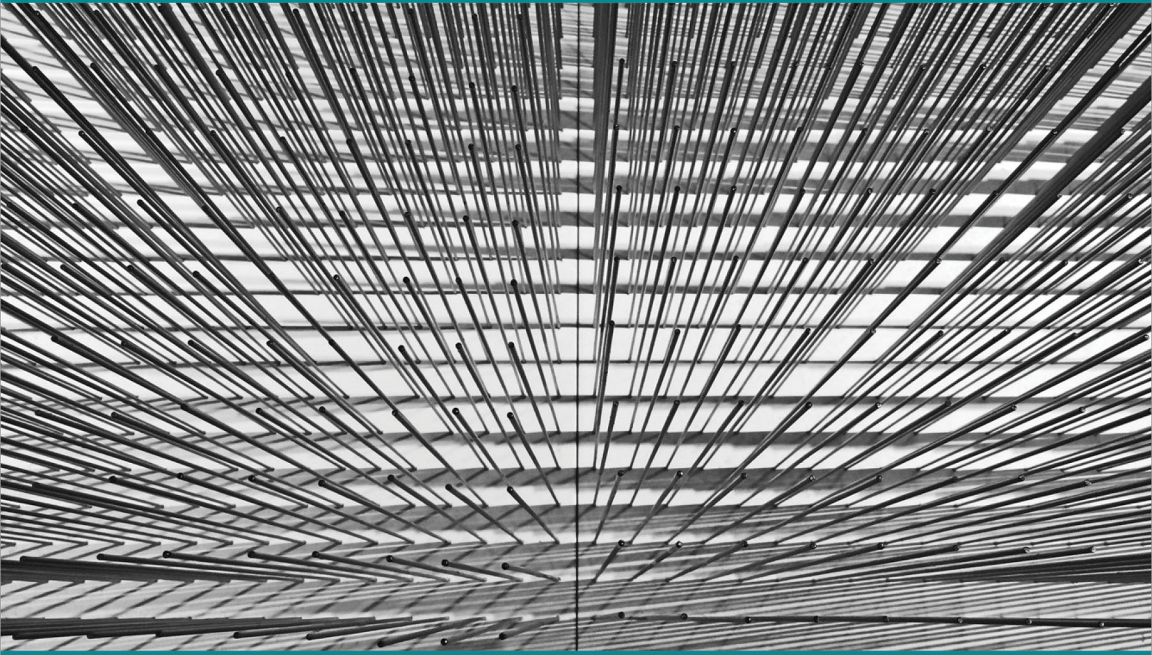
O'REILLY®

Network Automation with Ansible



Jason Edelman

Network Automation with Ansible

Jason Edelman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Network Automation with Ansible
by Jason Edelman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Brian Anderson and Courtney Allen
Production Editor: Nicholas Adams
Copyeditor: Amanda Kersey
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

March 2016: First Edition

Revision History for the First Edition
2016-03-07: First Release

978-1-491-93783-9

[LSI]

Table of Contents

CHAPTER 1

Network Automation

As the IT industry transforms with technologies from server virtualization to public and private clouds with self-service capabilities, containerized applications, and Platform as a Service (PaaS) offerings, one of the areas that continues to lag behind is the network.

Over the past 5+ years, the network industry has seen many new trends emerge, many of which are categorized as software-defined networking (SDN).

NOTE

SDN is a new approach to building, managing, operating, and deploying networks. The original definition for SDN was that there needed to be a physical separation of the control plane from the data (packet forwarding) plane, and the decoupled control plane must control several devices.

Nowadays, many more technologies get put under the *SDN umbrella*, including controller-based networks, APIs on network devices, network automation, whitebox switches, policy networking, Network Functions Virtualization (NFV), and the list goes on.

For purposes of this report, we refer to SDN solutions as solutions that include a network controller as part of the solution, and improve manageability of the network but don't necessarily decouple the control plane from the data plane.

One of these trends is the emergence of application programming interfaces (APIs) on network devices as a way to manage and operate these devices and truly offer machine to machine communication. APIs simplify the development process when it comes to automation and building network applications, providing more structure on how data is modeled. For example, when API-enabled devices return data in JSON/XML, it is structured and easier to work with as compared to CLI-only devices that return raw text that then needs to be manually parsed.

Prior to APIs, the two primary mechanisms used to configure and manage network devices were the command-line interface (CLI) and Simple Network Management Protocol (SNMP). If we look at each of those, the CLI was meant as a human interface to the device, and SNMP wasn't built to be a real-time programmatic interface for network devices.

Luckily, as many vendors scramble to add APIs to devices, sometimes *just because* it's a check in the box on an RFP, there is actually a great byproduct—enabling network automation. Once a true API is exposed, the process for accessing data within the device, as well as managing the configuration, is greatly simplified, but as we'll review in this report, automation is also possible using more traditional methods, such as CLI/SNMP.

NOTE

As network refreshes happen in the months and years to come, vendor APIs should no doubt be tested and used as key decision-making criteria for purchasing network equipment (virtual and physical). Users should want to know how data is modeled by the equipment, what type of transport is used by the API, if the vendor offers any libraries or integrations to automation tools, and if open standards/protocols are being used.

Generally speaking, network automation, like most types of automation, equates to doing things faster. While doing more faster is nice, reducing the time for deployments and configuration changes isn't always a problem that needs solving for many IT organizations.

Including speed, we'll now take a look at a few of the reasons that IT organizations of all shapes and sizes should look at gradually adopting network automation. You should note that the same principles apply to other types of automation as well.

Simplified Architectures

Today, every network is a unique snowflake, and network engineers take pride in solving transport and application issues with one-off network changes that ultimately make the network not only harder to maintain and manage, but also harder to automate.

Instead of thinking about network automation and management as a secondary or tertiary project, it needs to be included from the beginning as new architectures and designs are deployed. Which features work across vendors? Which extensions work across platforms? What type of API or automation tooling works when using particular network device platforms? When these questions get answered earlier on in the design process, the resulting architecture becomes simpler, repeatable, and easier to maintain *and* automate, all with fewer vendor proprietary extensions enabled throughout the network.

Deterministic Outcomes

In an enterprise organization, change review meetings take place to review upcoming changes on the network, the impact they have on external systems, and rollback plans. In a world where a human is touching the CLI to make those *upcoming changes*, the impact of typing the wrong command is catastrophic. Imagine a team with three, four, five, or 50 engineers. Every engineer may have his own way of making that particular *upcoming change*. And the ability to use a CLI or a GUI does not eliminate or reduce the chance of error during the control window for the change.

Using proven and tested network automation helps achieve more predictable behavior and gives the executive team a better chance at achieving deterministic outcomes, moving one step closer to having the assurance that the task is going to get done right the first time without human error.

Business Agility

It goes without saying that network automation offers speed and agility not only for deploying changes, but also for retrieving data from network devices as fast as the business demands. Since the advent of server virtualization, server and virtualization admins have had the ability to deploy new applications almost instantaneously. And the faster applications are deployed, the more questions are raised as to why it takes so long to configure a VLAN, route, FW ACL, or load-balancing policy.

By understanding the most common workflows within an organization and *why* network changes are really required, the process to deploy modern automation tooling such as Ansible becomes much simpler.

This chapter introduced some of the high-level points on why you should consider network automation. In the next section, we take a look at what Ansible is and continue to dive into different types of network automation that are relevant to IT organizations of all sizes.

CHAPTER 2

What Is Ansible?

Ansible is one of the newer IT automation and configuration management platforms that exists in the open source world. It's often compared to other tools such as Puppet, Chef, and SaltStack. Ansible emerged on the scene in 2012 as an open source project created by Michael DeHaan, who also created Cobbler and cocreated Func, both of which are very popular in the open source community. Less than 18 months after the Ansible open source project started, Ansible Inc. was formed and received $6 million in Series A funding. It became and is still the number one contributor to and supporter of the Ansible open source project. In October 2015, Red Hat acquired Ansible Inc.

But, what exactly is Ansible?

Ansible is a super-simple automation platform that is agentless and extensible.

Let's dive into this statement in a bit more detail and look at the attributes of Ansible that have helped it gain a significant amount of traction within the industry.

Simple

One of the most attractive attributes of Ansible is that you *DO NOT* need any special coding skills in order to get started. All instructions, or tasks to be automated, are documented in a standard, human-readable data format that anyone can understand. It is not uncommon to have Ansible installed and automating tasks in under 30 minutes!

For example, the following task from an Ansible playbook is used to ensure a VLAN exists on a Cisco Nexus switch:

```
- nxos_vlan: vlan_id=100 name=web_vlan
```

You can tell by looking at this almost exactly what it's going to do without understanding or writing any code!

NOTE The second half of this report covers the Ansible terminology (playbooks, plays, tasks, modules, etc.) in great detail. However, we have included a few brief examples in the meantime to convey key concepts when using Ansible for network automation.

Agentless

If you look at other tools on the market, such as Puppet and Chef, you'll learn that, by default, they require that each device you are automating have specialized software installed. This is *NOT* the case with Ansible, and this is the major reason why Ansible is a great choice for networking automation.

It's well understood that IT automation tools, including Puppet, Chef, CFEngine, SaltStack, and Ansible, were initially built to manage and automate the configuration of Linux hosts to increase the pace at which applications are deployed. Because Linux systems were being automated, getting agents installed was never a technical hurdle to overcome. If anything, it just delayed the setup, since now *N* number of hosts (the hosts you want to automate) needed to have software deployed on them.

On top of that, when agents are used, there is additional complexity required for DNS and NTP configuration. These are services that most environments do have already, but when you need to get something up fairly quick or simply want to see what it can do from a test perspective, it could significantly delay the overall setup and installation process.

Since this report is meant to cover Ansible for network automation, it's worth pointing out that having Ansible as an agentless platform is even more compelling to network admins than to sysadmins. Why is this?

It's more compelling for network admins because as mentioned, Linux operating systems are open, and anything can be installed on them. For networking, this is definitely not the case, although it is gradually changing. If we take the most widely deployed network operating system, Cisco IOS, as just one example and ask the question, *"Can third-party software be installed on IOS based platforms?"* it shouldn't come as a surprise that the answer is *NO*.

For the last 20+ years, nearly all network operating systems have been closed and vertically integrated with the underlying network hardware. Because it's not so easy to load an agent on a network device (router, switch, load balancer, firewall, etc.) without vendor support, having an automation platform like Ansible that was built from the ground up to be agentless and extensible is just what the doctor ordered for the network industry. We can finally start eliminating manual interactions with the network with ease!

Extensible

Ansible is also extremely extensible. As open source and code start to play a larger role in the network industry, having platforms that are extensible is a must. This means that if the vendor or community doesn't provide a particular feature or function, the open source community, end user, customer, consultant, or anyone else can *extend* Ansible to enable a given set of functionality. In the past, the network vendor or tool vendor was on the hook to provide the new plug-ins and integrations. Imagine using an automation platform like Ansible, and your network vendor of choice releases a new feature that you *really* need automated. While the network vendor or Ansible could in theory release the new plug-in to automate that particular feature, the great thing is, anyone from your internal engineers to your value-added reseller (VARs) or consultant could now provide these integrations.

It is a fact that Ansible is extremely extensible because as stated, Ansible was initially built to automate applications and systems. It is because of Ansible's extensibility that Ansible integrations have been written for network vendors, including but not limited to Cisco, Arista, Juniper, F5, HP, A10, Cumulus, and Palo Alto Networks.

CHAPTER 3

Why Ansible for Network Automation?

We've taken a brief look at what Ansible is and also some of the benefits of network automation, but why should Ansible be used for network automation?

In full transparency, many of the reasons already stated are what make Ansible such as great platform for automating application deployments. However, we'll take this a step further now, getting even more focused on networking, and continue to outline a few other key points to be aware of.

Agentless

The importance of an agentless architecture cannot be stressed enough when it comes to network automation, especially as it pertains to automating existing devices. If we take a look at all devices currently installed at various parts of the network, from the DMZ and campus, to the branch and data center, the lion's share of devices do *NOT* have a modern device API. While having an API makes things so much simpler from an automation perspective, an agentless platform like Ansible makes it possible to automate and manage those *legacy (traditional)* devices, for example, *CLI-based devices*, making it a tool that can be used in any network environment.

NOTE

If CLI-only devices are integrated with Ansible, the mechanisms as to how the devices are accessed for read-only and read-write operations occur through protocols such as telnet, SSH, and SNMP.

As standalone network devices like routers, switches, and firewalls continue to add support for APIs, SDN solutions are also emerging. The one common theme with SDN solutions is that they all offer a single point of integration and policy management, usually in the form of an SDN controller. This is true for solutions such as Cisco ACI, VMware NSX, Big Switch Big Cloud Fabric, and Juniper Contrail, as well as many of the other SDN offerings from companies such as Nuage, Plexxi, Plumgrid, Midokura, and Viptela. This even includes open source controllers such as OpenDaylight.

These solutions all simplify the management of networks, as they allow an administrator to start to migrate from box-by-box management to network-wide, single-system management. While this is a great step in the right direction, these solutions still don't eliminate the risks for human error during change windows. For example, rather than configure *N* switches, you may need to configure a single GUI that could take just as long in order to make the required configuration change—it may even be more complex, because after all, who prefers a GUI *over* a CLI! Additionally, you may possibly have different types of SDN solutions deployed per application, network, region, or data center.

The need to automate networks, for configuration management, monitoring, and data collection, does not go away as the industry begins migrating to controller-based network architectures.

As most software-defined networks are deployed with a controller, nearly all controllers expose a modern REST API. And because Ansible has an agentless architecture, it makes it extremely simple to automate not only legacy devices that may not have an API, but also software-defined networking solutions via REST APIs, all without requiring any additional software (agents) on the endpoints. The net result is being able to automate any type of device using Ansible with or without an API.

Free and Open Source Software (FOSS)

Being that Ansible is open source with all code publicly accessible on GitHub, it is absolutely free to get started using Ansible. It can literally be installed and providing value to network engineers in minutes. Ansible, the open source project, or Ansible Inc., do not require any meetings with sales reps before they hand over software either. That is stating the obvious, since it's true for all open source projects, but being that the use of open source, community-driven software within the network industry is fairly new and gradually increasing, we wanted to explicitly make this point.

It is also worth stating that Ansible, Inc. is indeed a company and needs to make money somehow, right? While Ansible is open source, it also has an enterprise product called Ansible Tower that adds features such as role-based access control (RBAC), reporting, web UI, REST APIs, multi-tenancy, and much more, which is usually a nice fit for enterprises looking to deploy Ansible. And the best part is that even Ansible Tower is *FREE* for up to 10 devices—so, at least you can get a taste of Tower to see if it can benefit your organization without spending a dime and sitting in countless sales meetings.

Extensible

We stated earlier that Ansible was primarily built as an automation platform for deploying Linux applications, although it has expanded to Windows since the early days. The point is that the Ansible open source project did not have the goal of automating network infrastructure. The truth is that the more the Ansible community understood how flexible and extensible the underlying Ansible architecture was, the easier it became to *extend* Ansible for their automation needs, which included networking. Over the past two years, there have been a number of Ansible integrations developed, many by industry independents such as Matt Oswalt, Jason Edelman, Kirk Byers, Elisa Jasinska, David Barroso, Michael Ben-Ami, Patrick Ogenstad, and Gabriele Gerbino, as well as by leading networking network vendors such as Arista, Juniper, Cumulus, Cisco, F5, and Palo Alto Networks.

Integrating into Existing DevOps Workflows

Ansible is used for application deployments within IT organizations. It's used by operations teams that need to manage the deployment, monitoring, and management of various types of applications. By integrating Ansible with the network infrastructure, it expands what is possible when new applications are turned up or migrated. Rather than have to wait for a new top of rack (TOR) switch to be turned up, a VLAN to be added, or interface speed/duplex to be checked, all of these network-centric tasks can be automated and integrated into existing workflows that already exist within the IT organization.

Idempotency

The term *idempotency* (pronounced item-potency) is used often in the world of software development, especially when working with REST APIs, as well as in the world of *DevOps* automation and configuration management frameworks, including Ansible. One of Ansible's beliefs is that all Ansible modules (integrations) should be idempotent. Okay, so what does it mean for a module to be idempotent? After all, this is a new term for most network engineers.

The answer is simple. Being idempotent allows the defined task to run one time or a thousand times without having an adverse effect on the target system, only ever making the change once. In other words, if a change is required to get the system into its desired state, the change is made; and if the device is already in its desired state, no change is made. This is unlike most traditional custom scripts and the copy and pasting of CLI commands into a terminal window. When the same command or script is executed repeatedly on the same system, errors are (sometimes) raised. Ever paste a command set into a router and get some type of error that invalidates the rest of your configuration? Was that fun?

Another example is if you have a text file or a script that configures 10 VLANs, the same commands are then entered 10 times *EVERY* time the script is run. If an idempotent Ansible module is used, the existing configuration is gathered first from the network device, and each new VLAN being configured is checked against the current configuration. Only if the new VLAN needs to be added (or changed—VLAN name, as an example) is a change or command actually pushed to the device.

As the technologies become more complex, the value of idempotency only increases because with idempotency, you shouldn't care about the *existing* state of the network device being modified, only the *desired* state that you are trying to achieve from a network configuration and policy perspective.

Network-Wide and Ad Hoc Changes

One of the problems solved with configuration management tools is configuration drift (when a device's desired configuration gradually drifts, or changes, over time due to manual change and/or having multiple disparate tools being used in an environment)—in fact, this is where tools like Puppet and Chef got started. Agents *phone home* to the head-end server, validate its configuration, and if a change is required, the change is made. The approach is simple enough. What if an outage occurs and you need to troubleshoot though? You usually bypass the management system, go direct to a device, find the fix, and quickly leave for the day, right? Sure enough, at the next time interval when the agent phones back home, the change made to fix the problem is overwritten (based on how the *master/head-end server* is configured). One-off changes should always be limited in highly automated environments, but tools that still allow for them are greatly valuable. As you guessed, one of these tools is Ansible.

Because Ansible is agentless, there is not a default push or pull to prevent configuration drift. The tasks to automate are defined in what is called an Ansible playbook. When using Ansible, it is up to the user to run the playbook. If the playbook is to be executed at a given time interval and you're not using Ansible Tower, you will definitely know how often the tasks are run; if you are just using the native Ansible command line from a terminal prompt, the playbook is run once and only once.

Running a playbook once by default is attractive for network engineers. It is added peace of mind that changes made manually on the device are not going to be automatically overwritten. Additionally, the scope of devices that a playbook is executed against is easily changed when needed such that even if a single change needs to automate only a single device, Ansible can still be used. The *scope* of devices is determined by what is called an Ansible inventory file; the inventory could have one device or a thousand devices.

The following shows a sample inventory file with two groups defined and a total of six network devices:

```
[core-switches]
dc-core-1
dc-core-2


[leaf-switches]
leaf1
leaf2
leaf3
leaf4
```

To automate all hosts, a snippet from your play definition in a playbook looks like this:

```
hosts: all
```

And to automate just one leaf switch, it looks like this:

```
hosts: leaf1
```

And just the core switches:

```
hosts: core-switches
```

NOTE

As stated previously, playbooks, plays, and inventories are covered in more detail later on this report.

Being able to easily automate one device or *N* devices makes Ansible a great choice for making those one-off changes when they are required. It's also great for those changes that are network-wide: possibly for shutting down all interfaces of a given type, configuring interface descriptions, or adding VLANs to wiring closets across an enterprise campus network.

CHAPTER 4

Network Task Automation with Ansible

This report is gradually getting more technical in two areas. The first area is around the details and architecture of Ansible, and the second area is about exactly what types of tasks can be automated from a network perspective with Ansible. The latter is what we'll take a look at in this chapter.

Automation is commonly equated with speed, and considering that some network tasks don't require speed, it's easy to see why some IT teams don't see the value in automation. VLAN configuration is a great example because you may be thinking, "How *fast* does a VLAN really need to get created? Just how many VLANs are being added on a daily basis? Do *I* really need automation?"

In this section, we are going to focus on several other tasks where automation makes sense such as device provisioning, data collection, reporting, and compliance. But remember, as we stated earlier, automation is much more than speed and agility as it's offering you, your team, and your business more predictable and more deterministic outcomes.

Device Provisioning

One of the easiest and fastest ways to get started using Ansible for network automation is creating device configuration files that are used for initial device provisioning and pushing them to network devices.

If we take this process and break it down into two steps, the first step is creating the configuration file, and the second is pushing the configuration onto the device.

First, we need to decouple the *inputs* from the underlying vendor proprietary syntax (CLI) of the config file. This means we'll have separate files with values for the configuration parameters such as VLANs, domain information, interfaces, routing, and everything else, and then, of course, a configuration template file(s). For this example, this is our standard golden template that's used for all devices getting deployed. Ansible helps bridge the gap between rendering the inputs and values with the configuration template. In less than a few seconds, Ansible can generate hundreds of configuration files predictably and reliably.

Let's take a quick look at an example of taking a current configuration and decomposing it into a template and separate variables (inputs) file.

Here is an example of a configuration file snippet:

```
hostname leaf1
ip domain-name ntc.com
!
vlan 10
   name web
!
vlan 20
   name app
!
vlan 30
   name db
!
vlan 40
   name test
!
vlan 50
   name misc
```

If we extract the input values, this file is transformed into a template.

NOTE

Ansible uses the Python-based Jinja2 templating language, thus the template called *leaf.j2* is a Jinja2 template.

Note that in the following example the *double curly braces* denote a variable.

The resulting template looks like this and is given the filename *leaf.j2*:

```
!
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
!
!
{% for vlan in vlans %}
vlan {{ vlan.id }}
  name {{ vlan.name }}
{% endfor %}
!
```

Since the double curly braces denote variables, and we see those values are not in the template, they need to be stored somewhere. They get stored in a variables file. A matching variables file for the previously shown template looks like this:

```
---
hostname: leaf1
domain_name: ntc.com
vlans:
  - { id: 10, name: web }
  - { id: 20, name: app }
  - { id: 30, name: db }
  - { id: 40, name: test }
  - { id: 50, name: misc }
```

This means if the team that controls VLANs wants to add a VLAN to the network devices, no problem. Have them change it in the variables file and regenerate a new config file using the Ansible module called `template`. This whole process is idempotent too; only if there is a change to the template or values being entered will a new configuration file be generated.

Once the configuration is generated, it needs to be *pushed* to the network device. One such method to push configuration files to network devices is using the open source Ansible module called `napalm_install_config`.

The next example is a sample playbook to *build and push* a configuration to network devices. Again, this playbook uses the `template` module to build the configuration files and the `napalm_install_config` to push them and activate them as the new running configurations on the devices.

Even though every line isn't reviewed in the example, you can still make out what is actually happening.

NOTE

The following playbook introduces new concepts such as the built-in variable `inventory_hostname`. These concepts are covered in Chapter 6.

```
---

  - name: BUILD AND PUSH NETWORK CONFIGURATION FILES
    hosts: leaves
    connection: local
    gather_facts: no

    tasks:
      - name: BUILD CONFIGS
      template:
        src=templates/leaf.j2
        dest=configs/{{inventory_hostname }}.conf

      - name: PUSH CONFIGS
        napalm_install_config:
          hostname={{ inventory_hostname }}
          username={{ un }}
          password={{ pwd }}
          dev_os={{ os }}
          config_file=configs/{{ inventory_hostname }}.conf
          commit_changes=1
          replace_config=0
```

This two-step process is the simplest way to get started with network automation using Ansible. You simply template your configs, build config files, and push them to the network device—otherwise known as the *BUILD and PUSH* method.

NOTE

Another example like this is reviewed in much more detail in "Ansible Network Integrations" on page 26.

Data Collection and Monitoring

Monitoring tools typically use SNMP—these tools poll certain management information bases (MIBs) and return data to the monitoring tool. Based on the data being returned, it may be more or less than you actually need. What if interface stats are being polled? You are likely getting back every counter that is displayed in a *show interface* command. What if you only need *interface resets* and wish to see these resets correlated to the interfaces that have CDP/LLDP neighbors on them? Of course, this is possible with current technology; it could be you are running multiple show commands and parsing the output manually, or you're using an SNMP-based tool but going between tabs in the GUI trying to find the data you actually need. How does Ansible help with this?

Being that Ansible is totally open and extensible, it's possible to collect and monitor the exact counters or values needed. This may require some up-front custom work but is totally worth it in the end, because the data being gathered is what you need, not what the vendor is providing you. Ansible also provides intuitive ways to perform certain tasks conditionally, which means based on data being returned, you can perform subsequent tasks, which may be to collect more data or to make a configuration change.

Network devices have *A LOT* of static and ephemeral data buried inside, and Ansible helps extract the bits you need.

You can even use Ansible modules that use SNMP behind the scenes, such as a module called `snmp_device_version`. This is another open source module that exists within the community:

```
- name: GET SNMP DATA
  snmp_device_version:
    host=spine
    community=public
    version=2c
```

Running the preceding task returns great information about a device and adds some level of discovery capabilities to Ansible. For example, that task returns the following data:

```
{"ansible_facts": {"ansible_device_os": "nxos", "ansible_device_vendor": "cisco", "ansible_device_version":
"7.0(3)I2(1)"}, "changed": false}
```

You can now determine what type of device something is without knowing up front. All you need to know is the read-only community string of the device.

Migrations

Migrating from one platform to the next is never an easy task. This may be from the same vendor or from different vendors. Vendors may offer a script or a tool to help with migrations. Ansible can be used to build out configuration templates for all types of network devices and operating systems in such a way that you could generate a configuration file for all vendors given a defined and common set of inputs (common data model). Of course, if there are vendor proprietary extensions, they'll need to be accounted for, too. Having this type of flexibility helps with not only migrations, but also disaster recovery (DR), as it's very common to have different switch models in the production and DR data centers, maybe even different vendors.

Configuration Management

As stated, configuration management is the most common type of automation. What Ansible allows you to do fairly easily is create *roles* to streamline the consumption of task-based automation. From a high level, a role is a logical grouping of reusable tasks that are automated against a particular group of devices. Another way to think about roles is to think about workflows. First and foremost, workflows and processes need to be understood before automation is going to start adding value. It's always important to start small and expand from there.

For example, a set of tasks that automate the configuration of routers and switches is very common and is a great place to start. But where do the IP addresses come from that are configured on network devices? Maybe an IP address management solution? Once the IP addresses are allocated for a given function and deployed, does DNS need to be updated too? Do DHCP scopes need to be created?

Can you see how the workflow can start small and gradually expand across different IT systems? As the workflow continues to expand, so would the role.

Compliance

As with many forms of automation, making configuration changes with any type of automation tool is seen as a risk. While making manual changes could arguably be riskier, as you've read and may have experienced firsthand, Ansible has capabilities to automate data collection, monitoring, and configuration building, which are all "read-only" and "low risk" actions. One *low risk* use case that can use the data being gathered is configuration compliance checks and configuration validation. Does the deployed configuration meet security requirements? Are the required networks configured? Is protocol XYZ disabled? Since each module, or integration, with Ansible returns data, it is quite simple to *assert* that something is *TRUE* or *FALSE*. And again, based on *it* being *TRUE* or *FALSE*, it's up to you to determine what happens next—maybe it just gets logged, or maybe a complex operation is performed.

Reporting

We now understand that Ansible can also be used to collect data and perform compliance checks. The data being returned and collected from the device by way of Ansible is up for grabs in terms of what you want to do with it. Maybe the data being returned becomes inputs to other tasks, or maybe you just want to create reports. Being that reports are generated from templates combined with the actual important data to be inserted into the template, the process to create and use reporting templates is the same process used to create configuration templates.

From a reporting perspective, these templates may be flat text files, markdown files that are viewed on GitHub, HTML files that get dynamically placed on a web server, and the list goes on. The user has the power to create the exact type of report she wishes, inserting the exact data she needs to be part of that report.

It is powerful to create reports not only for executive management, but also for the ops engineers, since there are usually different metrics both teams need.

CHAPTER 5

How Ansible Works

After looking at what Ansible can offer from a network automation perspective, we'll now take a look at how Ansible works. You will learn about the overall communication flow from an Ansible control host to the nodes that are being automated. First, we review how Ansible works *out of the box*, and we then take a look at how Ansible, and more specifically Ansible *modules*, work when network devices are being automated.

Out of the Box

By now, you should understand that Ansible is an automation platform. In fact, it is a lightweight automation platform that is installed on a single server or on every administrator's laptop within an organization. You decide. Ansible is easily installed using utilities such as pip, apt, and yum on Linux-based machines.

NOTE

The machine that Ansible is installed on is referred to as the *control host* through the remainder of this report.

The control host will perform all automation tasks that are defined in an Ansible playbook (don't worry; we'll cover playbooks and other Ansible terms soon enough). The important piece for now is to understand that a playbook is simply a set of automation tasks and instructions that gets executed on a given number of hosts.

When a playbook is created, you also need to define which hosts you want to automate. The mapping between the playbook and the hosts to automate happens by using what is known as an Ansible inventory file. This was already shown in an earlier example, but here is another sample inventory file showing two groups: `cisco` and `arista`:

```
[cisco]
nyc1.acme.com
nyc2.acme.com

[arista]
sfo1.acme.com
sfo2.acme.com
```

NOTE You can also use IP addresses within the inventory file, instead of hostnames. For these examples, the hostnames were resolvable via DNS.

As you can see, the Ansible inventory file is a text file that lists hosts and groups of hosts. You then reference a specific host or a group from within the playbook, thus dictating which hosts get automated for a given play and playbook. This is shown in the following two examples.

The first example shows what it looks like if you wanted to automate all hosts within the `cisco` group, and the second example shows how to automate just the *nyc1.acme.com* host:

```
---

  - name: TEST PLAYBOOK
    hosts: cisco

    tasks:
      - TASKS YOU WANT TO AUTOMATE
```

```
---

  - name: TEST PLAYBOOK
    hosts: nyc1.acme.com

    tasks:
      - TASKS YOU WANT TO AUTOMATE
```

Now that the basics of inventory files are understood, we can take a look at how Ansible (the control host) communicates with devices *out of the box* and how tasks are automated on Linux endpoints.

This is an important concept to understand, as this is usually different when network devices are being automated.

There are two main requirements for Ansible to work out of the box to automate Linux-based systems. These requirements are SSH and Python.

First, the endpoints must support SSH for transport, since Ansible uses SSH to connect to each target node. Because Ansible supports a pluggable connection architecture, there are also various plug-ins available for different types of SSH implementations.

The second requirement is how Ansible gets around the need to require an *agent* to preexist on the target node. While Ansible does not require a software agent, it does require an onboard Python execution engine. This execution engine is used to execute Python code that is transmitted from the Ansible control host to the target node being automated.

If we elaborate on this out of the box workflow, it is broken down as follows:

1. When an Ansible play is executed, the control host connects to the Linux-based target node using SSH.
2. For each task, that is, Ansible module being executed within the play, Python code is transmitted over SSH and executed directly on the remote system.
3. Each Ansible module upon execution on the remote system returns JSON data to the control host. This data includes information such as if the configuration changed, if the task passed/failed, and other module-specific data.
4. The JSON data returned back to Ansible can then be used to generate reports using templates or as inputs to subsequent modules.
5. Repeat step 3 for each task that exists within the play.
6. Repeat step 1 for each play within the playbook.

Shouldn't this mean that network devices should work out of the box with Ansible because they also support SSH? It is true that network devices do support SSH, but it is the first requirement combined with the second one that limits the functionality possible for network devices.

To start, most network devices do not support Python, so it makes using the default Ansible connection mechanism process a non-starter. That said, over the past few years, vendors have added Python support on several different device platforms. However, most of these platforms still lack the integration needed to allow Ansible to get direct access to a Linux shell over SSH with the proper permissions to copy over the required code, create temp directories and files, and execute the code on box. While all the parts are there for Ansible to work natively with SSH/Python *and* Linux-based network devices, it still requires network vendors to open their systems more than they already have.

NOTE

It is worth noting that Arista does offer native integration because it is able to drop SSH users directly into a Linux shell with access to a Python execution engine, which in turn does allow Ansible to use its default connection mechanism. Because we called out Arista, we need to also highlight Cumulus as working with Ansible's default connection mechanism, too. This is because Cumulus Linux is native Linux, and there isn't a need to use a vendor API for the automation of the Cumulus Linux OS.

Ansible Network Integrations

The previous section covered the way Ansible works by default. We looked at how Ansible sets up a connection to a device at the beginning of a *play*, executes tasks by copying Python code to the devices, executes the code, and then returns results back to the Ansible control host.

In this section, we'll take a look at what this process is when automating network devices with Ansible. As already covered, Ansible has a pluggable connection architecture. For *most* network integrations, the `connection` parameter is set to `local`. The most common place to make the connection type local is within the playbook, as shown in the following example:

```
---

- name: TEST PLAYBOOK
  hosts: cisco
  connection: local
  tasks:
    - TASKS YOU WANT TO AUTOMATE
```

Notice how within the play definition, this example added the `connection` parameter as compared to the examples in the previous section.

This tells Ansible not to connect to the target device via SSH and to just connect to the local machine running the playbook. Basically, this delegates the connection responsibility to the actual Ansible modules being used within the *tasks* section of the playbook. Delegating power for each type of module allows the modules to connect to the device in whatever fashion necessary; this could be NETCONF for Juniper and HP Comware7, eAPI for Arista, NX-API for Cisco Nexus, or even SNMP for traditional/legacy-based systems that don't have a programmatic API.

NOTE

Network integrations in Ansible come in the form of Ansible modules. While we continue to whet your appetite using terminology such as playbooks, plays, tasks, and modules to convey key concepts, each of these terms are finally covered in greater detail in Chapter 6 and Chapter 7.

Let's take a look at another sample playbook:

```
---

- name: TEST PLAYBOOK
  hosts: cisco
  connection: local

  tasks:
    - nxos_vlan: vlan_id=10 name=WEB_VLAN
```

If you notice, this playbook now includes a task, and this task uses the `nxos_vlan` module. The `nxos_vlan` module is just a Python file, and it is in this file where the connection to the Cisco NX-OS device is made using NX-API. However, the connection could have been set up using any other device API, and this is how vendors and users like us are able to build our own integrations. Integrations (modules) are typically done on a per-feature basis, although as you've already seen with modules like `napalm_install_config`, they can be used to *push* a full configuration file, too.

One of the major differences is that with the default connection mechanism, Ansible launches a persistent SSH connection to the device, and this connection persists for a given play. When the connection setup and teardown occurs within the module, as with many network modules that use `connection=local`, Ansible is logging in/out of the device on *every* task versus this happening on the play level.

And in traditional Ansible fashion, each network module returns JSON data. The only difference is the massaging of this data is happening locally on the Ansible control host versus on the target node. The data returned back to the playbook varies per vendor and type of module, but as an example, many of the Cisco NX-OS modules return back existing state, proposed state, and end state, as well as the commands (if any) that are being sent to the device.

As you get started using Ansible for network automation, it is important to remember that setting the connection parameter to local is taking Ansible out of the connection setup/teardown process and leaving that up to the module. This is why modules supported for different types of vendor platforms will have different ways of communicating with the devices.

CHAPTER 6

Ansible Terminology and Getting Started

This chapter walks through many of the terms and key concepts that have been gradually introduced already in this report. These are terms such as *inventory file*, *playbook*, *play*, *tasks*, and *modules*. We also review a few other concepts that are helpful to be aware of when getting started with Ansible for network automation.

Please reference the following sample inventory file and playbook throughout this section, as they are continuously used in the examples that follow to convey what each Ansible term means.

Sample inventory:

```
# sample inventory file
# filename inventory

[all:vars]
user=admin
pwd=admin

[tor]
rack1-tor1   vendor=nxos
rack1-tor2   vendor=nxos
rack2-tor1   vendor=arista
rack2-tor2   vendor=arista

[core]
core1
core2
```

Sample playbook:

```
---
# sample playbook
# filename site.yml

  - name: PLAY 1 - Top of Rack (TOR) Switches
    hosts: tor
    connection: local

    tasks:
      - name: ENSURE VLAN 10 EXISTS ON CISCO TOR SWITCHES
        nxos_vlan:
          vlan_id=10
          name=WEB_VLAN
          host={{ inventory_hostname }}
          username=admin
          password=admin
        when: vendor == "nxos"

      - name: ENSURE VLAN 10 EXISTS ON ARISTA TOR SWITCHES
        eos_vlan:
          vlanid=10
          name=WEB_VLAN
          host={{ inventory_hostname }}
          username={{ user }}
          password={{ pwd }}
        when: vendor == "arista"

  - name: PLAY 2 - Core (TOR) Switches
    hosts: core
    connection: local

    tasks:
      - name: ENSURE VLANS EXIST IN CORE
        nxos_vlan:
          vlan_id={{ item }}
          host={{ inventory_hostname }}
          username={{ user }}
          password={{ pwd }}
        with_items:
          - 10
          - 20
          - 30
          - 40
          - 50
```

Inventory File

Using an inventory file, such as the preceding one, enables us to automate tasks for specific hosts and groups of hosts by referencing the proper host/group using the `hosts` parameter that exists at the top section of each play.

It is also possible to store variables within an inventory file. This is shown in the example. If the variable is on the same line as a host, it is a host-specific variable. If the variables are defined within brackets such as `[all:vars]`, it means that the variables are in scope for the group `all`, which is a default group that includes *all* hosts in the inventory file.

NOTE

Inventory files are the quickest way to get started with Ansible, but should you already have a source of truth for network devices such as a network management tool or CMDB, it is possible to create and use a dynamic inventory script rather than a static inventory file.

Playbook

The playbook is the top-level object that is executed to automate network devices. In our example, this is the file *site.yml*, as depicted in the preceding example. A playbook uses YAML to define the set of tasks to automate, and each playbook is comprised of one or more plays. This is analogous to a football playbook. Like in football, teams have playbooks made up of plays, and Ansible playbooks are made up of plays, too.

NOTE

YAML is a data format that is supported by all programming languages. YAML is itself a superset of JSON, and it's quite easy to recognize YAML files, as they always start with three dashes (hyphens), `---`.

Play

One or more plays can exist within an Ansible playbook. In the preceding example, there are two plays within the playbook. Each starts with a *header* section where play-specific parameters are defined.

The two plays from that example have the following parameters defined:

`name`
: The text `PLAY 1 - Top of Rack (TOR) Switches` is arbitrary and is displayed when the playbook runs to improve readability during playbook execution and reporting. This is an optional parameter.

`hosts`
: As covered previously, this is the host or group of hosts that are automated in this particular play. This is a required parameter.

`connection`
: As covered previously, this is the type of connection mechanism used for the play. This is an optional parameter, but is commonly set to `local` for network automation plays.

Each play is comprised of one or more tasks.

Tasks

Tasks represent what is automated in a declarative manner without worrying about the underlying syntax or "how" the operation is performed.

In our example, the first play has two tasks. Each task ensures VLAN 10 exists. The first task does this for Cisco Nexus devices, and the second task does this for Arista devices:

```
tasks:
  - name: ENSURE VLAN 10 EXISTS ON CISCO TOR SWITCHES
    nxos_vlan:
      vlan_id=10
      name=WEB_VLAN
      host={{ inventory_hostname }}
      username=admin
      password=admin
    when: vendor == "nxos"
```

Tasks can also use the `name` parameter just like plays can. As with plays, the text is arbitrary and is displayed when the playbook runs to improve readability during playbook execution and reporting. It is an optional parameter for each task.

The next line in the example task starts with `nxos_vlan`. This tell us that this task will execute the Ansible module called `nxos_vlan`.

We'll now dig deeper into modules.

Modules

It is critical to understand modules within Ansible. While any programming language can be used to write Ansible modules as long as they return JSON key-value pairs, they are almost always written in Python. In our example, we see two modules being executed: `nxos_vlan` and `eos_vlan`. The modules are both Python files; and in fact, while you can't tell from looking at the playbook, the real filenames are *eos_vlan.py* and *nxos_vlan.py*, respectively.

Let's look at the first task in the first play from the preceding example:

```
- name: ENSURE VLAN 10 EXISTS ON CISCO TOR SWITCHES
  nxos_vlan:
    vlan_id=10
    name=WEB_VLAN
    host={{ inventory_hostname }}
    username=admin
    password=admin
  when: vendor == "nxos"
```

This task executes `nxos_vlan`, which is a module that automates VLAN configuration. In order to use modules, including this one, you need to specify the desired state or configuration policy you want the device to have. This example states: VLAN 10 should be configured with the name `WEB_VLAN`, and it should exist on each switch being automated. We can see this easily with the `vlan_id` and `name` parameters. There are three other parameters being passed into the module as well. They are `host`, `username`, and `password`:

`host`
: This is the hostname (or IP address) of the device being automated. Since the hosts we want to automate are already defined in the inventory file, we can use the built-in Ansible variable `inventory_hostname`. This variable is equal to what is in the inventory file. For example, on the first iteration, the host in the inventory file is `rack1-tor1`, and on the second iteration, it is `rack1-tor2`. These names are passed into the module and then within the module, a DNS lookup occurs on each name to resolve it to an IP address. Then the communication begins with the device.

`username`
: Username used to log in to the switch.

`password`
: Password used to log in to the switch.

The last piece to cover here is the use of the `when` statement. This is how Ansible performs conditional tasks within a play. As we know, there are multiple devices and types of devices that exist within the `tor` group for this play. Using `when` offers an option to be more selective based on any criteria. Here we are only automating Cisco devices because we are using the `nxos_vlan` module in this task, while in the next task, we are automating only the Arista devices because the `eos_vlan` module is used.

NOTE

This isn't the only way to differentiate between devices. This is being shown to illustrate the use of `when` and that variables can be defined within the inventory file.

Defining variables in an inventory file is great for getting started, but as you continue to use Ansible, you'll want to use YAML-based variables files to help with scale, versioning, and minimizing change to a given file. This will also simplify and improve readability for the inventory file and each variables file used. An example of a variables file was given earlier when the build/push method of device provisioning was covered.

Here are a few other points to understand about the tasks in the last example:

- Play 1 task 1 shows the `username` and `password` hardcoded as parameters being passed into the specific module (`nxos_vlan`).
- Play 1 task 1 and play 2 passed variables into the module instead of hardcoding them. This masks the `username` and `password` parameters, but it's worth noting that these variables are being pulled from the inventory file (for this example).
- Play 1 uses a *horizontal* key=value syntax for the parameters being passed into the modules, while play 2 uses the vertical key=value syntax. Both work just fine. You can also use vertical YAML syntax with "key: value" syntax.

- The last task also introduces how to use a *loop* within Ansible. This is by using `with_items` and is analogous to a for loop. That particular task is looping through five VLANs to ensure they all exist on the switch. Note: it's also possible to store these VLANs in an external YAML variables file as well. Also note that the alternative to not using `with_items` would be to have one task per VLAN—and that just wouldn't scale!

CHAPTER 7

Hands-on Look at Using Ansible for Network Automation

In the previous chapter, a general overview of Ansible terminology was provided. This covered many of the specific Ansible terms, such as playbooks, plays, tasks, modules, and inventory files. This section will continue to provide working examples of using Ansible for network automation, but will provide more detail on working with modules to automate a few different types of devices. Examples will include automating devices from multiple vendors, including Cisco, Arista, Cumulus, and Juniper.

The examples in this section assume the following:

- Ansible is installed.
- The proper APIs are enabled on the devices (NX-API, eAPI, NETCONF).
- Users exist with the proper permissions on the system to make changes via the API.
- All Ansible modules exist on the system and are in the library path.

NOTE

Setting the module and library path can be done within the *ansible.cfg* file. You can also use the `-M` flag from the command line to change it when executing a playbook.

The inventory used for the examples in this section is shown in the following section (with passwords removed and IP addresses changed). In this example, some hostnames are not FQDNs as they were in the previous examples.

Inventory File

```
[cumulus]
cvx  ansible_ssh_host=1.2.3.4 ansible_ssh_pass=PASSWORD

[arista]
veos1

[cisco]
nx1  hostip=5.6.7.8 un=USERNAME pwd=PASSWORD

[juniper]
vsrx hostip=9.10.11.12 un=USERNAME pwd=PASSWORD
```

NOTE Just in case you're wondering at this point, Ansible does support functionality that allows you to store passwords in encrypted files. If you want to learn more about this feature, check out Ansible Vault in the docs on the Ansible website.

This inventory file has four groups defined with a single host in each group. Let's review each section in a little more detail:

Cumulus
: The host `cvx` is a Cumulus Linux (CL) switch, and it is the only device in the `cumulus` group. Remember that CL is native Linux, so this means the default connection mechanism (SSH) is used to connect to and automate the CL switch. Because `cvx` is not defined in DNS or */etc/hosts*, we'll let Ansible know not to use the hostname defined in the inventory file, but rather the name/IP defined for `ansible_ssh_host`. The username to log in to the CL switch is defined in the playbook, but you can see that the password is being defined in the inventory file using the `ansible_ssh_pass` variable.

Arista
: The host called `veos1` is an Arista switch running EOS. It is the only host that exists within the `arista` group. As you can see for Arista, there are no other parameters defined within the inven-

tory file. This is because Arista uses a special configuration file for their devices. This file is called *.eapi.conf* and for our example, it is stored in the home directory. Here is the conf file being used for this example to function properly:

```
[connection:veos1]
host: 2.4.3.4
username: unadmin
password: pwadmin
```

This file contains all required information for Ansible (and the Arista Python library called *pyeapi*) to connect to the device using just the information as defined in the conf file.

Cisco

Just like with Cumulus and Arista, there is only one host (`nx1`) that exists within the `cisco` group. This is an NX-OS-based Cisco Nexus switch. Notice how there are three variables defined for `nx1`. They include `un` and `pwd`, which are accessed in the playbook and passed into the Cisco modules in order to connect to the device. In addition, there is a parameter called `hostip`. This is required because `nx1` is also not defined in DNS or configured in the */etc/hosts* file.

NOTE

We could have named this parameter anything. If automating a native Linux device, `ansible_ssh_host` is used just like we saw with the Cumulus example (if the name as defined in the inventory is not resolvable). In this example, we could have still used `ansible_ssh_host`, but it is not a requirement, since we'll be passing this variable as a parameter into Cisco modules, whereas `ansible_ssh_host` is automatically checked when using the default SSH connection mechanism.

Juniper

As with the previous three groups and hosts, there is a single host `vsrx` that is located within the `juniper` group. The setup within the inventory file is identical to that of Cisco's as both are used the same exact way within the playbook.

Playbook

The next playbook has four different plays. Each play is built to automate a specific group of devices based on vendor type. Note that this is only one way to perform these tasks within a single playbook. There are other ways in which we could have used conditionals (`when` statement) or created Ansible roles (which is not covered in this report).

Here is the example playbook:

```
---

  - name: PLAY 1 - CISCO NXOS
    hosts: cisco
    connection: local

    tasks:
      - name: ENSURE VLAN 100 exists on Cisco Nexus switches
        nxos_vlan:
          vlan_id=100
          name=web_vlan
          host={{ hostip }}
          username={{ un }}
          password={{ pwd }}

  - name: PLAY 2 - ARISTA EOS
    hosts: arista
    connection: local

    tasks:
      - name: ENSURE VLAN 100 exists on Arista switches
        eos_vlan:
          vlanid=100
          name=web_vlan
          connection={{ inventory_hostname }}

  - name: PLAY 3 - CUMULUS
    remote_user: cumulus
    sudo: true
    hosts: cumulus

    tasks:
      - name: ENSURE 100.10.10.1 is configured on swp1
        cl_interface: name=swp1  ipv4=100.10.10.1/24

      - name: restart networking without disruption
        shell: ifreload -a
```

```
- name: PLAY 4 - JUNIPER SRX changes
  hosts: juniper
  connection: local

  tasks:
  - name: INSTALL JUNOS CONFIG
    junos_install_config:
      host={{ hostip }}
      file=srx_demo.conf
      user={{ un }}
      passwd={{ pwd }}
      logfile=deploysite.log
      overwrite=yes
      diffs_file=junpr.diff
```

You will notice the first two plays are very similar to what we already covered in the original Cisco and Arista example. The only difference is that each group being automated (`cisco` and `arista`) is defined in its own play, and this is in contrast to using the `when` conditional that was used earlier.

There is no right way or wrong way to do this. It all depends on what information is known up front and what fits your environment and use cases best, but our intent is to show a few ways to do the same thing.

The third play automates the configuration of interface `swp1` that exists on the Cumulus Linux switch. The first task within this play ensures that `swp1` is a Layer 3 interface and is configured with the IP address 100.10.10.1. Because Cumulus Linux is native Linux, the networking service needs to be restarted for the changes to take effect. This could have also been done using Ansible handlers (out of the scope of this report). There is also an Ansible core module called `service` that could have been used, but that would disrupt networking on the switch; using `ifreload` restarts networking nondisruptively.

Up until now in this section, we looked at Ansible modules focused on specific tasks such as configuring interfaces and VLANs. The fourth play uses another option. We'll look at a module that *pushes* a full configuration file and immediately activates it as the new running configuration. This is what we showed previously using `napalm_install_config`, but this example uses a Juniper-specific module called `junos_install_config`.

This module `junos_install_config` accepts several parameters, as seen in the example. By now, you should understand what `user`, `passwd`, and `host` are used for. The other parameters are defined as follows:

`file`
: This is the config file that is copied from the Ansible control host to the Juniper device.

`logfile`
: This is optional, but if specified, it is used to store messages generated while executing the module.

`overwrite`
: When set to yes/true, the complete configuration is replaced with the file being sent (default is false).

`diffs_file`
: This is optional, but if specified, will store the diffs generated when applying the configuration. An example of the diff generated when just changing the hostname but still sending a complete config file is shown next:

```
# filename: junpr.diff
[edit system]
-  host-name vsrx;
+  host-name vsrx-demo;
```

That covers the detailed overview of the playbook. Let's take a look at what happens when the playbook is executed:

NOTE

Note: the `-i` flag is used to specify the inventory file to use. The `ANSIBLE_HOSTS` environment variable can also be set rather than using the flag each time a playbook is executed.

```
ntc@ntc:~/ansible/multivendor$ ansible-playbook -i inventory
demo.yml

PLAY       [PLAY      1       -       CISCO      NXOS]
*************************************************

TASK: [ENSURE VLAN 100 exists on Cisco Nexus switches]
*********************
changed: [nx1]
```

```
PLAY       [PLAY       2       -       ARISTA       EOS]
**************************************************

TASK:   [ENSURE   VLAN   100   exists   on   Arista   switches]
**************************
changed: [veos1]

PLAY        [PLAY        3        -        CUMULUS]
*****************************************************

GATHERING                         FACTS
*************************************************************
ok: [cvx]

TASK:   [ENSURE   100.10.10.1   is   configured   on   swp1]
***************************
changed: [cvx]

TASK:    [restart    networking    without    disruption]
******************************
changed: [cvx]

PLAY     [PLAY     4     -     JUNIPER     SRX     changes]
****************************************

TASK:          [INSTALL         JUNOS         CONFIG]
***********************************************
changed: [vsrx]

PLAY                               RECAP
***************************************************************
           to retry, use: --limit @/home/ansible/demo.retry

cvx                           : ok=3    changed=2    unreacha-
ble=0    failed=0
nx1                           : ok=1    changed=1    unreacha-
ble=0    failed=0
veos1                         : ok=1    changed=1    unreacha-
ble=0    failed=0
vsrx                          : ok=1    changed=1    unreacha-
ble=0    failed=0
```

You can see that each task completes successfully; and if you are on the terminal, you'll see that each changed task was displayed with an amber color.

Let's run this playbook again. By running it again, we can verify that all of the modules are *idempotent*; and when doing this, we see that NO changes are made to the devices and everything is green:

```
PLAY       [PLAY       1       -       CISCO       NXOS]
***************************************************

TASK:  [ENSURE  VLAN  100  exists  on  Cisco  Nexus  switches]
***********************
ok: [nx1]

PLAY       [PLAY       2       -       ARISTA       EOS]
***************************************************

TASK:   [ENSURE   VLAN   100   exists   on   Arista   switches]
****************************
ok: [veos1]

PLAY         [PLAY         3         -         CUMULUS]
******************************************************

GATHERING                         FACTS
**************************************************************
ok: [cvx]

TASK:   [ENSURE   100.10.10.1   is   configured   on   swp1]
*****************************
ok: [cvx]

TASK:     [restart     networking     without     disruption]
********************************
skipping: [cvx]

PLAY     [PLAY     4     -     JUNIPER     SRX     changes]
******************************************

TASK:          [INSTALL         JUNOS         CONFIG]
*************************************************
ok: [vsrx]

PLAY                               RECAP
***************************************************************
cvx                          : ok=2     changed=0     unreacha-
ble=0    failed=0
nx1                          : ok=1     changed=0     unreacha-
ble=0    failed=0
veos1                        : ok=1     changed=0     unreacha-
ble=0    failed=0
vsrx                         : ok=1     changed=0     unreacha-
ble=0    failed=0
```

Notice how there were 0 changes, but they still returned "ok" for each task. This verifies, as expected, that each of the modules in this playbook are idempotent.

CHAPTER 8
Summary

Ansible is a super-simple automation platform that is agentless and extensible. The network community continues to rally around Ansible as a platform that can be used for network automation tasks that range from configuration management to data collection and reporting. You can push full configuration files with Ansible, configure specific network resources with idempotent modules such as interfaces or VLANs, or simply just automate the collection of information such as neighbors, serial numbers, uptime, and interface stats, and customize reports as you need them.

Because of its architecture, Ansible proves to be a great tool available here and now that helps bridge the gap from *legacy CLI/SNMP* network device automation to modern *API-driven* automation.

Ansible's ease of use and agentless architecture accounts for the platform's increasing following within the networking community. Again, this makes it possible to automate devices without APIs (CLI/SNMP); devices that have modern APIs, including standalone switches, routers, and Layer 4-7 service appliances; and even those software-defined networking (SDN) controllers that offer RESTful APIs.

There is no device left behind when using Ansible for network automation.

About the Author

Jason Edelman, CCIE 15394 & VCDX-NV 167, is a born and bred network engineer from the great state of New Jersey. He was the typical "lover of the CLI" or "router jockey." At some point several years ago, he made the decision to focus more on software development practices and how they are converging with network engineering. Jason currently runs a boutique consulting firm, Network to Code, helping vendors and end users take advantage of new tools and technologies to reduce their operational inefficiencies. Jason has a Bachelor of Engineering from Stevens Institute of Technology in NJ and still resides locally in the New York City Metro Area. Jason also writes regularly on his personal blog, *jedelman.com*, and can be found on Twitter at *@jedelman8*.