name: ADDING INFO TO /etc/hosts
        template:
            src: npf_example_external_template.j2
            dest: hosts_{{ item.hostname }}.txt
        loop:
            - { hostname: XR1, ip: 192.168.1.11 }
            - { hostname: XR2, ip: 192.168.1.12 }
            - { hostname: NX1, ip: 192.168.1.21 }
            - { hostname: NX2, ip: 192.168.1.22 }
...
```

```
$ ansible-playbook npf_example_template_module.yml

PLAY [linux]
*******************************************************************************

TASK [ADDING INFO TO /etc/hosts]
************************************************************
changed: [localhost] => (item={u'ip': u'192.168.1.11', u'hostname': u'XR1'})
changed: [localhost] => (item={u'ip': u'192.168.1.12', u'hostname': u'XR2'})
changed: [localhost] => (item={u'ip': u'192.168.1.21', u'hostname': u'NX1'})
changed: [localhost] => (item={u'ip': u'192.168.1.22', u'hostname': u'NX2'})

PLAY RECAP
********************************************************************************
localhost                  : ok=1    changed=1    unreachable=0    failed=0
```

```
$ ls | grep 'hosts'
hosts_NX1.txt
hosts_NX2.txt
hosts_XR1.txt
hosts_XR2.txt
```

```
$ cat hosts_*
192.168.1.21    NX1
192.168.1.22    NX2
192.168.1.11    XR1
192.168.1.12    XR2
```

As in the previous example, in Example 19-29, the Jinja2 template is stored in a separate file with a *.j2 extension. The variables are stored in separate files, one for each network device, and imported using the **include_vars** module. Then the Ansible **template** module takes a template from **src**, fills in all the variables, and creates a file in **dest**. Both the **src** and **dest** paths can be relative or absolute. The advantage of this type of module is that you see the exact text (for example, a set of configuration lines) that will be sent to a remote network device. Many Ansible configuration modules have an option to take input from external files, and the idea is that you create a template, augment it with variables, render a final configuration file, and then push this file to a network device. If you prefer to use the **lookup** plug-in, as shown in Example 19-28, you can use this template module to troubleshoot your template. (You might want to do this because not everything is straightforward with Jinja2, especially when you're new to it.)

There is one drawback to the method used in Example 19-29. You can see that for every hostname, a dedicated file is created. This occurs because the loop was created within Ansible. On the other hand, it's possible to create loops and conditions inside a template itself by using Jinja2 syntax. (This is a handy feature, and you will learn how to do it in the next section.)

So far you have learned the main ways to work with Jinja2 templates. There are no strict guidelines regarding which approach you should use in specific use cases. However, there is one general consideration you need to be aware of: You should check whether the module you are using supports input from an external file (in which case you can use the **template** module to create a temporary file) or not (in which case you need to use the **lookup** plug-in or an inline template).

Variables, Loops, and Conditions

You are already familiar with Jinja2 variables, which are represented using the syntax **{{** *variable* **}}**. Examples 19-28 and 19-29 show this syntax for dedicated files with templates. However, this is just a small part of template capabilities, as templates support both loops and conditions inside a template. Before we jump into examining loops and conditions, note that the approaches described here are relevant for both inline templates (refer to Example 19-27) and dedicated files (refer to Examples 19-28 and 19-29). To help you better understand the approaches, this section shows examples using only one approach at a time.

The logic of loops in Jinja2 templates is very close to the logic in the Ansible (as well as other programming languages). To show the syntax of loops in Jinja2, Example 19-30 shows the same example as Example 19-27, which involves the templating of entries in /etc/hosts, but now using a Jinja2 template.

Example 19-30 *A Jinja2 Template with Loop Syntax*

```
$ cat npf_example_inline_template_loop.yml
---
- hosts: linux
  gather_facts: no
  connection: local
  become: yes

  vars:
      new_hosts:
            - { hostname: XR1, ip: 192.168.1.11 }
            - { hostname: XR2, ip: 192.168.1.12 }
            - { hostname: NX1, ip: 192.168.1.21 }
            - { hostname: NX2, ip: 192.168.1.22 }

  tasks:
      - name: ADDING INFO TO /etc/hosts
        blockinfile:
            dest: /etc/hosts
            marker: "#{mark} ANSIBLE MANAGED BLOCK"
            block: |
                {% for some_var_name in new_hosts %}
                {{ some_var_name.ip }}   {{ some_var_name.hostname }}
                {% endfor %}
...


$ ansible-playbook npf_example_inline_template_loop.yml -i hosts --ask-become-pass
SUDO password:

PLAY [linux]
*******************************************************************************

TASK [ADDING INFO TO /etc/hosts]
************************************************************
changed: [localhost]

PLAY RECAP
********************************************************************************
localhost                  : ok=1    changed=1    unreachable=0    failed=0


$ cat /etc/host
```

```
cat: /etc/host: No such file or directory

[karneliuka@devlelautomatron book]$ cat /etc/hosts

127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4

::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

#BEGIN ANSIBLE MANAGED BLOCK

192.168.1.11   XR1

192.168.1.12   XR2

192.168.1.21   NX1

192.168.1.22   NX2

#END ANSIBLE MANAGED BLOCK
```

You can immediately see the difference between Example 19-30 and Example 19-27: In Example 19-30, all the new entries are inserted as a single block in the /etc/hosts file rather than as individual entries, as in Example 19-27. Moreover, now you don't see the keyword **loop** associated with the module **blockinfile** because the loop is built using Jinja2 template methods. The required entries are collected in the **vars** section, in the **new_hosts** list of variables. Then, in the inline Jinja2 template, there is a construction that starts with **{% for some_var_name in new_hosts %}** and ends with **{% endfor %}**. With **{% for %}** and **{% endfor %}**, you define the beginning and the ending of the cycle. The variable **{{ some_var_name }}** has the same function as **{{ item }}** in the Ansible loop: It assigns the value of a single entry from the list, where a single entry is an abstract definition of one part of the list, even if it has a complex hierarchical structure inside. Besides the definition of the variable, there is also the keyword **in**, which points to the list from which the entries are coming. In Example 19-30, the list is called **{{ new_hosts }}**, and this is the name of the variable defined in **vars**.

The syntax in the loop in Example 19-30 is the same as the syntax for the Ansible loop in Example 19-27, and you call any particular key from the list entry using **{{** *item.key* **}}** syntax. With Jinja2 loops, you can have a hierarchy of cycles inside the cycles. Example 19-31 shows two levels of hierarchy for configuration interfaces in Cisco NX-OS.

Example 19-31 *A Jinja2 Template with a Hierarchical Loop and Associated Variables*

```
$ cat npf_NX1_l2_variables.yml

---

list_of_interfaces:

    - id: 1/1

      vlans:

          - 123

          - 234

          - 345

          - 456

    - id: 2/1

      vlans:

          - 123

          - 234

    - id: 1/7

      vlans:

          - 345

          - 456

    - id: 1/18

      vlans:

          - 123

          - 456

...



$ cat npf_template_double_loop.j2

{% for iface in list_of_interfaces %}

interface {{ interface.name_prefix }}{{ iface.id }}

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add {% for v in iface.vlans %}{{ v }},{% endfor %}


{% endfor %}
```

First of all, in Example 19-31, the variables in the dedicated files for devices are structured such that the top-level variable **list_of_interfaces** is a list containing dictionaries consisting of two variables, **id** and **vlans**. As this second list is called from the first loop, it is called using the variables' names within the loop (that is, **{{ iface.vlans }}**). This is an important point, and you need to pay attention to the current level of

hierarchy and call variables accordingly.

Using external files with variables, where each filename includes the name of the respective network device, as well as an external template, makes it possible to keep a playbook very small and easy to understand (see Example 19-32).

Example 19-32 *A Sample Playbook Containing External Variables and Templates*

```
$ cat npf_example_double_loops_jinja2.yml

---

- hosts: nexus

  gather_facts: yes

  connection: local


  vars:

      interface:

          name_prefix: Eth


  tasks:

      - name: IMPORTING VARS

        include_vars:

            file: npf_{{ inventory_hostname }}_l2_variables.yml


      - name: TEMPLATING CONFIG

        template:

            src: npf_template_double_loop.j2

            dest: "{{ inventory_hostname }}_intent_config.conf"

...
```

Example 19-33 shows the execution of this playbook and its outcome.

Example 19-33 *Playbook Execution and Results*

```
$ ansible-playbook npf_example_double_loops_jinja2.yml


PLAY [nexus]

*******************************************************************************


TASK [Gathering Facts]

*********************************************************************

ok: [NX1]

ok: [NX2]


TASK [IMPORTING VARS]

**********************************************************************

ok: [NX1]

ok: [NX2]


TASK [TEMPLATING CONFIG]

**********************************************************************

changed: [NX2]

changed: [NX1]


PLAY RECAP

*******************************************************************************

NX1                        : ok=3    changed=1    unreachable=0    failed=0

NX2                        : ok=3    changed=1    unreachable=0    failed=0



$ cat NX*.conf

interface Eth1/1
```

```
    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,234,345,456,

interface Eth2/1

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,234,

interface Eth1/7

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 345,456,

interface Eth1/18

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,456,
```

From the output in Example 19-33, you can see that there is a comma at the end of each configuration line, including the VLAN list. These commas are required by CLI syntax; to see this in Example 19-31, look for the template file **npf_template_double_loop.j2**, and in the second loop you can see the string **{{ v }},** (with a comma), which is templated. However, in Example 19-33, the comma is also presented after the last VLAN, which will be parsed by the network device and result in error. Hence, the last comma should be removed. You could solve this issue by using conditionals in a Jinja2 template.

Conveniently, that the conditional rules, operators, and notation in Jinja2 are the same as in Ansible (see the section "Conditionals," earlier in this chapter). Example 19-34 shows how to fix the problem with the last comma in the Jinja2 template shown in Example 19-33.

Example 19-34 *A Jinja2 Template with a Conditional Statement*

```
$ cat npf_template_double_loop.j2

{% for iface in list_of_interfaces %}

interface {{ interface.name_prefix }}{{ iface.id }}

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add {% for v in iface.vlans %}{% if loop.index == 1

%}{{ v }}{% else %},{{ v }}{% endif %}{% endfor %}



{% endfor %}




$ cat NX*.conf

interface Eth1/1

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,234,345,456

interface Eth2/1

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,234

interface Eth1/7

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 345,456

interface Eth1/18

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,456
```

The file with the external variables and the Ansible playbook and its execution are exactly the same as in Example 19-31, so they are omitted in Example 19-34. A conditional statement is framed starting with **{% if ... %}** *action1* **{% else %}** *action2* **{% endif %}**, where *action1* occurs if the condition is fulfilled, and *action2* occurs it isn't. The condition is provided in the first **{% if ... %}** statement; in this example, **loop.index == 1**, where **{{ loop.index }}** is a built-in variable that exists in any loop context **{% for ... %}** and whose value is the number of iterations since the beginning of the cycle. The discussion of Example 19-21, earlier in this chapter, explains how to use a similar approach with Ansible loops. As you can see in Example 19-34, the string **{{ v }}** is templated for the first iteration of the loop, and then **,{{ v }}** (with a leading comma) is

templated. The major difference is that in Example 19-34, the comma is added before the VLAN number and only starting from the second VLAN, whereas in Example 19-31, the comma is added after each VLAN number starting from the first one. The approach in Example 19-34 allows you to create the proper string for Nexus configuration with any number of VLANs in the list (even one).

In general, all the operators for comparison are the same as the ones described earlier in this chapter, in the section "Conditionals." In a nutshell, Ansible uses Jinja2 operators, although the Jinja2 syntax does not include the **{% if ... %}** framing.

Example 19-35 shows another operator that is called in conjunction with Example 19-34 to anticipate the case when the VLANs aren't provided explicitly.

Example 19-35 *A Jinja2 Template with Two Conditional Statements*

```
$ cat npf_NX1_l2_variables.yml

---

list_of_interfaces:

    - id: 1/1

      vlans:

          - 123

          - 234

          - 345

          - 456

    - id: 2/1

    - id: 1/7

    - id: 1/18

      vlans:

          - 123

          - 456

...



$ cat npf_template_double_loop.j2

{% for iface in list_of_interfaces %}

interface {{ interface.name_prefix }}{{ iface.id }}

    switchport

    switchport mode trunk

{% if iface.vlans is defined %}

    switchport trunk allowed vlan add {% for v in iface.vlans %}{% if loop.index == 1

%}{{ v }}{% else %},{{ v }}{% endif %}{% endfor %}

{% endif %}


{% endfor %}



$ cat NX1_intent_config.conf

interface Eth1/1

    switchport

    switchport mode trunk

    switchport trunk allowed vlan add 123,234,345,456

interface Eth2/1

    switchport

    switchport mode trunk


interface Eth1/7

    switchport

    switchport mode trunk


interface Eth1/18

    switchport

    switchport mode trunk
```

```
    switchport trunk allowed vlan add 123,456
```

Just as in Example 19-34, the Ansible playbook isn't provided in Example 19-35 because it hasn't changed. As you can see at the beginning of Example 19-35, not every interface in the file with variables has a list of associated VLANs. It is possible that an interface may not have associated VLANs; this may be the case, for example, when the interface is used for trunking all the available VLANs. Such a case needs to be anticipated in the Jinja2 template to avoid misbehavior of the network element caused by configuration created with syntax mistakes. In Example 19-35, this is achieved with the condition check **{% if iface.vlans is defined %}** *action* **{% endif %}**. You can see that the operator is used with a condition defined, precisely in the same way it is used earlier in this chapter, in Example 19-19.

The last important point you need to understand about conditional statements in Jinja2 is the possibility of including multiple conditional checks by using the syntax **{% if ... %}** *action1* **{% elif ... %}** *action2* **{% elif ... %}** *actionN* **{% endif %}**. This approach allows you to create a powerful template that reacts differently to different conditions. Example 19-36 extends the previous Jinja2 template with this new concept.

Example 19-36 *A Jinja2 Template with Multiple Choice Conditions*

```
$ cat npf_NX1_l2_variables.yml
---
list_of_interfaces:
    - id: 1/1
      vlans:
          - 123
          - 234
          - 345
          - 456
    - id: 2/1
    - id: 1/7
      ip: 10.0.0.0/31
    - id: 1/18
      vlans:
          - 123
          - 456
...



$ cat npf_template_double_loop.j2
{% for iface in list_of_interfaces %}
interface {{ interface.name_prefix }}{{ iface.id }}
{% if iface.vlans is defined %}
    switchport
    switchport mode trunk
    switchport trunk allowed vlan add {% for v in iface.vlans %}{% if loop.index == 1
%}{{ v }}{% else %},{{ v }}{% endif %}{% endfor %}
{% elif iface.ip is defined %}
    no switchport
    ip address {{ iface.ip }}
{% else %}
    shutdown
{% endif %}
!
{% endfor %}



$ cat NX1_intent_config.conf
interface Eth1/1
    switchport
    switchport mode trunk
    switchport trunk allowed vlan add 123,234,345,456
!
interface Eth2/1
    shutdown
```

```
!
interface Eth1/7
    no switchport
    ip address 10.0.0.0/31
!
interface Eth1/18
    switchport
    switchport mode trunk
    switchport trunk allowed vlan add 123,456
!
```

Note

Again, the Ansible playbook in Example 19-36 is the same as defined in Example 19-31, so it is not repeated here.

Example 19-36 shows one of the advantages of templates, as you can see here the strict separation between what is done (in the Jinja2 template) and how it is done (in the Ansible playbook). By just tuning the template—and not even touching the playbook—you get different outputs.

If you take a close look at the template in Example 19-36, you can see that there are several possible conditions to match. It's important to know that only one condition at a time can be matched, which means that if the first condition is fulfilled, the rest of the conditions are not even checked. That is, the evaluation is performed until the first match is found. The template in Example 19-36 also illustrates the concept of multiple conditions in a single statement united by AND logic, which works as explained earlier in this chapter in Example 19-21."

Using Python Functions in Jinja2

As mentioned at the beginning of this chapter, Jinja2 is a templating language for Python. This means that Jinja2 and Python are tightly entwined with each other, and you can use various filters in Jinja2 expressions, as shown earlier in this chapter.

In a nutshell, *filters* are Python functions that perform various tasks. In the context of network programmability, templates are aimed at dynamically creating full-blown configuration files with sets of variables as input. Therefore, the vast majority of the filters you use with networks are focused on text processing and modification. You can find a full list of filters supported in Jinja2 at the official Jinja2 website (https://jinja.palletsprojects.com/en/2.11.x/). The following sections provide some examples of the most commonly used Python functions and filters.

The join() Function

The **join()** function makes it possible to merge different pieces of information; this is useful when you're processing arrays. The **join()** function may also help you more easily make templates by preventing the creation of additional loops. Example 19-37 shows how to improve the template from Example 19-36 in terms of interface templating for Cisco NX-OS.

Example 19-37 *A Jinja2 Template with the **join()** Function*

```
$ cat npf_template_double_loop_join.j2
{% for iface in list_of_interfaces %}
interface {{ interface.name_prefix }}{{ iface.id }}
{% if iface.vlans is defined %}
    switchport
    switchport mode trunk
    switchport trunk allowed vlan add {{ iface.vlans | join (',')}}
{% endif %}
!
{% endfor %}
```

```
$ cat npf_example_double_loops_jinja2_join.yml
---
- hosts: nexus
  gather_facts: yes
  connection: local

  vars:
      interface:
          name_prefix: Eth

  tasks:
```

```
    - name: IMPORTING VARS
      include_vars:
          file: npf_{{ inventory_hostname }}_l2_variables.yml


    - name: TEMPLATING CONFIG
      template:
          src: npf_template_double_loop_join.j2
          dest: "{{ inventory_hostname }}_intent_config.conf"
...
```

```
$ ansible-playbook npf_example_double_loops_jinja2_join.yml
PLAY [nexus]
*******************************************************************************
! The output is truncated for brevity
PLAY RECAP
********************************************************************************
NX1                        : ok=3    changed=1    unreachable=0    failed=0
NX2                        : ok=3    changed=1    unreachable=0    failed=0
```

```
$ cat NX1_intent_config.conf
interface Eth1/1
    switchport
    switchport mode trunk
    switchport trunk allowed vlan add 123,234,345,456
!
interface Eth2/1
!
interface Eth1/7
!
interface Eth1/18
    switchport
    switchport mode trunk
    switchport trunk allowed vlan add 123,456
!
```

It is important to note here that the result of the template execution in Example 19-37 is the same as in Example 19-36; this indicates that you have a proper configuration file. On the other hand, the new template is much more elegant and shorter, thanks to the **join** filter. Its syntax is **{{** *input_array* **| join ('***separator***') }}**, and it works by merging together all the items of the **{{** *input_array* **}}** variable into a single string by using the separator value provided as an argument to the **join()** function.

The split() Function

The **split()** function is basically the opposite of the **join()** function: It allows you to split an input string into several pieces and manage them separately. Say that you need to install a new top-of-rack switch in a data center, wire all the servers to the proper ports, and prepare the configuration of the switch. There are a lot of fancy tools for managing network infrastructure, but table processors (such as Microsoft Excel) are still some of the most popular tools for documenting information for such an activity. Such a tool presents this information as shown in Table 19-6.

Table 19-6 *Wiring Request for a New Data Center Switch*

HOST	HOST_PORT	HOST@IP	SWITCH	PORT	VLAN
Server1	Eth0	192.168.1.10/24	NX1	1	120
Server2	Eth0	192.168.1.11/24	NX1	2	120
Server3	Ens0	192.168.2.5/24	NX1	5	130

The file formats .xls and .xlsx are not very friendly for the automation, but the .csv format works well. You are lucky if you get a task as a data_center_wiring.csv file; otherwise, you can translate a file from.xls format to .csv format. Example 19-38 shows how a .csv file is structured.

Example 19-38 *The Structure of a .csv File*

```
HOST,HOST_PORT,HOST@IP,SWITCH,PORT,VLAN
Server1,Eth0,192.168.1.10/24,NX1,1,120
```

```
Server2,Eth0,192.168.1.11/24,NX1,2,120

Server3,Ens0,192.168.2.5/24,NX1,5,130
```

You can use the **split()** function in Jinja2 templates to translate an .csv file to to a set of variables, which is ready for the configuration of a data center switch. Example 19-39 show how this works.

Example 19-39 *Using the **split()** Function in Ansible and Jinja2 Templates*

```
$ cat  npf_example_split.yml

---

- hosts: nexus

  gather_facts: yes

  connection: local


  tasks:

      - name: IMPORTING VARS

        set_fact:

            dummy_var: "{{ lookup('file','data_center_wiring.csv')}}"


      - name: TEMPLATING CONFIG

        debug:

            msg : |

                {% if item.split(',')[3] == inventory_hostname %}

                Interface Eth1/{{ item.split(',')[4] }}

                    switchport

                    switchport access vlan {{ item.split(',')[5] }}

                    description {{ item.split(',')[0] }}:{{ item.split(',')[1] }}---{{

item.split(',')[3] }}:Eth1/{{ item.split(',')[4] }}

                    no shutdown

                !

                {% endif %}

        loop: "{{ dummy_var.split('\n')}}"

...



$ ansible-playbook npf_example_split.yml -i hosts


PLAY [nexus]

*******************************************************************************


TASK [Gathering Facts]

*********************************************************************

ok: [NX2]

ok: [NX1]


TASK [IMPORTING VARS]

**********************************************************************

ok: [NX1]

ok: [NX2]


TASK [TEMPLATING CONFIG]

**********************************************************************

ok: [NX1] => (item=HOST,HOST_PORT,HOST@IP,SWITCH,PORT,VLAN) => {

    "msg": "\n"

}

ok: [NX1] => (item=Server1,Eth0,192.168.1.10/24,NX1,1,120) => {
```

```
    "msg": "Interface Eth1/1\n    switchport\n    switchport access vlan 120\n
description Server1:Eth0---NX1:Eth1/1\n    no shutdown\n!\n"
}
ok: [NX1] => (item=Server2,Eth0,192.168.1.11/24,NX1,2,120) => {
    "msg": "Interface Eth1/2\n    switchport\n    switchport access vlan 120\n
description Server2:Eth0---NX1:Eth1/2\n    no shutdown\n!\n"
}
ok: [NX1] => (item=Server3,Ens0,192.168.2.5/24,NX1,5,130) => {
    "msg": "Interface Eth1/5\n    switchport\n    swicthport access vlan 130\n
description Server3:Ens0---NX1:Eth1/5\n    no shutdown\n!\n"
}
ok: [NX2] => (item=HOST,HOST_PORT,HOST@IP,SWITCH,PORT,VLAN) => {
    "msg": "\n"
}
ok: [NX2] => (item=Server1,Eth0,192.168.1.10/24,NX1,1,120) => {
    "msg": "\n"
}
ok: [NX2] => (item=Server2,Eth0,192.168.1.11/24,NX1,2,120) => {
    "msg": "\n"
}
ok: [NX2] => (item=Server3,Ens0,192.168.2.5/24,NX1,5,130) => {
    "msg": "\n"
}


PLAY RECAP
*********************************************************************************
NX1                        : ok=3    changed=0    unreachable=0    failed=0
NX2                        : ok=3    changed=0    unreachable=0    failed=0
```

The syntax of **split()** function is **{{** *variable*.**split('***separator***')[***index***] }}**, where '*separator*' is a value used to split the original text into an array, and *index* calls for particular element out of the array.

Note

The **split()** function is applicable not only in the context of Jinja2 templates but also with Ansible in general.

The playbook in Example 19-39 assigns all the contents of the file data_center_wiring.csv by using the **lookup** plug-in. Then this value is split into separate strings by the **split()** function with '*\n*' (newline) separator. In this case, the **split()** function is also used to create a loop that calls a template for every string, which is split further by the same **split()** function, this time with the '**,**' separator, which is used in .csv file to merge the columns into a single string. Then, in the in-line template, a conditional check allows a template to be used only in event that **{{** *inventory_hostname* **}}** of the device corresponds the value *SWITCH* in the input table. If the check is successful, the template creates a configuration for a particular port by using the values from the array created using the **split()** function.

The map() Function

The **map()** function is handy for creating reports out of files as it searches for particular values in the list structure without using loops directly. Let's say you already have some data collected from a network or from an application, and you want to verify the elements for which the data is provided. Example 19-40 shows how you can use the **map()** function to get some insights.

Example 19-40 *Using the **map()** Function in a Jinja2 Template*

```
$ cat npf_example_map.yml
---
- hosts: linux
  gather_facts: no
  connection: local
  become: yes

  vars:
      new_hosts:
            - { hostname: XR1, ip: 192.168.1.11 }
            - { hostname: XR2, ip: 192.168.1.12 }
```

```
            - { hostname: NX1, ip: 192.168.1.21 }
            - { hostname: NX2, ip: 192.168.1.22 }

  tasks:
      - name: ADDING INFO TO /etc/hosts
        debug:
            msg: "{{ new_hosts | map(attribute='hostname') | join (', ') }}"
...
```

```
$ ansible-playbook npf_example_map.yml -i hosts

PLAY [linux]
*******************************************************************************

TASK [ADDING INFO TO /etc/hosts]
***********************************************************
ok: [localhost] => {
    "msg": "XR1, XR2, NX1, NX2"
}

PLAY RECAP
*******************************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0
```

The syntax for the **map()** function is **map(***attribute***='***key***')**, where '*key*' is the key value as provided in the list. The **map()** function is often used in conjunction with the **join()** function to form the proper output format. As you can see in Example 19-40, the **map()** function searches in the initial dictionary for all '*hostname*' keys and returns their values merged in a single string by using **join()** function with the comma separator.

As mentioned earlier, the full list of the available filters and functions is available at the official Jinja2 website (https://jinja.palletsprojects.com/en/2.11.x/). At this point, you have the knowledge you need to start automating your network with Ansible. of the rest of this chapter provides examples to help you cement your knowledge.

Using Ansible for Cisco IOS XE

To automate anything with Ansible, you need at least two components: a managing host that runs Ansible and a managed host that is to be controlled. Figure 19-5 shows a simple topology with a managing host running Linux with Ansible and three managed hosts (each running a different Cisco network OS). This topology is used throughout the rest of this chapter. Because this particular part of the chapter is dedicated to Cisco IOS XE, the main focus here is on the CSR2 router.

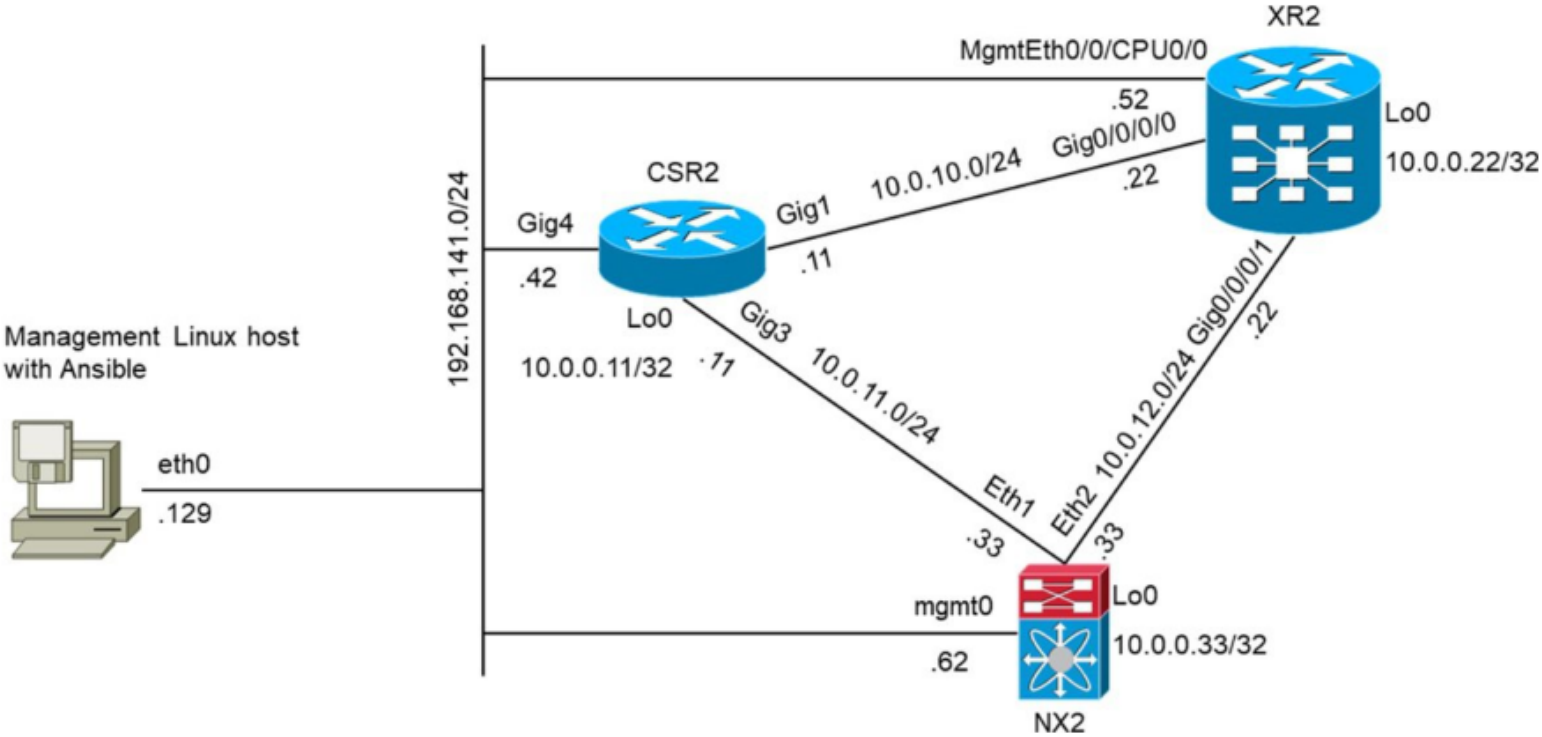

Figure 19-5 *Topology Used for Cisco Automation with Ansible*

Recall that Ansible is *agentless*, which means no agent needs to be installed on the managed host. By default, Ansible uses SSH to establish a communication channel with the host; therefore, the destination host must have an SSH server configured as well as a corresponding

user/password pair. Example 19-41 shows how to prepare a Cisco IOS/Cisco IOS XE router, assuming that it is a freshly booted router without any configuration.

Example 19-41 *Preparing a Cisco IOS XE Device to Be Managed by Ansible*

```
Router# configure terminal

Router(config)# hostname CSR2

CSR2(config)# username cisco secret cisco

CSR2(config)# username cisco privilege 15

CSR2(config)# ip domain-name npf

CSR2(config)# crypto key generate rsa general-keys modulus 2048

The name for the keys will be: CSR2.npf


% The key modulus size is 2048 bits

% Generating 2048 bit RSA keys, keys will be non-exportable...

[OK] (elapsed time was 1 seconds)


CSR2(config)# ip ssh version 2

CSR2(config)# interface gig4

CSR2(config-if)# ip address 192.168.141.42 255.255.255.0

CSR2(config-if)# no shutdown

*Jan  5 18:12:41.061: %LINK-3-UPDOWN: Interface GigabitEthernet4, changed state to up

*Jan  5 18:12:42.061: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet4,

changed state to up

CSR2(config-if)# line vty 0 4

CSR2(config-line)# login local

CSR2(config-line)# end

CSR2(config)# copy run start
```

After this basic configuration is performed, you need to ensure that CSR2 is reachable from the managed host, and you should be able to log in to it, as demonstrated in Example 19-42.

Example 19-42 *Verifying That Cisco IOS XE Is Ready to Be Managed Using Ansible*

```
$ ping CSR2 -c 1

PING CSR2 (192.168.141.42) 56(84) bytes of data.

64 bytes from CSR2 (192.168.141.42): icmp_seq=1 ttl=255 time=0.460 ms


--- CSR2 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 0.460/0.460/0.460/0.000 ms




$ ssh cisco@CSR2

Password:


CSR2#
```

Now that connectivity has been established, and CSR2 is ready to be automated, there is one more activity to do on the managing host: You need to update the Ansible hosts file to reflect the actual network OS versions, as well as the credentials for accessing the network. Example 19-43 shows how you do this.

Example 19-43 *Updating a File with Ansible Hosts with Additional Variables*

```
$ cat hosts

[linux]

localhost


[ios]

CSR1

CSR2


[iosxr]
```

```
XR1
XR2

[nexus]
NX1
NX2

[nexus:vars]
ansible_network_os=nxos
ansible_user=cisco
ansible_ssh_pass=cisco

[iosxr:vars]
ansible_network_os=iosxr
ansible_user=cisco
ansible_ssh_pass=cisco

[ios:vars]
ansible_network_os=ios
ansible_user=cisco
ansible_ssh_pass=cisco
```

Operational Data Verification Using the ios_command Module

Earlier in this chapter, you got a glimpse of the module **iosxr_command**, which deals with various commands (besides configuration) in Cisco IOS XR. For Cisco IOS/IOS XE, a comparable module exists: **ios_command**. Example 19-44 shows how to use this module.

Example 19-44 *Using the Ansible Module **ios_command** to Collect **show** Output*

```
$ cat npf_example_ios_command.yml
---
- hosts: ios
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        ios_command:
            commands:
                - show inventory
                - show ip interface brief
        register: ios_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ ios_output.stdout_lines }}"
...



$ ansible-playbook npf_example_ios_command.yml --limit=CSR2

PLAY [ios]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [CSR2]
```

```
TASK [COLLECTING OPEPRATION INFORMATION FROM 'ios']
*******************************************
ok: [CSR2]

TASK [VERIFICATION OF THE OUTPUT]
***********************************************************
ok: [CSR2] => {
    "msg": [
        [
            "NAME: \"Chassis\", DESCR: \"Cisco CSR1000V Chassis\"",
            "PID: CSR1000V          , VID: V00, SN: 9L4Z3MVC8LQ",
            "",
            "NAME: \"module R0\", DESCR: \"Cisco CSR1000V Route Processor\"",
            "PID: CSR1000V          , VID: V00, SN: JAB1303001C",
            "",
            "NAME: \"module F0\", DESCR: \"Cisco CSR1000V Embedded Services
Processor\"",
            "PID: CSR1000V          , VID:    , SN:"
        ],
        [
            "Interface              IP-Address      OK? Method Status
Protocol",
            "GigabitEthernet1       unassigned      YES NVRAM  administratively down
down    ",
            "GigabitEthernet3       unassigned      YES unset  administratively down
down    ",
            "GigabitEthernet4       192.168.141.42  YES manual up                    up"
        ]
    ]
}

PLAY RECAP
*********************************************************************************
CSR2                       : ok=3    changed=0    unreachable=0    failed=0
```

Because only CSR2 is part of the reference lab topology, during the execution of the Ansible playbook, the tag --limit=CSR2 is used to avoid running the playbook against all the hosts from the ios group. Then, as credentials are provided in the hosts file and **gather_facts** is enabled, there is no need to enter any usernames or passwords during the execution of the playbook, which makes the automation smooth. In the module **ios_command**, the critical component is the section **commands**, which lists all the commands to be performed on the remote host. In Example 19-44, two commands are executed: **show inventory** and **show ip interface brief**. As these commands generate output, the output is saved in the registered value **{{** *ios_output* **}}** (as explained earlier in this chapter). Then the output is displayed using the **debug** module to verify that it's appropriately collected. As you now know, there are other options here, such as parsing values, saving them to a database (with Ansible's modules to work with databases), or reporting further. Obviously, you can use the mechanism shown in Example 19-44 to collect the running configuration from all your network elements for backup.

In addition to executing **show** commands on Cisco IOS XE devices, it's possible to perform some other commands—and they might be even interactive. Example 19-45 shows, for instance, how to clear the counters on a Cisco IOS XE device.

Example 19-45 *Using the Ansible Module* ***ios_command*** *to Clear Counters*

```
$ cat npf_example_ios_command_clear.yml
---
- hosts: ios
  connection: network_cli
  gather_facts: yes

  tasks:
    - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
      ios_command:
```

```
            commands:
                - command: show interfaces GigabitEthernet4 | inc packets
                - command: clear counters GigabitEthernet4
                  prompt: '[confirm]'
                  answer: "\r"
                - command: show interfaces GigabitEthernet4 | inc packets
        register: ios_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ ios_output.stdout_lines }}"
...
```

```
$ ansible-playbook npf_example_ios_command_clear.yml --limit=CSR2

PLAY [ios]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [CSR2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'ios']
*****************************************
ok: [CSR2]

TASK [VERIFICATION OF THE OUTPUT]
***********************************************************
ok: [CSR2] => {
    "msg": [
        [
            "5 minute input rate 0 bits/sec, 0 packets/sec",
            "  5 minute output rate 0 bits/sec, 0 packets/sec",
            "     681 packets input, 80415 bytes, 0 no buffer",
            "     576 packets output, 75856 bytes, 0 underruns"
        ],
        [
            "Clear \"show interface\" counters on this interface [confirm]"
        ],
        [
            "5 minute input rate 0 bits/sec, 0 packets/sec",
            "  5 minute output rate 0 bits/sec, 0 packets/sec",
            "     0 packets input, 0 bytes, 0 no buffer",
            "     0 packets output, 0 bytes, 0 underruns"
        ]
    ]
}

PLAY RECAP
*******************************************************************************
CSR2                       : ok=3    changed=0    unreachable=0    failed=0
```

All the commands in this example start with the key **command**, and the keys **prompt** and **answer** are introduced. The **command** key contains the pattern that is expected and the string '\r' (a carriage return) that is to be sent in response. As you can see in the playbook in Example 19-

45, the counters are collected and then cleared and collected again. The second collection confirms that the counters are cleared successfully.

General Configuration Using the ios_config Module

You might say that so far you have seen no examples of real automation and network programming; you have only learned how to collect information from a remote Cisco IOS XE network device and how to use several modules for text processing/templating. Well, that's partially true, and now it's time to look at how to configure network functions in an automated way. There are plenty of modules for Cisco IOS/IOS XE devices, and they allow you to configure different aspects of a router or switch. One particular module, called **ios_config**, relies on traditional CLI commands. Therefore, to use it, you need to be proficient with the CLI. However, it allows you to automate everything, as long as you can create proper templates.

To get started with automation, let's start with some basics. Example 19-46 shows the configuration of the interface at CSR2 with all the parameters hardcoded.

Example 19-46 *Using the Ansible Module* ***ios_config*** *Without Variables*

```
$ cat npf_example_ios_config_simple.yml
---
- hosts: ios
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: CONFIGURING THE INTERFACE
        ios_config:
            lines:
                - ip address 10.0.10.11 255.255.255.0
                - no shutdown
            parents:
                - interface GigabitEthernet1

      - name: WAITING FOR CARRIER-DELAY
        pause:
            seconds: 5

      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        ios_command:
            commands:
                - show ip interface brief
        register: ios_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ ios_output.stdout_lines }}"
...


$ ansible-playbook npf_example_ios_config_simple.yml -i hosts --limit=CSR2

PLAY [ios]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [CSR2]

TASK [CONFIGURING THE INTERFACE]
***********************************************************
changed: [CSR2]
```

```
TASK [WAITING FOR CARRIER-DELAY]
************************************************************
Pausing for 5 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [CSR2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'ios']
*****************************************
ok: [CSR2]

TASK [VERIFICATION OF THE OUTPUT]
************************************************************
ok: [CSR2] => {
    "msg": [
        [
            "Interface              IP-Address      OK? Method Status
Protocol",
            "GigabitEthernet1       10.0.10.11      YES manual up                    up
",
            "GigabitEthernet3       unassigned      YES unset  administratively down
down    ",
            "GigabitEthernet4       192.168.141.42  YES manual up
up"
        ]
    ]
}

PLAY RECAP
*********************************************************************************
CSR2                       : ok=5    changed=1    unreachable=0    failed=0
```

In the **ios_config** module, the key components are the section **lines**, which contains the configuration commands, and **parents**, which contains the target context where the configuration must be applied. **parentslatter** is essential because without it, the commands would be applied to the global configuration context. In Example 19-46, the **ios_config**, **ios_command**, and **debug** modules are used to verify that changes have been applied successfully. In addition, the module **pause** delays the execution of the Ansible playbook by a defined number of seconds. This is very useful if the results of the execution of the previous module aren't immediate. In this case, it takes roughly 2 seconds (the default carrier-delay timer of the Cisco IOS XE device) before the interface's operational state changes to up; this is why the module **pause** is used. After this timer expires, the playbook is executed further, and the verification by the **ios_command** module shows that the newly created interface is up. When you look at the log of the playbook, you can see that in the second task, called CONFIGURING THE INTERFACE, the status is changed; this means there were some changes in the configuration of the remote device.

Providing all the parameters directly in a playbook as hardcoded parameters is not the best way to automate network operations. Example 19-47 shows the use of loops and variables to achieve a higher degree of automation.

Example 19-47 *Using the Ansible Module **ios_config** with Loops and Variables*

```
$ cat npf_example_ios_config_extended.yml
---
- hosts: ios
  connection: network_cli
  gather_facts: yes

  vars:
      interfaces:
          - id: interface GigabitEthernet1
            ipv4:
                address: 10.0.10.11
                mask: 255.255.255.0
            state: no shutdown
          - id: interface GigabitEthernet3
            ipv4:
```

```
                address: 10.0.11.11
                mask: 255.255.255.0
            state: no shutdown
          - id: interface Loopback0
            ipv4:
                address: 10.0.0.11
                mask: 255.255.255.255
            state: no shutdown

  tasks:
      - name: CONFIGURING THE INTERFACE
        ios_config:
            lines:
                - ip address {{ item.ipv4.address }} {{ item.ipv4.mask }}
                - "{{ item.state }}"
            parents: "{{ item.id }}"
            save_when: modified
        loop: "{{ interfaces }}"

      - name: WAITING FOR CARRIER-DELAY
        pause:
            seconds: 5

      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        ios_command:
            commands:
                - show ip interface brief
        register: ios_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ ios_output.stdout_lines }}"
...
```

```
$ ansible-playbook npf_example_ios_config_extended.yml --limit=CSR2

PLAY [ios]
*******************************************************************************

TASK [Gathering Facts]
***********************************************************************
ok: [CSR2]

TASK [CONFIGURING THE INTERFACE]
*************************************************************
changed: [CSR2] => (item={u'state': u'no shutdown', u'id': u'interface
GigabitEthernet1', u'ipv4': {u'mask': u'255.255.255.0', u'address': u'10.0.10.11'}})
changed: [CSR2] => (item={u'state': u'no shutdown', u'id': u'interface
GigabitEthernet3', u'ipv4': {u'mask': u'255.255.255.0', u'address': u'10.0.11.11'}})
changed: [CSR2] => (item={u'state': u'no shutdown', u'id': u'interface Loopback0',
u'ipv4': {u'mask': u'255.255.255.255', u'address': u'10.0.0.11'}})
```

```
TASK [WAITING FOR CARRIER-DELAY]
************************************************************
Pausing for 5 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [CSR2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'ios']
*****************************************
ok: [CSR2]

TASK [VERIFICATION OF THE OUTPUT]
************************************************************
ok: [CSR2] => {
    "msg": [
        [
            "Interface              IP-Address      OK? Method Status
Protocol",
            "GigabitEthernet1       10.0.10.11      YES manual up                    up
",
            "GigabitEthernet3       10.0.11.11      YES manual up                    up
",
            "GigabitEthernet4       192.168.141.42  YES manual up                    up
",
            "Loopback0              10.0.0.11       YES manual up
up"
        ]
    ]
}

PLAY RECAP
*********************************************************************************
CSR2                       : ok=5    changed=1    unreachable=0    failed=0
```

Now the situation looks much better, as **vars** at the beginning of the playbook contains the dictionary with the variables that are used first to create the loop and then to create the interface configuration for each entry. In the module **ios_config**, there is a new key, **save_when**, with the value **modified**. As the name suggests, this key saves the configuration on the removed node to the startup configuration (in case the running configuration) is successfully modified.

Although the situation with the automation now is much better than it was before, there are still a lot of ways we could improve it. For instance, we haven't yet used templates. Example 19-48 shows the possible structure of the files with variables and Jinja2 templates to configure the OSPF routing protocol.

Example 19-48 *Using the Ansible Module* ***ios_config*** *in Conjunction with Templates*

```
$ cat CSR2_vars.yml
---
routing:
    - protocol: ospf
      id: 1
      router_id: 10.0.0.11
      interfaces:
          - id: GigabitEthernet1
            type: point-to-point
            passive: false
            area: 0.0.0.0
          - id: GigabitEthernet3
            type: point-to-point
            passive: false
            area: 0.0.0.0
```

```
          - id: Loopback0
            passive: true
            area: 0.0.0.0
...
```

```
$ cat npf_template_ios_config_pro.j2
{% if routing is defined %}
{% for rp in routing %}
{% if rp.protocol == 'ospf' %}
router {{ rp.protocol }} {{ rp.id }}
    router-id {{ rp.router_id }}
{% for r_if in rp.interfaces %}
{% if r_if.passive %}
    passive-interface {{ r_if.id }}
{% endif %}
{% endfor %}
!
{% for r_if in rp.interfaces %}
interface {{ r_if.id }}
    ip {{ rp.protocol }} {{ rp.id }} area {{ r_if.area }}
{% if r_if.type is defined %}
    ip {{ rp.protocol }} network {{ r_if.type }}
{% endif %}
!
{% endfor %}
{% endif %}
{% endfor %}
{% endif %}
```

In the same way shown in Example 19-32, the Ansible playbook for building the configuration of the Cisco IOS XR–based network device is fairly generic and, depending on the templates and variables, it could be used to create any configuration, as shown Example 19-49.

Example 19-49 *Using the Ansible Module* ***ios_config*** *in Conjunction with Templates, Continued*

```
$ cat npf_example_ios_config_pro.yml
---
- hosts: ios
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: IMPORTING VARIABLES
        include_vars:
            file: "{{ inventory_hostname }}_vars.yml"

      - name: TEMPLATING CONFIG
        template:
            src: npf_template_ios_config_pro.j2
            dest: "{{ inventory_hostname }}_temp.conf"

      - name: CONFIGURING THE DEVICE
        ios_config:
            src: "{{ inventory_hostname }}_temp.conf"
            save_when: modified
...
```

```
$ ansible-playbook npf_example_ios_config_pro.yml --limit=CSR2


PLAY [ios]
*******************************************************************************


! The output is truncated for brevity


PLAY RECAP
*******************************************************************************
CSR2                       : ok=4    changed=2    unreachable=0    failed=0




$ cat CSR2_temp.conf
router ospf 1
    router-id 10.0.0.11
    passive-interface Loopback0
!
interface GigabitEthernet1
    ip ospf 1 area 0.0.0.0
    ip ospf network point-to-point
!
interface GigabitEthernet3
    ip ospf 1 area 0.0.0.0
    ip ospf network point-to-point
!
interface Loopback0
    ip ospf 1 area 0.0.0.0
!
```

Examples 19-48 and 19-49 show examples of many concepts you've already used to perform automation in network operations. The variables are contained in the dedicated file CSR2_vars.yml, and they are imported at the beginning of the execution of the Ansible playbook. Then, using template npf_template_ios_config_pro.j2, the final configuration file CSR2_temp.conf is created. (The content of CSR2_temp.conf are shown at the end of Example 19-49.) As you can see, the template extensively uses loops and conditionals to build the configuration correctly. In the end, the module **ios_config** uses the key **src** with the value of the path to generate the configuration file CSR2_temp.conf instead of using **lines** and **parents**. These two methods are mutually exclusive, so you need to decide which one to use in a particular case. Example 19-50 shows the verification done directly on the router CSR2. (For brevity, the module **ios_command** is not included in this example.)

Example 19-50 *Verifying the OSPF Configuration on a Cisco IOS XE Router*

```
CSR2#show ip ospf interface brief
Interface     PID   Area            IP Address/Mask    Cost  State Nbrs F/C
Lo0           1     0.0.0.0         10.0.0.11/32       1     LOOP  0/0
Gi3           1     0.0.0.0         10.0.11.11/24      1     P2P   0/0
Gi1           1     0.0.0.0         10.0.10.11/24      1     P2P   0/0
```

In terms of a network-wide solution, Example 19-49 might be the most suitable option as the only thing you would need to do many times is to create the dedicated file with variables for each node (and even this could be automated further); the playbook and template would be created only once.

Configuration Using Various ios_* Modules

Some of the most critical goals of network programmability and automation are the simplification and acceleration of network operation processes. The engineers from Red Hat who develop Ansible are concerned with making these goals easy to reach. They have created a bunch of modules for declarative management, which works as follows: Users provide the desired state of the network, such as the state of an interface or IP addresses that should be present on a system, and the corresponding Ansible module analyzes the current operational state of the interfaces against the desired state. If any actions are needed to get the device to the desired state, they are done automatically, without additional effort from users. In a nutshell, this means that all the complexity of a particular CLI syntax is hidden from the users.

The previous section shows a declarative approach via variables defined using free syntax and templates that create proper CLI configuration based on those variables. This is the same declarative management just described, shown with implementation details. However, for specific tasks, you can use declarative modules from Ansible directly and avoid the creation of templates. You can find a full list of these modules on the official Ansible website (https://docs.ansible.com/ansible/latest/collections/index.html). This section provides examples of three such modules: **ios_interface**, **ios_l3_interface** (to compare with the earlier examples), and **ios_lldp**.

All these modules perform declarative management; that is, they compare the actual operational state of the network elements with the desired state and change the configuration only if necessary (for example, if the desired state is different from the operational state). Example 19-51 shows the use of the modules **ios_interface** and **ios_l3_interface** together.

Example 19-51 *Using the Ansible Modules* ***ios_interface*** *and* ***ios_l3_interface***

```
$ cat npf_example_ios_interface_l3.yml
---
- hosts: ios
  connection: network_cli
  gather_facts: yes

  vars:
      interfaces:
          - id: GigabitEthernet1
            ipv4:
                address: 10.0.10.11
                mask: 24
            description: CSR2<--->XR2
          - id: GigabitEthernet3
            ipv4:
                address: 10.0.11.11
                mask: 24
            description: CSR2<--->NX2
          - id: Loopback0
            ipv4:
                address: 10.0.0.11
                mask: 32
            description: RID

  tasks:
      - name: CONFIGURING THE INTERFACE
        ios_interface:
            name: "{{ item.id }}"
            enabled: true
            description: "{{ item.description }}"
        loop: "{{ interfaces }}"

      - name: CONFIGURE THE INTERFACE (L3)
        ios_l3_interface:
            name: "{{ item.id }}"
            ipv4: "{{ item.ipv4.address }}/{{ item.ipv4.mask }}"
        loop: "{{ interfaces }}"
...


$ ansible-playbook npf_example_ios_interface_l3.yml -i hosts --limit=CSR2

PLAY [ios]
*******************************************************************************

TASK [Gathering Facts]
***********************************************************************
ok: [CSR2]

TASK [CONFIGURING THE INTERFACE]
*************************************************************
changed: [CSR2] => (item={u'id': u'GigabitEthernet1', u'ipv4': {u'mask': 24,
```

```
u'address': u'10.0.10.11'}, u'description': u'CSR2<--->XR2'})

changed: [CSR2] => (item={u'id': u'GigabitEthernet3', u'ipv4': {u'mask': 24,

u'address': u'10.0.11.11'}, u'description': u'CSR2<--->NX2'})

changed: [CSR2] => (item={u'id': u'Loopback0', u'ipv4': {u'mask': 32, u'address':

u'10.0.0.11'}, u'description': u'RID'})


TASK [CONFIGURE THE INTERFACE (L3)]

*********************************************************

ok: [CSR2] => (item={u'id': u'GigabitEthernet1', u'ipv4': {u'mask': 24, u'address':

u'10.0.10.11'}, u'description': u'CSR2<--->XR2'})

ok: [CSR2] => (item={u'id': u'GigabitEthernet3', u'ipv4': {u'mask': 24, u'address':

u'10.0.11.11'}, u'description': u'CSR2<--->NX2'})

ok: [CSR2] => (item={u'id': u'Loopback0', u'ipv4': {u'mask': 32, u'address':

u'10.0.0.11'}, u'description': u'RID'})


PLAY RECAP

*********************************************************************************

CSR2                       : ok=3    changed=1    unreachable=0    failed=0
```

Example 19-51 is similar to Example 19-47, where interfaces are also configured. However, the variables have changed a bit: The state has been removed, the description has been added, and the mask has changed format from dotted decimal to prefix because the module **ios_l3_interaces** requires prefix format for the subnet mask.

The module **ios_interface** is used to configure physical parameters of an interface, its operational state, description, and some others parameters. Because no descriptions have been configured so far, you can see that the status of the Ansible task is **changed**, which means the configuration was updated. Then the module **ios_l3_interface** checks the configuration of IP addresses (it can work both IPv4 and IPv6 addresses) and updates them if they don't match. The status **ok** for this task means no changes were made; the IP addresses were configured properly earlier. If the IP addresses were missing or configured incorrectly, they would be updated to the desired values, and the status of the task would be **changed**.

The last module we need to cover for Cisco IOS XE in general and declarative management is **ios_lldp**, whose role is to configure LLDP. Example 19-52 provides details on its use.

Example 19-52 *Using the Ansible Module* ***ios_lldp***

```
$ cat npf_example_ios_lldp.yml

---

- hosts: ios

  connection: network_cli

  gather_facts: yes


  tasks:

      - name: ENABLING LLDP

        ios_lldp:

            state: present

...



$ ansible-playbook npf_example_ios_lldp.yml -i hosts --limit=CSR2


PLAY [ios]

*********************************************************************************


TASK [Gathering Facts]

*********************************************************************

ok: [CSR2]


TASK [ENABLING LLDP]

**********************************************************************

changed: [CSR2]
```

```
PLAY RECAP
*********************************************************************************
CSR2                       : ok=2    changed=1    unreachable=0    failed=0
```

The structure of the **ios_lldp** module is relatively easy, as it has only one key, **state**, with possible values **present** or **absent** that enable or disable LLDP globally on a Cisco IOS/IOS XE network device. Because LLDP isn't enabled by default, you can see in this example that the task ENABLING LLDP has changed state. In addition, there is not much happening in the background: The module either sends **lldp run** or **no lldp run** in global configuration mode.

Using Ansible for Cisco IOS XR

Following the same approach shown earlier for devices that run Cisco IOS/IOS XE software, a Cisco IOS XR device must be configured with the proper username, password, and SSH server in order for the managing host with Ansible to connect to it. Example 19-53 shows the preparation of the Cisco IOS XR router.

Note

Refer to the Figure 19-5, earlier in this chapter, for the topology information used in the following examples.

Example 19-53 *Preparing a Cisco IOS XR Device to Be Managed Using Ansible*

```
RP/0/0/CPU0:ios(config)#show configuration
Mon Jan  7 07:54:50.772 UTC
Building configuration...
!! IOS XR Configuration 6.5.1
hostname XR2
vrf mgmt
 address-family ipv4 unicast
 !
!
line console
 exec-timeout 0 0
!
control-plane
 management-plane
  out-of-band
   vrf mgmt
   interface MgmtEth0/0/CPU0/0
    allow SSH
   !
  !
 !
!
interface MgmtEth0/0/CPU0/0
 vrf mgmt
 ipv4 address 192.168.141.52 255.255.255.0
!
ssh server v2
ssh server vrf mgmt
end
RP/0/0/CPU0:ios(config)#commit
Mon Jan  7 07:55:16.530 UTC
RP/0/0/CPU0:XR2(config)#exit
RP/0/0/CPU0:XR2#crypto key generate rsa general-keys
Mon Jan  7 07:55:58.607 UTC
The name for the keys will be: the_default
% You already have keys defined for the_default
Do you really want to replace them? [yes/no]: yes
  Choose the size of the key modulus in the range of 512 to 4096 for your General
Purpose Keypair. Choosing a key modulus greater than 512 may take a few minutes.
```

```
How many bits in the modulus [2048]: 2048

Generating RSA keys ...

Done w/ crypto generate keypair

[OK]
```

The process shown in Example 19-53 is a bit different for the process used for CSR2 (refer to Example 19-41), which has no candidate configuration. Because Cisco IOS XR does have candidate configuration, Example 19-53 shows the fully prepared candidate configuration and commit process rather than per-line entry. In addition, in Cisco IOS XR there is management VRF instance configured, and the OOB management port MgmtEth0/0/CPU0/0 has been assigned to that instance. In Example 19-53, there is no username configured because Cisco IOS devices have a built-in account with cisco as both the username and password.

Note

The Cisco IOS XE examples earlier in this chapter have no management VRF instance configuration, though you could do it as well. Whether you use a VRF instance configuration depends on your network design.

When the configuration is committed and keys are generated, you should check the connectivity from the managing host to XR2, as shown in Example 19-54.

Example 19-54 *Verifying That the Cisco IOS XR Device Is Ready to Be Managed Using Ansible*

```
$ ping XR2 -c 1

PING XR2 (192.168.141.52) 56(84) bytes of data.

64 bytes from XR2 (192.168.141.52): icmp_seq=1 ttl=255 time=1.23 ms



--- XR2 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 1.237/1.237/1.237/0.000 ms




$ ssh cisco@XR2

Password:



RP/0/0/CPU0:XR2#
```

You can see in this output that XR2 is ready to be managed by Ansible. Recall from Example 19-43 that the Ansible hosts file has been updated for all network elements in the lab topology. Therefore, you don't need to update it again at this point.

Operational Data Verification Using the iosxr_command Module

In the same manner as for Cisco IOS XE devices, the review of the Ansible modules for Cisco IOS XR starts with the Ansible module **iosxr_command**, which is used for verification of operational information. You are already a bit familiar with it, as you used this module earlier in this chapter. Example 19-55 reminds you how it works to collect the same information collected for Cisco IOS XE earlier in this chapter.

Example 19-55 *Using the Ansible Module **iosxr_command** to Collect **show** Output*

```
$ cat npf_example_iosxr_command.yml

---

- hosts: iosxr

  connection: network_cli

  gather_facts: yes


  tasks:

      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'

        iosxr_command:

            commands:

                - show inventory

                - show ip interface brief

        register: iosxr_output


      - name: VERIFICATION OF THE OUTPUT

        debug:

            msg: "{{ iosxr_output.stdout_lines }}"

...
```

```
$ ansible-playbook npf_example_iosxr_command.yml --limit=XR2


PLAY [iosxr]
*******************************************************************************


TASK [Gathering Facts]
*********************************************************************
ok: [XR2]


TASK [COLLECTING OPEPRATION INFORMATION FROM 'iosxr']
****************************************
ok: [XR2]


TASK [VERIFICATION OF THE OUTPUT]
*********************************************************
ok: [XR2] => {
    "msg": [
        [
            "NAME: \"0/0/CPU0\", DESCR: \"Route Processor type (16, 0)\"",
            "PID: IOSXRV, VID: V01, SN: N/A"
        ],
        [
            "Interface                      IP-Address      Status          Protocol
Vrf-Name",
            "MgmtEth0/0/CPU0/0              192.168.141.52  Up              Up
mgmt    ",
            "GigabitEthernet0/0/0/0         unassigned      Shutdown        Down
default ",
            "GigabitEthernet0/0/0/1         unassigned      Shutdown        Down
default"
        ]
    ]
}


PLAY RECAP
********************************************************************************
XR2                        : ok=3    changed=0    unreachable=0    failed=0
```

If you compare this example with Example 19-45, you can see that the only difference is that now you are using the prefix **iosxr_** instead of the prefix **ios_**. You can see the similarities in the Ansible module structures, and you can see that the Cisco IOS XE and Cisco IOS XR commands are the same.

Because the credentials and the **ansible_network_os** value are stored in the Ansible hosts file (refer to Example 19-43), connectivity to XR2 is established using the **network_cli** connection plug-in, and no additional information needs to be provided because **gather_facts** is enabled.

As you did earlier, you can now use the **iosxr_command** module again to perform some interactive commands at the CLI level. Example 19-56 shows how to use this module to clear counters.

Example 19-56 *Using the Ansible Module **iosxr_command** to Clear Counters*

```
$ cat npf_example_iosxr_command_clear.yml
---
- hosts: iosxr
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
```

```
        iosxr_command:
            commands:
                - command: show interfaces MgmtEth 0/0/CPU0/0 | inc packets
                - command: clear counters interface MgmtEth 0/0/CPU0/0
                  prompt: '[confirm]'
                  answer: "\r"
                - command: show interfaces MgmtEth 0/0/CPU0/0 | inc packets
        register: iosxr_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ iosxr_output.stdout_lines }}"
...
```

```
$ ansible-playbook npf_example_iosxr_command_clear.yml --limit=XR2

PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'iosxr']
****************************************
ok: [XR2]

TASK [VERIFICATION OF THE OUTPUT]
************************************************************
ok: [XR2] => {
    "msg": [
        [
            "5 minute input rate 0 bits/sec, 0 packets/sec",
            "  5 minute output rate 0 bits/sec, 0 packets/sec",
            "     132 packets input, 14832 bytes, 0 total input drops",
            "     Received 1 broadcast packets, 0 multicast packets",
            "     107 packets output, 18008 bytes, 0 total output drops",
            "     Output 0 broadcast packets, 0 multicast packets"
        ],
        [
            "Clear \"show interface\" counters on this interface [confirm]"
        ],
        [
            "5 minute input rate 0 bits/sec, 0 packets/sec",
            "  5 minute output rate 0 bits/sec, 0 packets/sec",
            "     3 packets input, 286 bytes, 0 total input drops",
            "     Received 0 broadcast packets, 0 multicast packets",
            "     2 packets output, 316 bytes, 0 total output drops",
            "     Output 0 broadcast packets, 0 multicast packets"
        ]
    ]
}
```

```
PLAY RECAP
*********************************************************************************
XR2                        : ok=3    changed=0    unreachable=0    failed=0
```

Example 19-56 shows the process that occurs when the interface's counters are collected and then cleared and collected again to verify that clearance was successful.

Note

Example 19-45 and the discussion of that example provide details about the **command**, **prompt**, and **answer** entries, so refer to that example and text for clarification.

General Configuration Using the iosxr_config Module

For configuration of Cisco IOS XR devices using Ansible, there is a module called **iosxr_config** that you need to understand. This module allows you to execute a predefined set of CLI commands in a configuration context, which means you need to be familiar with the CLI syntax of Cisco IOS XR. Earlier in this chapter, you saw how to automate the implementation of the configuration for Cisco IOS XE and verify it, so this section shows only the relevant playbooks and not execution and verification. Example 19-57 shows the simplest way to configure the IOS XR interface without any variables and with all values hardcoded in the playbook.

Example 19-57 *Using the Ansible Module **iosxr_config** Without Variables*

```
$ cat npf_example_iosxr_config_simple.yml
---
- hosts: iosxr
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: CONFIGURING THE INTERFACE
        iosxr_config:
            lines:
                - ip address 10.0.10.22/24
                - ipv6 address fc00:10:0:10::22/64
                - no shutdown
            parents:
                - interface GigabitEthernet0/0/0/0
...
```

If you compare Example 19-57 with the Cisco IOS XE configuration in Example 19-47, you see only two differences. The first difference is the format of the IP address, which is not related to Ansible but rather to the CLI syntax of IOS XR. (You could also provide the IP address by using a subnet mask in dotted-decimal format.) The other difference is the name of the module; IOS-XR devices use **iosxr_config** instead of **ios_config**, which is used with IOS/IOS XE devices. The similarity in the module structure allows you to focus on the content of your automation rather than on the format of different Ansible modules.

In situations where you need to provide repetitive actions, such as configuring multiple interfaces, you can use loops and Ansible variables together with the **iosxr_config** module, as shown in Example 19-58.

Example 19-58 *Using the Ansible Module **iosxr_config** with Loops and Variables*

```
$ cat npf_example_iosxr_config_extended.yml
---
- hosts: iosxr
  connection: network_cli
  gather_facts: yes

  vars:
      interfaces:
          - id: interface GigabitEthernet0/0/0/0
            ipv4:
                address: 10.0.10.22
                mask: 24
            state: no shutdown
          - id: interface GigabitEthernet0/0/0/1
            ipv4:
                address: 10.0.12.22
```

```
                mask: 24
            state: no shutdown
          - id: interface Loopback0
            ipv4:
                address: 10.0.0.22
                mask: 32
            state: no shutdown
            ipv6:
                address: fc00:10::22
                mask: 128
            state: no shutdown




  tasks:
      - name: CONFIGURING THE INTERFACE
        iosxr_config:
            lines:
              - {% if item.ipv4 is defined %}ip address {{ item.ipv4.address }}/{{
item.ipv4.mask }}{% endif %}
              - {% if item.ipv6 is defined %}ip address {{ item.ipv6.address }}/{{
item.ipv6.mask }}{% endif %}
              - "{{ item.state }}"
            parents: "{{ item.id }}"
        loop: "{{ interfaces }}"
...
```

You should be rather familiar with the contents of Example 19-58 because you have already learned about all the components used here. The module **iosxr_config** is looped over the entries from the dictionary **vars**, where all variables for the interfaces are stored. You can see one difference compared to **ios_config**: with **iosxr_config**, the **save_when** key is absent. It is absent because of the architecture of Cisco IOS XR, which has no concept of running and startup configurations; the configuration is saved only when it's been committed, which in the case of **iosxr_config** happens automatically. Otherwise, the structure of the variables in Example 19-58 is the same as the structure you have used for Cisco IOS XE. This illustrates the importance of being aware of platform specifics when you do the automation.

Example 19-58 introduces the concept of device abstraction. When a network device is abstracted, actions such as packet forwarding are completed in a generic way rather in terms of a particular device instance. From a network services perspective, it doesn't matter how the interfaces are internally called within the router or switch, as long as, for example, traffic can freely enter the port 1 on the face plane and exit port 2. The naming conventions for the ports in Cisco IOS XE, IOS XR, and NX-OS are different. Therefore, device abstraction impacts the configuration syntax of a particular instance of a network device but not its logic (see Example 19-59).

Example 19-59 *Using the Ansible Module **iosxr_config** with a Jinja2 Template*

```
$ cat XR2_vars.yml
---
routing:
    - protocol: ospf
      id: 1
      router_id: 10.0.0.22
      interfaces:
          - id: GigabitEthernet0/0/0/0
            type: point-to-point
            passive: false
            area: 0.0.0.0
          - id: GigabitEthernet0/0/0/1
            type: point-to-point
            passive: false
            area: 0.0.0.0
          - id: Loopback0
            passive: true
            area: 0.0.0.0
...
```

```
$ cat npf_template_iosxr_config_pro.j2
{% if routing is defined %}
{% for rp in routing %}
{% if rp.protocol == 'ospf' %}
router {{ rp.protocol }} {{ rp.id }}
    router-id {{ rp.router_id }}
    !
{% for r_if in rp.interfaces %}
    area {{ r_if.area }}
        interface {{ r_if.id }}
{% if r_if.type is defined %}
            network {{ r_if.type }}
{% endif %}
{% if r_if.passive %}
            passive
{% endif %}
        !
    !
{% endfor %}
!
{% endif %}
{% endfor %}
{% endif %}
```

```
$ cat npf_example_iosxr_config_pro.yml
---
- hosts: iosxr
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: IMPORTING VARIABLES
        include_vars:
            file: "{{ inventory_hostname }}_vars.yml"

      - name: TEMPLATING CONFIG
        template:
            src: npf_template_iosxr_config_pro.j2
            dest: "{{ inventory_hostname }}_temp.conf"

      - name: CONFIGURING THE DEVICE
        iosxr_config:
            src: "{{ inventory_hostname }}_temp.conf"
...
```

In Example 19-59, thanks to device abstraction, with the XR2 variable **XR2_vars.yml** has precisely the same structure as the similar variable for CSR2 (refer to Example 19-48). Example 19-59 does not change the variables to Cisco IOS XR syntax. Rather, the template **npf_template_iosxr_config_pro.j2** translates abstract modeling to the particular syntax of the Cisco IOS XR CLI in the same way that the template translates the syntax to the Cisco IOS XE CLI in Example 19-48. Both the module **include_vars** and the template are used without any changes from previous examples. The module **iosxr_config** also has the same structure as the module **ios_config** from earlier in this chapter in terms of providing a path to the configuration file in the key **src**.

Configuration Using Various iosxr_* Modules

Earlier in this chapter, the section "Configuration Using Various ios_* Modules" explains the concept of declarative management in the context of Cisco IOS XE. For Cisco IOS XR there are also a couple modules for declarative management, but their number is much smaller compared

to the number of modules for Cisco IOS XE or NX-OS. This might be due to the fact that, traditionally, the skill level of network engineers at service providers was higher than the skill level in enterprises or data centers, which is why the demand for declarative management is lower in enterprises or data centers. Alternatively, it might be that the automation approaches in a service provider are different, and there are not too many requirements for Ansible declarative management modules. In any case, Example 19-60 shows how the module **iosxr_interface** is used for configuration of interfaces in Cisco IOS XR.

Example 19-60 *Using the Ansible Module **iosxr_interface***

```
$ cat npf_example_iosxr_interface.yml

---

- hosts: iosxr

  connection: network_cli

  gather_facts: yes


  vars:

      interfaces:

          - id: GigabitEthernet0/0/0/0

            ipv4:

                address: 10.0.10.22

                mask: 24

            description: XR2<--->CSR2

          - id: GigabitEthernet0/0/0/1

            ipv4:

                address: 10.0.12.22

                mask: 24

            description: XR2<--->NX2

          - id: Loopback0

            ipv4:

                address: 10.0.0.22

                mask: 32

            description: RID


  tasks:

      - name: CONFIGURING THE INTERFACE

        iosxr_interface:

            name: "{{ item.id }}"

            enabled: true

            description: "{{ item.description }}"

        loop: "{{ interfaces }}"

...
```

You can see here that the variables and structure of **iosxr_interface** are the same as for CSR2 earlier in this chapter. However, there are no modules for declarative management for Cisco IOS XR that can configure the Layer 3 parts of the interface, such as IPv4 or IPv6 addresses. Also, you can see that **connection** is set to **network_cli**, which enables SSH connectivity with CLI commands. However, if a Cisco IOS XR device has a NETCONF server enabled and **connection** is set to **netconf**, the module **iosxr_interface** automatically translates the configuration in a NETCONF message by using the Cisco native YANG module.

The bottom line is that the modules **ios_command/ios_config** and **iosxr_command/iosxr_config** are almost identical, and this makes possible a unified approach to network automation using Ansible. This is especially true if you are using network device abstraction and proper templates to translate abstract network device models to particular implementations on Cisco IOS XE or Cisco IOS XR.

Using Ansible for Cisco NX-OS

Cisco NX-OS is a network operation system that is heavily used primarily in data centers but also in enterprise networks. Unlike Cisco IOS XE and IOS XR, NX-OS is built as an application on Linux. With NX-OS, it is possible to directly access the underlying Linux infrastructure and run shell or Python code directly on NX-OS switches, as discussed in Chapter 17, "Programming Cisco Devices." When using Ansible, however, you want to be able to remotely manage devices similarly to the way you would with IOS XE and IOS XR, leveraging network device abstraction.

To automate a Cisco NX-OS network element by using Ansible, you need the same activities and information are required as for IOS XE and IOS XR: credentials for remote access, IP connectivity, and an SSH server (see Example 19-61).

Example 19-61 *Preparing a Cisco NX-OS Device to Be Managed Using Ansible*

```
switch# configure terminal

switch(config)# hostname NX2

NX2(config)# no password strength-check

NX2(config)# feature privilege
```

```
NX2(config)# username cisco priv-lvl 15 password cisco

WARNING: Minimum recommended length of 8 characters.

WARNING: Password should contain characters from at least three of the following

classes: lower case letters, upper case letters, digits and special characters.

WARNING: it is too short

WARNING: Configuration accepted because password strength check is disabled

NX2(config)# interface mgmt 0

NX2(config-if)# vrf member management

NX2(config-if)# ip address 192.168.141.62/24

NX2(config-if)# exit

NX2(config)# ssh key rsa 2048 force

deleting old rsa key.....

ssh server is enabled, cannot delete/generate the keys

NX2(config)#

NX2(config)# exit

NX2# copy run startup-config

[#######################################] 100%

Copy complete, now saving to disk (please wait)...

Copy complete.
```

Example 19-61 shows the preparation of a freshly booted Cisco NX-OS device without any configuration applied. By default, NX-OS has enhanced security in terms of passwords, and to be able to use the insecure cisco/cisco credentials, you need to disable those security checks by issuing the **no password strength-check** command. In addition, with Cisco NX-OS, to be able use certain functionality, you need to explicitly enable specific features in the configuration context as the vast majority of features are disabled by default. For example, you need to enable **feature privilege** before you can use **priv-lvl** to define the proper access for the user. The next step is to assign an IP address and a VRF instance to the management interface. This is the final step because, by default, SSH is enabled on Cisco NX-OS devices, and if you try to enable it manually, you get an error indicating that the SSH server is enabled, and no new keys can be generated. The default key length is 1024 bytes, and if you want to improve security by extending the key length to 2048, you need to stop the SSH server, generate new keys, and then enable it again. (How to extend the key length is beyond the scope of this book.)

Example 19-43 shows the important Ansible information in the hosts file, including hostnames, credentials, and network OS values, and that information still applies with NX-OS. Example 19-62 shows the last check you need to do before starting to deal with NX2 through Ansible: You need to check reachability from the managing host.

Example 19-62 *Verifying That the Cisco NX-OS Device Is Ready to Be Managed Using Ansible*

```
$ ping NX2 -c 1

PING NX2 (192.168.141.62) 56(84) bytes of data.

64 bytes from NX2 (192.168.141.62): icmp_seq=1 ttl=255 time=0.660 ms



--- NX2 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 0.660/0.660/0.660/0.000 ms




$ ssh cisco@NX2

Password:



NX2#
```

The output shows that NX2 is reachable. Like CSR2 and XR2 earlier in this chapter, NX2 is now ready to be automated.

Operational Data Verification Using the nxos_command Module

As with Cisco IOS XE and Cisco IOS XR, we begin here with the collection of operational data using the module **nxos_command**, as shown in Example 19-63.

Example 19-63 *Using the Ansible Module **nxos_command** to Collect **show** Output*

```
$ cat npf_example_nxos_command.yml

---

- hosts: nexus

  connection: network_cli

  gather_facts: yes
```

```
  tasks:
      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        nxos_command:
            commands:
                - show inventory
                - show ip interface brief vrf all
        register: nxos_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ nxos_output.stdout_lines }}"
...
```

```
$ ansible-playbook npf_example_nxos_command.yml --limit=NX2

PLAY [nexus]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [NX2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'nxos']
****************************************
ok: [NX2]

TASK [VERIFICATION OF THE OUTPUT]
***********************************************************
ok: [NX2] => {
    "msg": [
        [
            "NAME: \"Chassis\",  DESCR: \"Nexus9000 9000v Chassis\"               ",
            "PID: N9K-9000v           ,  VID: V02 ,  SN: 99LE15NXK53          ",
            "",
            "NAME: \"Slot 1\",  DESCR: \"Nexus 9000v Ethernet Module\"           ",
            "PID: N9K-9000v           ,  VID: V02 ,  SN: 99LE15NXK53          ",
            "",
            "NAME: \"Fan 1\",  DESCR: \"Nexus9000 9000v Chassis Fan Module\"    ",
            "PID: N9K-9000v-FAN       ,  VID: V01 ,  SN: N/A                  ",
            "",
            "NAME: \"Fan 2\",  DESCR: \"Nexus9000 9000v Chassis Fan Module\"    ",
            "PID: N9K-9000v-FAN       ,  VID: V01 ,  SN: N/A                  ",
            "",
            "NAME: \"Fan 3\",  DESCR: \"Nexus9000 9000v Chassis Fan Module\"    ",
            "PID: N9K-9000v-FAN       ,  VID: V01 ,  SN: N/A"
        ],
        [
            "IP Interface Status for VRF \"default\"(1)",
            "Interface            IP Address      Interface Status",
            "",
            "IP Interface Status for VRF \"management\"(2)",
            "Interface            IP Address      Interface Status",
            "mgmt0                192.168.141.62  protocol-up/link-up/admin-up"
```

```
        ]
    ]
}

PLAY RECAP
*********************************************************************************
NX2                        : ok=3    changed=0    unreachable=0    failed=0
```

The module **nxos_command** plays the same role and has the same structure as **ios_command** for Cisco IOS XE and **iosxr_command** for IOS XR: It collects operational data by using **show** commands and performs some interactive commands in privileged mode. The commands themselves are provided in the **commands** dictionary. Because this module relies on the CLI commands, you need to know them. You can see in Example 19-63 that the command **show ip interface brief vrf all** is used to display IP addresses of all the interfaces, regardless of what VRF instances they belong to; recall from earlier in this chapter that for Cisco IOS or IOS XR, **show ip interface brief** is enough.

Example 19-64 provides details on issuing interactive commands in Cisco NX-OS.

Example 19-64 *Using the Ansible Module **nxos_command** for Interactive Commands in Privileged Mode*

```
$ cat npf_example_nxos_command_clear.yml
---
- hosts: nexus
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        nxos_command:
            commands:
                - command: show interface mgmt 0  | include packets
                - command: clear counters interface mgmt 0
                  prompt: '[confirm]'
                  answer: "\r"
                - command: show interface mgmt 0 | include packets
        register: nxos_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ nxos_output.stdout_lines }}"
...



$ ansible-playbook npf_example_nxos_command_clear.yml --limit=NX2


PLAY [nexus]
*********************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [NX2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'nxos']
****************************************
ok: [NX2]

TASK [VERIFICATION OF THE OUTPUT]
***********************************************************
ok: [NX2] => {
    "msg": [
```

```
        [
            "1 minute input rate 0 bits/sec, 0 packets/sec",
            "  1 minute output rate 32 bits/sec, 0 packets/sec",
            "    1204 input packets 1043 unicast packets 128 multicast packets",
            "    33 broadcast packets 148020 bytes",
            "    1039 output packets 801 unicast packets 236 multicast packets",
            "    2 broadcast packets 250243 bytes"
        ],
        [
            ""
        ],
        [
            "1 minute input rate 0 bits/sec, 0 packets/sec",
            "  1 minute output rate 0 bits/sec, 0 packets/sec",
            "    3 input packets 3 unicast packets 0 multicast packets",
            "    0 broadcast packets 306 bytes",
            "    2 output packets 2 unicast packets 0 multicast packets",
            "    0 broadcast packets 604 bytes"
        ]
    ]
}


PLAY RECAP
*********************************************************************************
NX2                        : ok=3    changed=0    unreachable=0    failed=0
```

Note that although the values of counters are reduced in Example 19-64, they are not zero. The reason for this is that the Ansible script interacts with NX2 via this management interface. Therefore, after the counters are cleared, you send the request to collect information, and NX2 responds to that request.

The interactive management of Cisco NX-OS from privileged mode is precisely the same as it is for other Cisco operating systems. In fact, Example 19-64was created by copying earlier examples and changing the module and registered variable name keys from **ios_** to **nxos_**. You can see that automation across different Cisco network operating systems is relatively easy if you know the CLI syntax and have adequately mastered the Ansible basics presented at the beginning of this chapter.

General Configuration Using the nxos_config Module

Automation of network devices running Cisco NX-OS starts with the **nxos_config** module, which enables you to automate the configuration of any service by using the CLI. You have already seen interface configuration for CSR2 and XR2, and now Example 19-65 shows interface configuration for Cisco NX-OS.

Example 19-65 *Using the Ansible Module **nxos_config** with Hardcoded Values*

```
$ cat npf_example_nxos_config_simple.yml
---
- hosts: nexus
  connection: network_cli
  gather_facts: yes


  tasks:
      - name: CONFIGURING THE INTERFACE
        nxos_config:
            lines:
                - no switchport
                - ip address 10.0.11.33/24
                - no shutdown
            parents:
                - interface Ethernet1/1
...
```

By this point in the chapter, you should readily understand what is happening in Example 19-65. The deviations from earlier examples in this chapter are minimal. With NX-OS, the first command in the **lines** dictionary, **no switchport**, is necessary because, by default, all ports on Cisco NX-OS devices are Layer 2 ports, and IP addresses can't be assigned unless they are changed to Layer 3 ports. Otherwise, the module **nxos_config** follows the same rules and has the same structure as **ios_config** and **iosxr_config**.

Example 19-66 shows the usage of loops and variables in conjunction with **nxos_config**. Note again how similar this is to the IOS XE and IOS XR examples presented earlier in this chapter.

Example 19-66 *Using the Ansible Module **nxos_config** with Loops and Variables*

```
$ cat npf_example_nxos_config_extended.yml
---
- hosts: nexus
  connection: network_cli
  gather_facts: yes

  vars:
      interfaces:
          - id: interface Ethernet 1/1
            mode: no switchport
            ipv4:
                address: 10.0.11.33
                mask: 24
            state: no shutdown
          - id: interface Ethernet 1/2
            mode: no switchport
            ipv4:
                address: 10.0.12.33
                mask: 24
            state: no shutdown
          - id: interface Loopback0
            mode:
            ipv4:
                address: 10.0.0.33
                mask: 32
            state: no shutdown

  tasks:
      - name: CONFIGURING THE INTERFACE
        nxos_config:
            lines:
                - "{{ item.mode }}"
                - ip address {{ item.ipv4.address }}/{{ item.ipv4.mask }}
                - "{{ item.state }}"
            parents: "{{ item.id }}"
            save_when: modified
        loop: "{{ interfaces }}"
...
```

In addition to providing the command **no switchport** for the physical interfaces, you need to add the variable **mode** to each interface. However, the **no switchport** command is not applicable to loopback interfaces, so its value is empty for those interfaces. The rest of the content and keys in the module **nxos_config** are precisely the same as for Cisco IOS XE.

The most scalable option for automation with Ansible is the use of Jinja2 templates. Earlier examples show this option for Cisco IOS XE and IOS XR, and Example 19-67 shows this option for Cisco NX-OS.

Example 19-67 *Using the Ansible Module **nxos_config** with Jinja2 Templates*

```
$ cat NX2_vars.yml
---
routing:
    - protocol: ospf
      id: 1
      router_id: 10.0.0.33
      interfaces:
```

```
          - id: Ethernet1/1
            type: point-to-point
            passive: false
            area: 0.0.0.0
          - id: Ethernet1/2
            type: point-to-point
            passive: false
            area: 0.0.0.0
          - id: Loopback0
            passive: true
            area: 0.0.0.0
...
```

```
$ cat npf_template_nxos_config_pro.j2
{% if routing is defined %}
{% for rp in routing %}
{% if rp.protocol == 'ospf' %}
feature ospf
!
router {{ rp.protocol }} {{ rp.id }}
    router-id {{ rp.router_id }}
!
{% for r_if in rp.interfaces %}
interface {{ r_if.id }}
    ip router {{ rp.protocol }} {{ rp.id }} area {{ r_if.area }}
{% if r_if.type is defined %}
    ip {{ rp.protocol }} network {{ r_if.type }}
{% endif %}
{% if r_if.passive and 'Loopback' not in r_if.id %}
    ip {{ rp.protocol }} passive-interface
{% endif %}
!
{% endfor %}
{% endif %}
{% endfor %}
{% endif %}
```

```
$ cat npf_example_nxos_config_pro.yml
---
- hosts: nexus
  connection: network_cli
  gather_facts: yes

  tasks:
      - name: IMPORTING VARIABLES
        include_vars:
            file: "{{ inventory_hostname }}_vars.yml"

      - name: TEMPLATING CONFIG
        template:
            src: npf_template_nxos_config_pro.j2
            dest: "{{ inventory_hostname }}_temp.conf"
```

```
    - name: CONFIGURING THE DEVICE
      nxos_config:
          src: "{{ inventory_hostname }}_temp.conf"
          save_when: modified
...
```

Thanks to network device abstraction, the variables file **NX2_vars.yml** is to the same as for CSR or XR2, but the interface names are different. However, the template **npf_template_nxos_config_pro.j2** is adapted for NX-OS syntax, including activation of **feature ospf**, which is disabled by default. The playbook that ultimately executes the configuration of OSPF on NX-OS uses the module **nxos_config** with the key **src** pointing to the templated configuration.

Configuration Using Various nxos_* Modules

The number of modules for declarative management for Cisco NX-OS is much bigger than the number for Cisco IOS XE. (Remember that for Cisco IOS XR there almost no such modules.) Example 19-68 shows how to use the **nxos_interface** and **nxos_l3_interface** modules to configure physical and IP parameters of an interface.

Example 19-68 *Using the Ansible Modules **nxos_interface** and **nxos_l3_interface***

```
$ cat npf_example_nxos_interface_l3.yml
---
- hosts: nexus
  connection: network_cli
  gather_facts: yes

  vars:
      interfaces:
          - id: Ethernet1/1
            mode: layer3
            ipv4:
                address: 10.0.11.33
                mask: 24
            description: NX2<--->CSR2
          - id: Ethernet1/2
            mode: layer3
            ipv4:
                address: 10.0.12.33
                mask: 24
            description: NX2<--->XR2
          - id: Loopbacka0
            mode: layer3
            ipv4:
                address: 10.0.0.33
                mask: 32
            description: RID

  tasks:
      - name: CONFIGURING THE INTERFACE
        nxos_interface:
            name: "{{ item.id }}"
            admin_state: up
            description: "{{ item.description }}"
            mode: "{{ item.mode }}"
        loop: "{{ interfaces }}"

      - name: CONFIGURE THE INTERFACE (L3)
        nxos_l3_interface:
            name: "{{ item.id }}"
```

```
            ipv4: "{{ item.ipv4.address }}/{{ item.ipv4.mask }}"

        loop: "{{ interfaces }}"

...
```

All the **nxos_*** modules for declarative management are designed to work solely with Cisco NX-OS network devices, and the specifics in the NX-OS operation must be reflected. In Example 19-68, you can see that the **vars** dictionary contains a new entry called **mode** with the value **layer3**. In the context of the **nxos_interface** module, this value is responsible for issuing the **no switchport** command. The good thing is that the declarative modules have internal validation mechanisms. For example, the presence of the **mode** key even for the loopback interface, where there is no **switchport** command at all, doesn't impact the execution, and the configuration can be applied successfully.

The last module related to declarative management that is covered in this book, is **nxos_lldp**. Example 19-69 shows this module used to enable LLDP on NX-OS.

Example 19-69 *Using the Ansible Module* ***nxos_lldp***

```
$ cat npf_example_nxos_lldp.yml

---

- hosts: nexus

  connection: network_cli

  gather_facts: yes


  tasks:

      - name: ENABLING LLDP

        nxos_lldp:

            state: present

...
```

This module enables LLDP on Cisco NX-OS network devices in much the same way the declarative management module **ios_lldp** is used to enable LLDP on Cisco IOS XE. For Cisco IOS XR, LLDP is not yet enabled because there is no such declarative module. At the end of this chapter, you will learn how to enable it using NETCONF/YANG.

At this point in the chapter, you should be familiar with the modules used for each Cisco network operating system and how to enrich their output by using Jinja2 templates and other Ansible tools. As you have seen, various modules have similar structure, and you can easily extend your skills to other platforms if you have gained enough experience with any of them.

Using Ansible in Conjunction with NETCONF

The last topic for this chapter is the use of the NETCONF/YANG network management approach in conjunction with Ansible. There are several reasons you might want to use this approach. For example, you might want to test NETCONF/YANG management in general, and Ansible makes it possible to do this. In addition, you might want to transition to NETCONF/YANG operation (especially with open-standard YANG modules) to unify the configuration and operational data formats across different network operating systems of different vendors. You may have other reasons.

Before you can use NETCONF, it must be enabled on the network devices as it's disabled by default. Example 19-70 shows how to enable NETCONF on CSR2 and verify whether it's working from the managing host.

Note

Refer to the Figure 19-5, earlier in this chapter, for the topology information used in the following examples.

Example 19-70 *Preparing a Cisco IOS XE Host to Be Managed via NETCONF and Verifying That It Works*

```
CSR2# configure terminal

CSR2(config)# netconf ssh




$ ssh cisco@CSR2 -s netconf

Password:

<?xml version="1.0" encoding="UTF-8"?>

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

  <capabilities>

    <capability>urn:ietf:params:netconf:base:1.0</capability>

    <capability>urn:ietf:params:netconf:capability:writeable-running:1.0</capability>

    <capability>urn:ietf:params:netconf:capability:startup:1.0</capability>

    <capability>urn:ietf:params:netconf:capability:url:1.0</capability>

    <capability>urn:cisco:params:netconf:capability:pi-data-model:1.0</capability>

    <capability>urn:cisco:params:netconf:capability:notification:1.0</capability>

  </capabilities>

  <session-id>699879112</session-id>

</hello>
```

```
]]>]]>
```

IANA has allocated TCP port 830 to NETCONF, but some early deployments use other ports. The software release of CSR2 is 3.14.01.S, which is a bit old, and this explains why NETCONF is running on the TCP port, port 22. In Example 19-70, you can see that NETCONF is enabled using the **netconf ssh** command in Cisco IOS XE global configuration mode. NETCONF is running over SSH, so using port 22 port is not a problem in this case; however, to distinguish it from CLI-driven traffic, the NETCONF stream is marked with the string **netconf**, and you can see in Example 19-70 that the key **-s netconf** is added upon establishing the SSH session from the managing host to CSR2. In newer software versions, such as Cisco IOS XE 16.*, the configuration command is **netconf-yang**, and NETCONF runs on the TCP port, port 830, so it is important to check your version if you experience problems with the launch.

In order to manage network devices using NETCONF, it's essential to find proper YANG modules as they must be explicitly called in the NETCONF message. Chapter 13, "YANG," describes how to find the proper modules for all Cisco operating systems in different releases.

Another network function that is to be connected with the managing host is XR2 running Cisco IOS XR. Example 19-71 provides the details.

Example 19-71 *Preparing a Cisco IOS XR Host to Be Managed via NETCONF and Verifying That It Works*

```
RP/0/0/CPU0:XR2(config)#show conf

Sun Jan 13 17:16:39.792 UTC

Building configuration...

!! IOS XR Configuration 6.5.1.34I

control-plane

 management-plane

  out-of-band

   vrf mgmt

   interface MgmtEth0/0/CPU0/0

    allow NETCONF

   !

  !

 !

!

netconf-yang agent

 ssh

!

ssh server netconf vrf MGMT

end

RP/0/0/CPU0:XR2(config)#commit
```

```
$ ssh cisco@XR2 -s netconf -p 830

Password:

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

 <capabilities>

  <capability>urn:ietf:params:netconf:base:1.1</capability>

  <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>

  <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>

  <capability>urn:ietf:params:netconf:capability:validate:1.1</capability>

  <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.1</capability>

  <capability>http://cisco.com/ns/yang/Cisco-IOS-XR-aaa-aaacore-cfg?module=Cisco-IOS-

XR-aaa-aaacore-cfg&amp;revision=2018-09-04</capability>

! Further output is truncated for brevity
```

If for security reasons you don't limit access to the management plane, you can skip it and focus on the **netconf-yang agent ssh** and **ssh server netconf** commands, which are mandatory to enable NETCONF/YANG on Cisco IOS XR. As you learned in Chapter 14, "NETCONF and RESTCONF," the information about supported YANG modules is transferred in **capabilities** in the NETCONF hello message. Cisco IOS XR supports several hundred YANG modules, depending on the software version, and only a few capabilities are shown in Example 19-71. In Cisco IOS XR, the NETCONF protocol is running on its default port, port 830 (TCP), and you need to add the key **-p 830** when you check the connectivity to the NETCONF agent on XR2 from the managing host.

Last but not least, we look at the configuration of the NETCONF/YANG server on the NX2 network function running Cisco NX-OS. Example 19-72 shows how to enable NETCONF on NX2 and verify its operation.

Example 19-72 *Preparing a Cisco NX-OS Host to Be Managed via NETCONF and Verifying That It Works*

```
NX2# configure terminal

Enter configuration commands, one per line. End with CNTL/Z.
```

```
NX2(config)# feature netconf
```

```
$ ssh cisco@NX2 -s netconf -p 830
User Access Verification
Password:
<?xml version="1.0" encoding="UTF-8"?>

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.1</capability>
    <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.1</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.1</capability>
    <capability>http://cisco.com/ns/yang/cisco-nx-os-device?revision=2018-07-
17&amp;module=Cisco-NX-OS-device&amp;deviations=Cisco-NX-OS-device-
deviations</capability>
! Further output is truncated for brevity
```

In Cisco NX-OS, the NETCONF server is running on the default port, port 830 (TCP), just as it does on Cisco IOS XR. To enable it, you need to enable the **netconf** feature.

Operational Data Verification Using the netconf_get Module

To collect both configuration and operational data in this case, the first step is to collect the actual information by using the module **netconf_get**. Chapter 13 explains YANG module naming as well as where these modules can be found. Example 19-73 shows how you can extract the configuration of OSPF in the Cisco IOS XR native YANG module from XR2. (All these commands are executed from the managing host that has Ansible installed.)

Example 19-73 *Using the Ansible Module **netconf_get** to Collect Configuration Information in YANG*

```
$ ls ~/Yang/yang/vendor/cisco/xr/651/ | grep 'ospf'
Cisco-IOS-XR-ipv4-ospf-act.yang
Cisco-IOS-XR-ipv4-ospf-cfg.yang
Cisco-IOS-XR-ipv4-ospf-oper-sub1.yang
Cisco-IOS-XR-ipv4-ospf-oper-sub2.yang
Cisco-IOS-XR-ipv4-ospf-oper-sub3.yang
Cisco-IOS-XR-ipv4-ospf-oper.yang
Cisco-IOS-XR-ipv6-ospfv3-act.yang
Cisco-IOS-XR-ipv6-ospfv3-cfg.yang
Cisco-IOS-XR-ipv6-ospfv3-oper-sub1.yang
Cisco-IOS-XR-ipv6-ospfv3-oper.yang
```

```
$ cat ~/Yang/yang/vendor/cisco/xr/651/Cisco-IOS-XR-ipv4-ospf-cfg.yang | grep
'namespace'
  namespace "http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-ospf-cfg";
```

```
$ cat ~/Yang/yang/vendor/cisco/xr/651/Cisco-IOS-XR-ipv4-ospf-cfg.yang | grep '^
container'
  container ospf {
```

```
$ cat npf_example_iosxr_netconf_get_cfg.yml
---
- hosts: iosxr
```

```
  connection: netconf
  gather_facts: yes

  tasks:
      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        netconf_get:
            filter: <ospf xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-ospf-cfg"/>
            display: xml
        register: iosxr_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ iosxr_output.stdout_lines }}"
...
```

```
$ ansible-playbook npf_example_iosxr_netconf_get_cfg.yml --limit=XR2

PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'iosxr']
****************************************
ok: [XR2]

TASK [VERIFICATION OF THE OUTPUT]
************************************************************
ok: [XR2] => {
    "msg": {
    "msg": [
        "<data xmlns=\"urn:ietf:params:xml:ns:netconf:base:1.0\"
xmlns:nc=\"urn:ietf:params:xml:ns:netconf:base:1.0\">",
        "  <ospf xmlns=\"http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-ospf-cfg\">",
        "   <processes>",
        "    <process>",
        "     <process-name>1</process-name>",
        "     <default-vrf>",
        "      <router-id>10.0.0.22</router-id>",
        "      <area-addresses>",
        "       <area-address>",
        "        <address>0.0.0.0</address>",
        "        <running/>",
        "        <name-scopes>",
        "         <name-scope>",
        "          <interface-name>Loopback0</interface-name>",
        "          <running/>",
        "          <passive>true</passive>",
        "         </name-scope>",
        "         <name-scope>",
        "          <interface-name>GigabitEthernet0/0/0/0</interface-name>",
```

```
        "          <running/>",
        "          <network-type>point-to-point</network-type>",
        "         </name-scope>",
        "         <name-scope>",
        "          <interface-name>GigabitEthernet0/0/0/1</interface-name>",
        "          <running/>",
        "          <network-type>point-to-point</network-type>",
        "         </name-scope>",
        "        </name-scopes>",
        "       </area-address>",
        "      </area-addresses>",
        "     </default-vrf>",
        "     <start/>",
        "    </process>",
        "   </processes>",
        "  </ospf>",
        " </data>"
    ]
}

PLAY RECAP
*********************************************************************************
XR2                        : ok=3    changed=0    unreachable=0    failed=0
```

The output in Example 19-73 is a bit long, but it provides a lot of details about how to create an Ansible playbook. First of all, you need to find the module you are going to use by looking in the folder with all Cisco IOS XR modules that you cloned from GitHub with the YANG modules. Because OSPF was configured earlier, it makes sense to extract OSPF configuration; therefore, you use **ls** to list all the files with output modification or **grep 'ospf'** to list only OSPF-related modules. Because the goal here is to get the configuration, you use the module with **cfg** in the name. The next step is to find the relevant XML namespace value and the top YANG container, which you do by parsing the content of the YANG module **Cisco-IOS-XR-ipv4-ospf-cfg.yang** for the **namespace** entry and the top container (which starts two spaces from the beginning of the new line, so **grep '^ container'** is used). Having collected both of these pieces of information that are necessary to create the body of a NETCONF message, the Ansible playbook is created.

The connection plug-in **netconf** is used for NETCONF communication. The module that allows you to collect the information in YANG format through NETCONF is called **netconf_get**. The key **filter** is the most important one in this module, as it defines precisely which information is extracted from the device. The filter is composed as a top-level container of a YANG module together with the namespace of this module. The key **display** instructs the module **netconf_get**, which is the output format (either XML or JSON) of the collected information. This output is stored in the registered variable **iosxr_output**, which is displayed using the **debug** module in the second task. After the execution of the playbook, you can see the whole OSPF configuration in the Cisco IOS XR native YANG module, which is equivalent to the CLI commands shown in Example 19-59. This approach is useful for reverse engineering, such as if you need to create XML templates for configuration of Cisco IOS XR devices, along with the **pyang** tool (refer to Chapter 13).

As mentioned earlier, it's also possible to collect operational data by using YANG modules that contain a lot of details about the network device. Example 19-74 shows how to obtain live OSPF telemetry.

Example 19-74 *Using the Ansible Module **netconf_get** to Collect Operational Data in YANG*

```
$ cat npf_example_iosxr_netconf_get_oper.yml
---
- hosts: iosxr
  connection: netconf
  gather_facts: yes

  tasks:
      - name: COLLECTING OPEPRATION INFORMATION FROM '{{ ansible_network_os }}'
        netconf_get:
            filter: <ospf xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-ospf-oper"/>
            display: json
        register: iosxr_output

      - name: VERIFICATION OF THE OUTPUT
        debug:
            msg: "{{ iosxr_output.output.data }}"
...
```

```
$ ansible-playbook npf_example_iosxr_netconf_get_oper.yml -i hosts --limit=XR2

PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR2]

TASK [COLLECTING OPEPRATION INFORMATION FROM 'iosxr']
****************************************
ok: [XR2]

TASK [VERIFICATION OF THE OUTPUT]
***********************************************************
ok: [XR2] => {
    "msg": {
        "ospf": {
            "processes": {
                "process": {
                    "default-vrf": {
                        "adjacency-information": {
                            "neighbor-details": {
                                "neighbor-detail": [
                                    {
                                        "adjacency-sid-protected": "false",
                                        "interface-name": "GigabitEthernet0/0/0/0",
                                        "interface-type": "mgmt-if-point-to-point",
                                        "last-oob-time": "0",
                                        "lfa-neighbor-id": "0.0.0.0",
                                        "lfa-neighbor-revision": "0",
                                        "lfa-next-hop": "0.0.0.0",
                                        "neighbor-ack-list-count": "0",
                                        "neighbor-ack-list-high-watermark": "0",
                                        "neighbor-address": "10.0.0.11",
                                        "neighbor-area-id": "0.0.0.0",
                                        "neighbor-backup-designated-router-address": "0.0.0.0",
                                        "neighbor-bfd-information": {
                                            "bfd-intf-enable-mode": "0",
                                            "bfd-status-flag": "1"
                                        },
! Further output is truncated for brevity
```

Note

The details of how to create the filter in the **netconf_get** module are omitted here, as the process is shown in Example 19-73 example.

The information in Example 19-74 is actually telemetry data related to the OSPF process. In the full output, you can see all the neighbors, LSAs, routes, and other information related to OSPF. With telemetry, the collected data follows the chosen YANG module (in this case, Cisco native) and is transmitted using NETCONF. This is not the best way to collect telemetry, and Cisco is promoting gRPC/gNMI for telemetry collection, but if you don't have a telemetry collector, you can use NETCONF as explained in the next section. With telemetry data, it's handy to change the **display** value to **json**, as then you can save data directly in JSON format, which is easy to for any application to process.

General Configuration Using the netconf_config Module

You configure network functions via NETCONF in Ansible by using the **netconf_config** module. In Chapter 14, you learned about different

operations that NETCONF can perform. The **netconf_config** module performs the **edit-config** operation, and you need to specify only the body itself, which provides the configuration. Example 19-75 shows how to enable LLDP on the Cisco IOS XR network function by using NETCONF/YANG.

Example 19-75 *Using the **netconf_config** Module to Push a NETCONF/YANG Configuration*

```
$ pyang -f tree -p . Cisco-IOS-XR-ethernet-lldp-cfg.yang
module: Cisco-IOS-XR-ethernet-lldp-cfg
  +--rw lldp
     +--rw tlv-select!
     |  +--rw system-name
     |  |  +--rw disable?   boolean
     |  +--rw port-description
     |  |  +--rw disable?   boolean
     |  +--rw system-description
     |  |  +--rw disable?   boolean
     |  +--rw system-capabilities
     |  |  +--rw disable?   boolean
     |  +--rw management-address
     |  |  +--rw disable?   boolean
     |  +--rw tlv-select-enter       boolean
     +--rw holdtime?              uint32
     +--rw extended-show-width?   boolean
     +--rw enable-subintf?        boolean
     +--rw enable-mgmtintf?       boolean
     +--rw timer?                 uint32
     +--rw reinit?                uint32
     +--rw enable?                boolean

  augment /a1:interface-configurations/a1:interface-configuration:
    +--rw lldp!
       +--rw transmit
       |  +--rw disable?   boolean
       +--rw receive
       |  +--rw disable?   boolean
       +--rw lldp-intf-enter    boolean
       +--rw enable?            boolean


$ cat npf_example_iosxr_netconf_config_simple.yml
---
- hosts: iosxr
  connection: netconf
  gather_facts: yes

  tasks:
      - name: CONFIGURING LLDP '{{ ansible_network_os }}'
        netconf_config:
            content: |
                <config>
                    <lldp xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ethernet-lldp-
cfg">
                        <enable>true</enable>
                    </lldp>
                </config>
...
```

```
$ ansible-playbook npf_example_iosxr_netconf_config_simple.yml --limit=XR2


PLAY [iosxr]

*******************************************************************************


TASK [Gathering Facts]

*********************************************************************

ok: [XR2]


TASK [CONFIGURING LLDP 'iosxr']

************************************************************

changed: [XR2]


PLAY RECAP

********************************************************************************

XR2                        : ok=2    changed=1    unreachable=0    failed=0
```

The **pyang** tool is a very useful for configuring network elements with YANG. First of all, you build the tree to analyze how the message within the XML body should look. To enable LLDP on a Cisco IOS XR device, you can simply configure **lldp** in global configuration mode, without further details. The NETCONF/YANG equivalent of the command in the Ansible playbook **npf_example_iosxr_netconf_config_simple.yml** uses the key **content** from the **netconf_config** module. The **<config></config>** framing is mandatory, and you can use it a basis for all your configuration. The YANG module drives further internal structure within this framing. You might note that the top-level container together with the XML namespace is precisely the same as the **filter** entry used in the **netconf_get** module. This is true because when you extract the configuration, you extract exactly what is configured in a specific tree, and so the top-level container and XML namespaces are the same in both cases.

True automation of network management with Ansible is achieved through Jinja2 templates. It's possible to use them together with NETCONF as shown in Example 19-76.

Example 19-76 *Using the Ansible Module **netconf_config** to Push a NETCONF/YANG Configuration with Templates*

```
$ cat npf_template_iosxr_netconf.j2

<config>

{% if routing is defined %}

{% for rp in routing %}

{% if rp.protocol == 'ospf' %}

  <ospf xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-ospf-cfg">

    <processes>

      <process>

        <process-name>{{ rp.id }}</process-name>

        <default-vrf>

          <router-id>{{ rp.router_id }}</router-id>

          <area-addresses>

{% for r_if in rp.interfaces %}

            <area-address>

              <address>{{ r_if.area }}</address>

              <running/>

              <name-scopes>

                <name-scope>

                  <interface-name>{{ r_if.id }}</interface-name>

                  <running/>

{% if r_if.passive %}

                  <passive>true</passive>

{% endif %}

{% if r_if.type is defined %}

                  <network-type>{{ r_if.type }}</network-type>

{% endif %}

                </name-scope>
```

```
            </name-scopes>
          </area-address>
{% endfor %}
        </area-addresses>
      </default-vrf>
      <start/>
    </process>
  </processes>
 </ospf>
{% endif %}
{% endfor %}
{% endif %}
</config>
```

```
$ cat npf_example_iosxr_netconf_config_pro.yml
---
- hosts: iosxr
  connection: netconf
  gather_facts: yes

  tasks:
      - name: IMPORTING VARIABLES
        include_vars:
            file: "{{ inventory_hostname }}_vars.yml"

      - name: CONFIGURING THE DEVICE
        netconf_config:
            content: "{{ lookup('template', 'npf_template_iosxr_netconf.j2') }}"
...
```

```
$ ansible-playbook npf_example_iosxr_netconf_config_pro.yml --limit=XR2

PLAY [iosxr]
*******************************************************************************

TASK [Gathering Facts]
*********************************************************************
ok: [XR2]

TASK [IMPORTING VARIABLES]
*********************************************************************
ok: [XR2]

TASK [CONFIGURING THE DEVICE]
*********************************************************************
changed: [XR2]

PLAY RECAP
*******************************************************************************
XR2                        : ok=3    changed=1    unreachable=0    failed=0
```

The template in Example 19-76 creates a configuration in XML/YANG format to be sent via NETCONF; it is the same configure as in Example 19-60, where the template creates a set of CLI commands to configure the OSPF process. The key **content** from the **netconf_config** module

plug-in **lookup** is used to template the configuration directly to the network element, bypassing the creation of a temporary configuration file. Together, this template together and network abstraction enable true automation.

Summary

In this chapter, you have learned a lot about Ansible and management of network elements running Cisco IOS XE, IOS XR, and NX-OS, including the following:

• Ansible is an agentless automation framework.

• To connect to managed devices, Ansible relies on the inventory file, which contains information about the devices' hostnames, IP addresses, and other details.

• Ansible can operate an ad hoc manner (with individual commands) or using playbooks (that is, automation scripts).

• Ansible has myriad modules for various tasks, including modules for configuration of network elements.

• In a playbook, each task is based on a specific module; that is, there is one module for one task.

• A basic Ansible playbook defines the hosts to manage and the tasks to be executed on those hosts. It may also include variables, conditions, and loops to support sophisticated solutions.

• In Ansible, variables can be part of a playbook, they can exist in an external file, or they can be created dynamically during playbook execution.

• It is possible to control the execution of an Ansible playbook (for example, modifying the scope or passing additional variables) via additional arguments passed during launch.

• Jinja2 is a templating language that is used in Ansible. It includes filters, variables, loops, and conditions.

• Jinja2 can be used directly in Ansible playbooks inline or via a specific module leveraging external Jinja2 template files.

• Templates are handy for dynamic creation of configuration files used in network management. In conjunction with network device abstraction, Ansible and templates create a good framework for managing a multiple-OS network.

• Ansible has dedicated modules for each Cisco network OS. For configuration of network elements running Cisco IOS XE, IOS XR, and NX-OS, Ansible has modules for both imperative and declarative management.

• Via dedicated modules, Ansible supports NETCONF so that you can collect information or configure network elements by using NETCONF.

Part IX: Looking Ahead

Chapter 20. Looking Ahead

Congratulations, you now have a shiny new set of tools in your automation and programmability toolbox! You know what the tools are used for, and we've given you the basics of how to use them. So, are you done? Not by a long shot.

This short final chapter provides you with some direction on how to apply what you've already learned in this book, as well as some ideas on how to enhance and advance what you've learned. This chapter, and the book, concludes with some thoughts on what automation and programmability mean for your career.

Some Rules of Thumb

You probably know a number of aphorisms about tools. You probably know a number of aphorisms about tools. The right tool for the right job. If it ain't broke don't fix it. When all you have is a hammer, every problem looks like a nail. A bad workman quarrels with his tools. The mechanic that would perfect his work must first sharpen his tools. If all you have is a hammer, it's hard to eat spaghetti.

Interestingly, most of those old saws (sorry) apply to the tools you've just acquired. No matter what book or article you pick up on network programmability and automation, you're bound to encounter at least a few rules of thumb. They're called rules of thumb—handy approximations—because they are backed by collected practical wisdom.

Let's look at a few.

Automate the Painful Stuff

Want to know where to begin applying your new-found skills? Look at your daily duties. What causes you the most pain? What tasks are the most repetitive? What jobs are the most boring? These are not only candidates for scripting, they are likely to be, by merit of being simple (but boring / repetitive/ time-consuming) jobs that lend themselves to beginning scripting before you move on to more complex programming challenges.

Don't Automate a Broken Process

This is easily the most popular network automation rule of thumb. Be sure that the process you're automating is right, otherwise, your automation will only propagate a mistake faster and break your network faster.

Clean Up Your Network

This rule of thumb might strike you as less obvious than the first two. The less consistent your network is, the more complex your automation must be to deal with corner cases, "snowflake" designs, and "one-offs." Consolidate on the fewest protocols possible. Get rid of unused ACLs, prefix list entries, route maps, and so on. Make a template for your configurations and ensure that every node complies with it.

Find Your Sources of Truth

Eventually your automation is going to be extracting information from your network devices: config files, state, tables, telemetry, counters, and more. Often your automation uses this information to determine actionable data, identify symptoms and root causes for troubleshooting, or validate an intended change. The problem is, not every operating system reports this information in the same way. Even different versions of the same operating system might report information differently.

That's one of the advantages of APIs, as you've learned in this book: Normalizing data.

Distributed control planes mean many sources of truth and can complicate network automation. As mentioned in Chapter 1, "The Network Programmability and Automation Ecosystem," automation that has to reach out to every control plane and then analyze its information is problematic. This is where network abstraction comes in. Whether that abstraction is a graphical database, a repository of network data, or some other model of the network, the value of abstraction is that it becomes a single source of truth (SSoT) that you can trust for reliable automation.

An SSoT should represent both intent and network state. According to one study,[1] the most valuable data comprising the SSoT according to enterprises are:

[1] Shamus McGillicuddy, "A Network Source of Truth Promotes Trust in Network Automation," Enterprise Management Associates, May 2020.

1. Device state metrics

2. Data from shared services such as DDI (DNS, DHCP, and IPAM) and Active Directory

3. Network flows from sources such as NetFlow

4. Configuration data normalized into key/value pairs

Developing an SSoT might be as simple as purchasing an automation platform that creates its own. If you are building your own platform, things might be more complicated. This is where bringing in one or more coders with much deeper programming knowledge could be cost-effective.

Avoid Automation You Can't Reuse

You don't want to invest hours and hours developing a program you are only going to run once, unless that one-time run returns extraordinary benefits. You want automation that shows value over and over. If you have followed the first rule of thumb in this section, automating the painful

stuff, then by definition you're going to be writing code that you can use regularly.

Keep in mind that I'm not talking about occasionally whipping up a quick script for a single simple task that you might or might not use again. You've gained some skills, so use them. I'm talking about ensuring that the benefits of the program you write outweigh the time investment you put into writing it. You'll see more about cost/benefit analysis a little further down the list.

Document What You Do

Can anyone else use your program? Can anyone modify your program if they need to? Or is there a chance that you could return from a nice vacation to find that your associates are very unhappy with you because they couldn't figure out how to use that magic bit of automation you created? Even worse, is there a chance that they will desperately call you while you're still on vacation because no one in the office can get your software to work?

Understand What Level of Complexity You're Willing to Handle

I've already encouraged you to identify your pain points and automate them. But what if what's causing you pain is a complex workflow rather than a few simple tasks? In this case, you need to weigh what you're willing to invest to make the pain go away.

Any automation solution is going to impact you in three areas:

• People

• Time

• Money

These three factors don't exist all on their own; they influence each other. You might be able to reduce project time by adding more people, but that's going to cost you more money. You might be able to reduce cost by stretching the project time. You've participated in plenty of projects, and you have probably managed more than a few. You know this dynamic.

You might find that the people, time, and money required to address the complexity of your problem simply aren't worth the benefit and decide you can live with the pain.

Do a Cost/Benefit Analysis

As you've read through these rules of thumb, you've probably thought, "What he's telling me is to perform a cost/benefit analysis." Yep, I am.

But here's another perspective on the cost/benefit analysis: Adding automation to your network does not necessarily mean writing the program yourself. Certainly, knowing more about automation and programmability puts you in a superior position to understand the true magnitude of an automation project, but that understanding might lead you to consider alternatives to do-it-yourself (DIY) programming.

Here are your potential alternatives:

• DIY programming

• Hiring professional coders, either full time or part time

• Buying commercial ("shrink-wrapped") automation software

Each of these alternatives has pros and cons, and expense is not necessarily a determining factor here. For example, a professional coder might be able to develop in three weeks a solution that might take you—as a router jockey with some useful but basic programming experience—six months to develop. In this case, hiring the coder might be the more economical investment—especially if you are eliminating a serious operational bottleneck in three weeks instead of six months.

A commercial solution might also be cheaper than developing a solution in-house, and, if you have shopped wisely, it comes with knowledgeable software support. On the other hand, your custom-built solution is made for your specific network. You're not paying for features you don't need or a solution you have to shoe-horn into the way you run your network.

The point is, costs include more than just money, and benefits include more than just making your operations more efficient. Be sure you are considering all angles when considering which approach is right for you. And stating the blatantly obvious, keep in mind that your analysis is applicable to only one problem. DIY might be the best answer for one problem, professional coders for another, and commercial software for yet another.

What Do You Study Next?

An obvious answer to the question of what to study next is to expand on the basics we've given you in Python, Linux, APIs, modeling, and automation applications such as Ansible. You'll need them all.

But what about some ideas for branching out from the basics to take your automation toolbox to the next level? This section gives you a few ideas.

Model-Driven Telemetry

One of the practices of service management, discussed in Chapter 1, involves event and incident management. To maintain a healthy network and target minimal service downtime, it's common practice to "proactively" monitor your network in order to detect network events and incidents before they become service outages. If a network is designed properly, a network failure doesn't necessarily translate into a service outage unless the network failure is not resolved in a timely manner. For example, an interface failure on a router would not cause a service outage if

there is an alternate route for the traffic to take to reach its destination. However, if another interface fails, causing that alternate route to fail as well, the service becomes unavailable. This highlights the importance of monitoring your network.

Network monitoring is usually performed by collecting logs describing the different events taking place on the network. Some of these events are simple informational messages, and others reflect critical conditions such as interface failures or, worse, resource depletion on a device, such as high CPU or memory consumption.

Until recently, syslog messages and SNMP traps were the predominant methods used to receive logs at a network monitoring station. These logs and traps are typically received by a network management station and used to create visual indicators of the status of the different network components. For metrics (versus logs), a network engineer would use a CLI and the myriad **show** commands to retrieve operational data, such as interface counters and CPU utilization. For the purpose of the discussion, these methods are labeled "legacy" network management methods.

Battle-tested legacy protocols have served us well for decades, but a number of issues render them unfit for the next era of network management. When we compare legacy SNMP and syslog with model-based telemetry, the following points stands out:

• **Resource consumption**: SNMP and syslog tend to consume significant resources on the devices on which these protocols are used, especially if SNMP polling is used in a pull-based model.

• **Speed**: Legacy methods are notably slow. If you have ever attempted to use snmpwalk with a network device, you know what we mean.

• **Ability to be integrated**: SNMP traps, syslog text-based messages, and data received as output to a **show** command through the CLI need significant grooming before they can be integrated with other systems for processing. In additionally, the format of the output from CLI **show** commands is vendor and device specific.

• **Size**: In some cases, the data acquired using legacy methods is usually bulky, consuming substantial network bandwidth to transmit and storage to maintain. Note that this is not always the case; even with modern telemetry, data collection, storage, and transmission can be massive.

Streaming telemetry is used to receive metrics from a device at a certain cadence or event-triggered logs. It operates using push functionality and has much less impact on system resources than legacy protocols. The data retrieved using telemetry is encoded in either XML, JSON, or GPB, so the data is well formatted. GBP in particular provides the added value of being compact, especially with a flavor of GBP encoding called *compact GBP*. Data retrieved via telemetry references a data model to describe the data structure of the acquired data. This, coupled with standard encoding, makes the data ready for integration with other systems in the tool chain.

Any automation engineer involved with event, incident, or fault management (the naming varies depending on the referenced framework) should seriously start looking into model-based streaming telemetry and plan on migrating from legacy protocols as soon as feasible. Model-based streaming telemetry is the future of network monitoring (or *network visibility*, as some like to call it).

Containers: Docker and Kubernetes

The concept of virtualization is briefly covered in Chapter 1. Most engineers involved with automation have run into virtual machines at some point in their careers. However, a lesser number are familiar with the concept of containers. Although Docker containers are covered in some detail in Chapter 6, "Python Applications," the subject's importance does solicit a little more discussion here.

Whereas virtual machines abstract the host hardware and provide a virtualized version of different hardware components, such as CPU, memory, storage, and other peripheral devices, containers virtualize the operating system. More specifically, containers virtualize all software running on top of the operating system kernel.

Chapter 2, "Linux Fundamentals," introduces the concept of kernel space and user space software. Kernel space software is the core layer of the operating system that is closer to the hardware. A container is an autonomous entity that virtualizes all software components of the operating system running in the user space, including all the optional software packages that you choose to install. The container may also contain any user applications that you want to run. This means that all containers running on a system share the same hardware and kernel. So you may create and run a container that provides a virtualized version of Fedora and another container that provides a virtualized version of Ubuntu, both on the same host, since both use the same Linux kernel. However, you cannot run a Windows container on top of a Linux kernel. The difference between a virtual machine and a container should be somewhat clear by now.

A typical use case for a container is running an application that requires a specific version of an operating system with specific packages and libraries (dependencies) installed. You may need to run an Apache server and a custom Python application, both on the same host but each requiring a different OS version or the same OS version but a different set of dependencies. One solution is to run each application in its own container, where each container provides the user space OS components required to run the application software of choice.

But why containers? Containers provide a more efficient solution to virtualization than virtual machines. A container uses significantly fewer system resources than a VM, and it boots up much more quickly. You can run many more containers on a host than you can run VMs, which means they provide higher density and better utilization than VMs. Where hardware virtualization isn't required and a common kernel is not a showstopper, containers are usually the solution to the requirement of different OS environments. Different applications requiring different software dependencies to run is sometimes referred to as *dependency hell*.

The most popular container software today is Docker. Docker is open-source software and is considered the industry standard for containers. Docker has versions for Microsoft Windows Server as well as several Linux distros. Docker is composed of the core Docker Engine software and a Docker CLI for managing the Docker containers on the system. Current versions of Docker also provide an API to interface programmatically with the Docker Engine and automate tasks such as starting and stopping containers. Docker is covered in some detail in Chapter 6.

What happens when the number of containers you need to manage increases, and the applications in one container have dependencies on applications running in another container? For example, what if one container is the backup of another container, and it needs to start if the original container fails? What if you need to load balance traffic to two different containers? Several software applications on the market today

provide container management and orchestration. Two of them stand out: Kubernetes and Docker Swarm. Both of these applications are open source. And, as you might expect, each of the three biggest cloud providers—AWS, Microsoft Azure, and Google—has its own cloud-native container management and orchestration software.

If you plan to work with automation software of any kind, you will more likely than not run into containers. Make sure that this subject is on your roadmap for future studies.

Application Hosting

The majority of network vendors today base the network operating system (NOS) running on their equipment on Linux. In other words, the NOS runs as user space software over some Linux distro. As mentioned in several chapters in this book, and particularly in Chapter 17, "Programming Cisco Devices," IOX XR, IOS XE, and NX-OS all run on top of a Linux distro called Wind River. This device architecture, which is based on Linux, opens the door for a broad spectrum of innovations, including application hosting.

Software that needs to interact with a network device may be run on a remote host (that is, off-box). The software communicates with the router either over a legacy channel, such as SSH coupled with CLI commands, or over one of the APIs exposed by the device. But because most equipment is based on Linux, this software may also be run on the device itself (that is, on-box). This is application hosting. Application hosting generally can be implemented using two models:

• **Native**: This application hosting model involves developing software for the particular Linux distro running on the device. This is similar to developing software that runs natively on Windows or CentOS. The software runs only on that particular OS or OS family. This is typically done using the software development kit (SDK) provided by the equipment vendor for the particular Linux distro.

• **Containerized**: This application hosting model involves running software inside a container. For example, Cisco's IOS XR supports running on-box Docker or LXC containers. As explained in the previous section, a container constitutes the application software that you want to run as well as the OS components required to run the software. So, you may develop software that performs a global function on any Linux distro, such as collecting data from the router (not telemetry collection on the router itself, which can have serious performance impacts) or run custom functions not natively supported by IOS XR. Then you can package this software in a container, along with the OS it was developed on, and run this container on the IOS XR–based router or, better, on any network device that supports Docker containers.

The level of maturity that application hosting on network devices has reached provides innovation opportunities that were never feasible before. Cisco describes a number of very interesting use cases for using application hosting in its *Application Hosting Configuration Guides* available for a number of Cisco platforms.

One use case describes installing an iPerf server on one router and an iPerf client on another. The iPerf software measures path quality in terms of bandwidth, jitter, delay, and packet loss. Packet loss is then simulated on one path. The data provided by iPerf indicating low path quality via one interface on the router is used to increase the OSPF cost of that interface and, in the process, fail over the traffic to a different interface. This is only one example of what can be achieved using application hosting, but the sky is the limit.

Software Development Methodologies

An engineer wishing to automate the configuration of a VLAN on 100 switches may write a short Python script on a text editor such as **vim** or Notepad and then run the code on her local machine to get the task completed. However, when a team of software engineers want to develop a sophisticated multitier application, it is nearly impossible to use this method to develop software. Several software methodologies have been formulated and adopted over the past few decades to manage the process of software development in a team.

Generally speaking, the software development life cycle (SDLC) is divided into five main phases:

• Requirement specification and analysis

• Design

• Development or coding

• Testing

• Maintenance

A software development methodology is a framework that organizes the implementation of the SDLC. Several methodologies exist, such as Waterfall, Prototyping, Incremental and Iterative, Spiral, Rapid Application Development, Extreme Programming, V-Model Methodology, Scrum, Cleanroom, Dynamic Systems Development, and the Rational Unified Process.

Consider these examples:

• Waterfall is the most structured but also the most rigid methodology. It follows the analogy of a waterfall, where the water flows only downhill and never uphill. This means that each phase has a well-defined start and end and, once the phase is completed, there is no going back. So, for example, once the requirements are collected and analyzed, and the design phase starts, there is no going back to the previous phase and changing the requirements. Waterfall seemingly makes a lot of sense and is quite simple to follow, but it is hardly fit for today's development environments, where software products are in a constant state of development. Think of how frequently mobile applications such as Facebook and Google Maps get updated.

• Many of the other methodologies are described as Agile. An Agile methodology is one that conforms to the 12 principles listed in the Agile Manifesto. One of these principles describes the flexibility of such methodologies as follows: "Welcome changing requirements, even late in development." Agile methodologies also strive to produce a viable and shippable product as early as possible in the development process, and they depend on small software increments to that initial product. The principle states "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale." (The 12 principles are listed at https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/.) Agile methodologies are very flexible.

If you plan on getting involved in big software projects, whether as a software developer or as project manager, you need to understand the philosophies and practices involved in each of these methodologies in order to blend in with the rest of the team.

Miscellaneous Topics

The following are just a few of the areas that you might want to look into when you are done with this book:

• At some point in your software development progression, you will need some auxiliary tools to complement your programs. When the data you collect starts to be of considerable size, the data needs to be stored in a structured manner so that it can be both stored and searched efficiently. At that point, you will need to look into databases and search engines. Different types of databases exist today, and you might want to explore relational databases, graph databases, time-series databases, SQL and NoSQL databases, and software for storing and searching structured and unstructured data, such as Elasticsearch and Splunk.

• A massive potential area for further study is clouds and automation. Clouds, by definition, are platforms that automate the provisioning of resources and rent out these resources to customers. It is likely that no other area in IT utilizes automation more than cloud infrastructures. This applies to both public clouds such as AWS, Google Cloud, and Microsoft Azure and to private clouds that are custom built for specific customers.

• At some point in your programmability career, you will run into situations where you will need to learn about and/or employ advanced API features. When you do an API call and an error occurs, you need to implement proper error handling in order to isolate the users interfacing with your program from the internals of the API call. When you do an API call and the data you receive is significantly large, you may need to employ pagination to split the data received into different pages and then load subsequent pages as needed (or not). An API call may fail. You will need to look into API call retries. Caches and proxies are integral components of the Internet and some intranets. Your API calls may run into either of them.

• Chapters 17 and 18, "Programming non-Cisco Devices," cover a number of APIs exposed by Cisco and non-Cisco devices. Keep in mind that these are only examples. The vast majority of devices, systems, and networks developed and sold by Cisco and the other major vendors expose APIs. As a matter of fact, programmatic access has become part of the design requirements of any new product sold by Cisco. APIs are not an aftermath consideration anymore. Cisco is now developing products with programmatic access as a core feature. Make sure to check out the *Programmability Guide* for the product you are working on to understand the available APIs and how to work with them.

What Does All This Mean for Your Career?

When I was a young—well, younger— engineer in the late 1980s and early 1990s, teaching networking classes and working with ancient technologies like Token Ring, Frame Relay, ATM, and SNA, I came to realize that what I love about the networking field is that it was almost continually changing, and everyone was trying to figure out where the industry was going.

It was a time when one executive at a certain Ethernet hub vendor told me with great fervor that Ethernet hubs (Not switches. Hubs) were the future of the industry and would soon drive routers into obsolescence. That company ceased to exist in 1994, and while the vision of a world built around Ethernet hubs is now amusingly quaint, it highlights the struggle we all had with understanding what technologies were important to learn and what technologies were limping toward obsolescence. SNA and Token Ring were known to be on their way out, but ATM and Frame Relay were still new and cool. You had to know DECnet, IPX, XNS, and AppleTalk to pass the CCIE exam, but you didn't even have to know how to spell IPv6. Certain circles still thought OSI CLNS/CLNP would replace TCP/IP just as soon as the ISO standards committees started agreeing on something. I had to disabuse my networking students of the idea that OSPF was complicated and should only be used in complicated networks.

Abandoning my trip down memory lane and getting to the point: Networking in those days was kind of a circus. The extant technologies were not all that hard to learn, but there were *a lot* of them to learn. I was an eager learner. I loved it.

Predicting industry directions today is much easier because the role networking plays in our business and personal lives is clear. Anything that enhances that role—any technology that makes it easier to design, build, and operate networks that serve that role—is important.

I ended Chapter 1 posing a couple of questions from a couple of hypothetical engineers:

• **Older engineers**: "So I need to be a programmer now? Are software developers going to take my job if I don't?"

• **Newer engineers**: "I've invested enormous time and money into earning the certifications that will set me on the career path I want. Most of my study time has been spent configuring and troubleshooting through the CLI. Is all that a waste of time?"

The answers I gave to those questions at the end of Chapter 1:

• **Yes**, if you want to keep up with where the industry is going, you need to acquire some programming skills and understand the protocols supporting modern automation trends. This book and a mountain of other resources can help you get up to speed. If you are a seasoned engineer, none of this is different from what you have done your entire career: keeping up with new technologies by keeping up with the latest RFCs, reading the right trade journals and blogs, and attending industry events like CiscoLive! and your regional network operators' groups. If you're just starting out, you're already in deep learning mode, and you'll find that enhancing your growing skill set is not that hard at all. And I guarantee that it makes you more valuable as an engineer.

• **No**, automation and programmability do not mean that your jobs are going to be eliminated or taken over by software developers. Software developers' programming abilities go far beyond what is needed for networking, and for the most part, they know little about networking itself. You only have to know enough about programming to make your own job easier. Automating the mundane parts of your job just means you have more time for bringing your deep knowledge of networking to bear. The network is better for it, and you are most certainly the better for it.

Even if you need to bring software developers in to build automation code that is beyond your abilities or available time, you are the one who understands networking and must oversee their production. They need for you to understand what they do, and they need your superior knowledge of networking.

Learning network automation and programmability is essential to your future as a network engineer or architect. The more you know, the more valuable you are to your existing organization and the more marketable you are in the industry as a whole.

We all—Khaled, Vinit, and Jeff—hope that this book has provided you with a solid foundation for building those new skills and given you a clear-eyed insight into where your career is going.