



Red Hat Training and Certification

Student Workbook (ROLE)

Red Hat Ansible Engine 2.5 DO457

Red Hat Ansible for Network Automation

Edition 2

Red Hat Ansible for Network Automation



Red Hat Ansible Engine 2.5 DO457
Red Hat Ansible for Network Automation
Edition 2 20200820
Publication date 20200820

Author: Steven Laesch
Editor: Steve Bonneville, Sean Cavanaugh

Copyright © 2018 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2018 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send email to training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, Hibernate, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a registered trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

The OpenStack® word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation in the United States and other countries, and are used with the OpenStack Foundation's permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the OpenStack community.

All other trademarks are the property of their respective owners.

Document Conventions	ix
Introduction	xi
Red Hat Ansible for Network Automation	xi
Orientation to the Classroom Environment	xii
1. Deploying Ansible	1
Preparing to Install Ansible	2
Orientation to Lab Activities	6
Installing Ansible	16
Guided Exercise: Installing Ansible on the Control Node	18
Guided Exercise: Identifying Resources for Installed Plug-ins	20
Defining Ansible's Scope	24
Creating Ansible Inventories	25
Guided Exercise: Creating Host Inventories	28
Configuring Ansible	32
Guided Exercise: Configuring Ansible	39
Lab: Deploying Ansible	46
2. Running Commands and Plays	51
Executing Ad Hoc Commands	52
Guided Exercise: Executing Ad Hoc Commands	54
Preparing Ansible Playbooks	56
Guided Exercise: Converting an Ad Hoc Command to a Play	60
Building a Play with Multiple Tasks	62
Guided Exercise: Building a Play with Multiple Tasks	67
Composing Playbooks with Multiple Plays	70
Guided Exercise: Composing Playbooks with Multiple Plays	74
Lab: Running Commands and Plays	76
3. Parameterizing Automation	81
Defining Variables	82
Guided Exercise: Defining and Using Variables	87
Controlling Tasks with Loops and Conditions	93
Guided Exercise: Controlling Tasks with Loops and Conditions	101
Transforming Variable Data with Filters	104
Guided Exercise: Looping Over a Filtered List	106
Working with Roles	108
Guided Exercise: Creating and Using Roles	115
Customizing Data with Jinja2 Templates	120
Guided Exercise: Generating Config Statements with Jinja2	124
Lab: Parameterizing Automation	128
4. Administering Ansible	139
Ansible in the Enterprise	140
Safeguarding Sensitive Data with Ansible Vault	143
Guided Exercise: Safeguarding Sensitive Data with Ansible Vault	146
Running Plays with Encrypted Data	151
Guided Exercise: Running Plays with Encrypted Data	152
Protecting Resources with Ansible Vault	155
Guided Exercise: Protecting Resources with Ansible Vault	157
Creating Inventories Using YAML	162
Guided Exercise: Creating Inventories Using YAML	164
Generating and Using Dynamic Inventories	167
Guided Exercise: Generating and Using Dynamic Inventories	171
Centrally Running Ansible with Red Hat Ansible Tower	174
Guided Exercise: Navigating the Red Hat Ansible Tower Web Interface	180

Guided Exercise: Creating Inventories in Red Hat Ansible Tower	183
Lab: Administering Ansible	186
5. Automating Simple Network Operations	189
Gathering Network Information with Ansible	190
Guided Exercise: Gathering Facts	194
Guided Exercise: Viewing System Settings	199
Configuring Network Devices	203
Guided Exercise: Backing Up Network Device Configurations	205
Configuring the Host Name	207
Guided Exercise: Configuring the Host Name	208
Configuring System Settings	211
Guided Exercise: Configuring System Settings	212
Generating Configuration Settings from Jinja2 Templates	218
Guided Exercise: Configuring From Templates	220
Guided Exercise: Configuring Settings From Templates	223
Enabling and Disabling Interfaces	226
Guided Exercise: Bouncing an Interface	227
Guided Exercise: Bouncing Specified IOS Interfaces	232
Reinitializing Layer 3 Interfaces	234
Guided Exercise: Reinitializing Layer 3 on VyOS Devices	236
Provisioning the Start-up Network	239
Guided Exercise: Provisioning the Start-up Network	240
Provisioning Spine and Leaf Devices	244
Guided Exercise: Provisioning Spine and Leaf Devices	245
Setting Parameters with Ansible Tower Surveys	251
Guided Exercise: Setting Parameters with Ansible Tower Surveys	252
Lab: Automating Simple Network Operations	254
6. Automating Complex Network Operations	261
Aggregating Logged Events to Syslog	262
Guided Exercise: Aggregating Logged Events to Syslog	263
Managing Access Control Lists on IOS	266
Guided Exercise: Managing Access Control Lists on IOS	267
Enabling SNMP	269
Guided Exercise: Enabling SNMP	270
Overcoming Real-world Challenges	273
Guided Exercise: Provisioning the Consolidation Network	276
Implementing Dynamic Routing with OSPF	281
Guided Exercise: Implementing Dynamic Routing with OSPF	283
Guided Exercise: Verifying End-to-End Reachability	288
Implementing OSPF with Multiple Autonomous Systems	291
Guided Exercise: Provisioning the Break Up Network	292
Implementing Dynamic Routing with EBGP	298
Guided Exercise: Implementing Dynamic Routing with EBGP	300
Upgrading the Network	306
Guided Exercise: Upgrading VyOS	308
Lab: Automating Complex Operations	311
7. Comprehensive Review	315
Comprehensive Review	316
Lab: Deploying Ansible	318
Lab: Executing Commands and Plays	324
Lab: Parameterizing Automation	329
Lab: Administering Automation	336
Lab: Automating Simple Network Operations	342

Lab: Automating Complex Network Operations	350
A. Table of Lab Network Hosts and Groups	361
.....	362
B. Connection and Authentication Variables	365
.....	366
C. Editing Files with Vim	367
Guided Exercise: Editing Files with Vim	368
D. IOS Minimal Management Configuration	371
Restoring IOS Configuration Settings	372
E. Layer 3 Addresses for Lab Network	373
.....	374
F. Ansible Variable Types	375
Ansible Variable Types and Precedence	376
G. Changing the Screen Resolution	381
Changing the Screen Resolution	382

Document Conventions



References

"References" describe where to find external documentation relevant to a subject.



Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.

Introduction

Red Hat Ansible for Network Automation

Red Hat Ansible Engine and Red Hat Ansible Tower power the administration of today's top enterprise infrastructures.

After this course, you will be able to implement and integrate a broad range of automation capabilities using this state-of-the-art platform.

Course Objectives	Automate the configuration and management of your network infrastructure using Red Hat Ansible Automation for Networking.
Audience	Network administrators, network automation engineers, and infrastructure automation engineers who want to learn how to use Ansible to automate the administration, deployment, and configuration management of the network infrastructure of their organization or enterprise.
Prerequisites	<ul style="list-style-type: none">• Experience with network administration, including a solid understanding of TCP/IP, routers, and managed switches. Students should be familiar with managing network devices from the command line, preferably with one or more of Cisco IOS, IOS XR, or NX-OS; Juniper JUNOS; Arista EOS; or VyOS.• The students will work with text files and run commands in a Red Hat Enterprise Linux environment. A working knowledge of Linux, including how to edit text files and run commands from the shell, and how to use SSH to log in to remote systems. Knowledge equivalent to "Red Hat System Administration I" (RH124) or better is recommended.• Prior Ansible knowledge is not required.

Orientation to the Classroom Environment

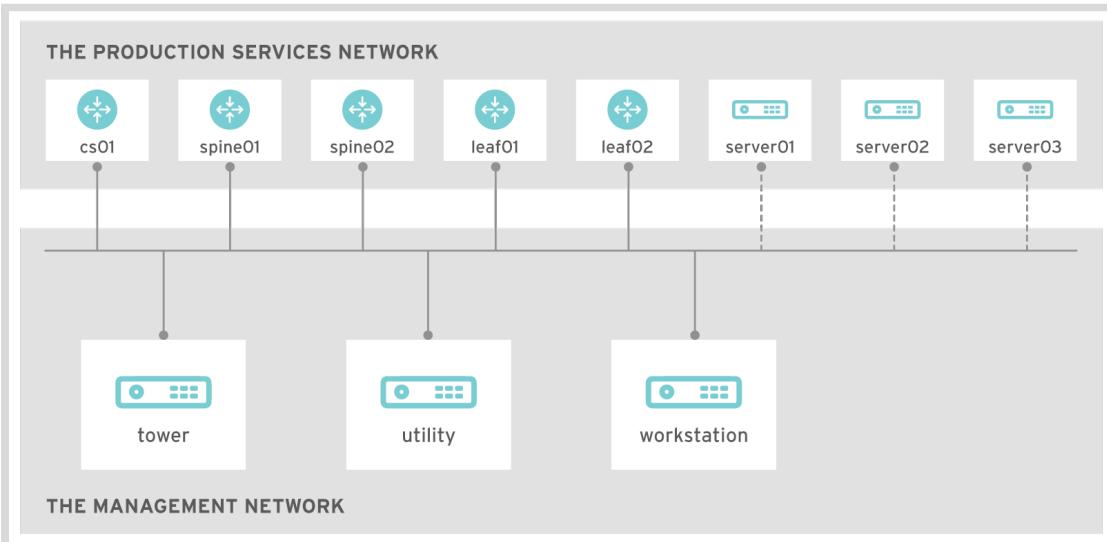


Figure 0.1: Classroom environment (management network, 172.25.250.0/24)

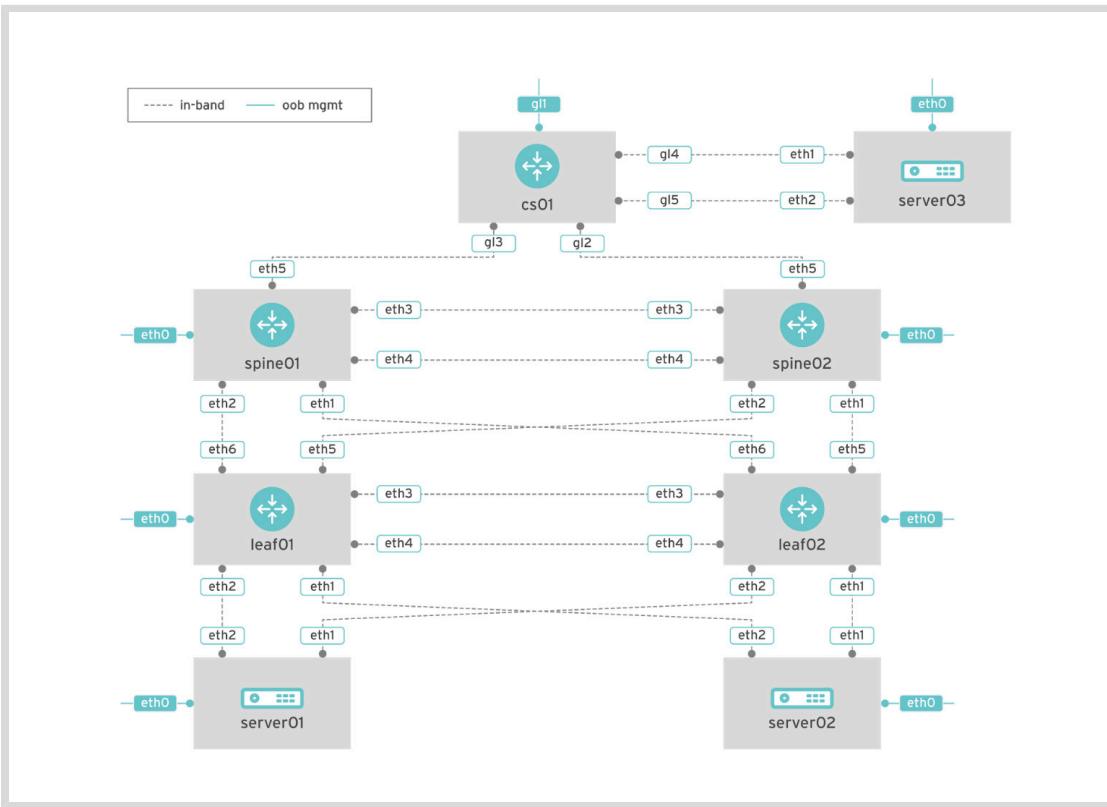


Figure 0.2: Classroom environment (lab network, Layer 2 interconnections). Interfaces labeled "oob mgmt" are connected to the management network, 172.25.250.0/24.

Introduction

In this course, you will learn how to use Red Hat Ansible Automation Platform to automate your management of network devices. A number of virtual network devices have been provided for you, running two network operating systems: Cisco IOS and VyOS.

A *management network* that provides out-of-band management access to all network devices and servers in the classroom environment has been provided for you. That network assigns hosts IPv4 addresses from **172.25.250.0/24**.

A *lab network* (or *production services network*) connecting your network devices and three servers represents the "production" network that you are using Ansible to configure. The layer 2 diagram of the lab network is illustrated in the second of the two preceding diagrams.

Your network devices and production servers are as follows:

Network Devices and Production Servers

Machine name	OS	Role
cs01.lab.example.com	Cisco IOS-XE 16.3.5	Cisco CSR1000V router representing a cloud-hosted device.
spine01.lab.example.com	VyOS 1.1.8	VyOS router representing a spine or top-level router on-site.
spine02.lab.example.com	VyOS 1.1.8	VyOS router representing a spine or top-level router on-site.
leaf01.lab.example.com	VyOS 1.1.8	VyOS router representing a leaf router or access-level L3 switch on-site.
leaf02.lab.example.com	VyOS 1.1.8	VyOS router representing a leaf router or access-level L3 switch on-site.
server01.lab.example.com	RHEL 7.5	Represents an on-site Linux server.
server02.lab.example.com	RHEL 7.5	Represents an on-site Linux server.
server03.lab.example.com	RHEL 7.5	Represents a cloud-hosted Linux server.

For administrative access, the **admin** account on the Cisco router has the password **student**, the **vyos** accounts on the VyOS routers have the password **vyos**, and the **root** accounts on the RHEL servers have the password **redhat**.

You have also been provided with three additional servers on the management network:

Management Systems

Machine name	IP address	Role
workstation.lab.example.com	172.25.250.254	Graphical Red Hat Enterprise Linux 7 workstation for network administration.
utility.lab.example.com (git.lab.example.com)	172.25.250.8	Server for Git repositories that contain your Ansible automation code.
tower.lab.example.com	172.25.250.9	Red Hat Ansible Tower server.

You will start most hands-on activities on **workstation**, which you can use to edit files and run a web browser.

All Linux-based computer systems have a standard user account, **student**, which has the password **student**. The **root** password on all Linux servers is **redhat**.

The URL of your Git repository server is <http://git.lab.example.com:3000/student>.

The DNS domain name used in the classroom environment is **example.com**. Your machines are assigned names in the **lab** subdomain (**lab.example.com**).

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning classroom. They are accessed through a web application hosted at rol.redhat.com [<http://rol.redhat.com>]. You should log in to this site using your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through a web page. The state of each virtual machine in the classroom is displayed on the page under the **Online Lab** tab.

Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots into the same state as when it was shut down (the disk will have been preserved).
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

Depending on the state of a machine, a selection of the following actions is available.

Classroom/Machine Actions

Button or Action	Description
PROVISION LAB	Create the ROL classroom. Creates all of the virtual machines needed for the classroom and starts them. Can take several minutes to complete.
DELETE LAB	Delete the ROL classroom. Destroys all virtual machines in the classroom. Caution: Any work generated on the disks is lost.
START LAB	Start all virtual machines in the classroom.
SHUTDOWN LAB	Stop all virtual machines in the classroom.
OPEN CONSOLE	Open a new tab in the browser and connect to the console of the virtual machine. Students can log in directly to the virtual machine and run commands. In most cases, students should log in to the workstation virtual machine and use ssh to connect to the other virtual machines.
ACTION → Start	Start (power on) the virtual machine.
ACTION → Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION → Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION → Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. Caution: Any work generated on the disk is lost.

At the start of an exercise, if instructed to reset a single virtual machine node, click **ACTION → Reset** for only the specific virtual machine.

At the start of an exercise, if instructed to reset all virtual machines, click **ACTION → Reset**

If you want to return the classroom environment to its original state at the start of the course, you can click **DELETE LAB** to remove the entire classroom environment. After the lab has been deleted, you can click **PROVISION LAB** to provision a new set of classroom systems.



Warning

The **DELETE LAB** operation cannot be undone. Any work you have completed in the classroom environment up to that point will be lost.

The Autostop Timer

The Red Hat Online Learning enrollment entitles students to a certain amount of computer time. To help conserve allotted computer time, the ROL classroom has an associated countdown timer, which shuts down the classroom environment when the timer expires.

To adjust the timer, click **MODIFY** to display the **New Autostop Time** dialog box. Set the number of hours and minutes until the classroom should automatically stop. Note that there is a maximum time of ten hours. Click **ADJUST TIME** to apply this change to the timer settings.

Chapter 1

Deploying Ansible

Goal

Install and configure Ansible.

Objectives

- Install Red Hat Ansible Engine on a Red Hat Enterprise Linux control node.
- Set up and validate the hosts inventory used in the labs.

Sections

- Preparing to Install Ansible
- Orientation to Lab Activities
- Installing Ansible (and Guided Exercises)
- Define Ansible's Scope
- Creating Ansible Inventories (and Guided Exercise)
- Configuring Ansible (and Guided Exercise)

Lab

Deploying Ansible

Preparing to Install Ansible

Objectives

After completing this section, you should be able to:

- Define key technical terms: network automation, control node, managed nodes, and so on.
- Explain what Ansible and Red Hat Ansible Tower are and what their role is in network automation.

Key Terms

Automation

A process or procedure performed with minimal or reduced human intervention.

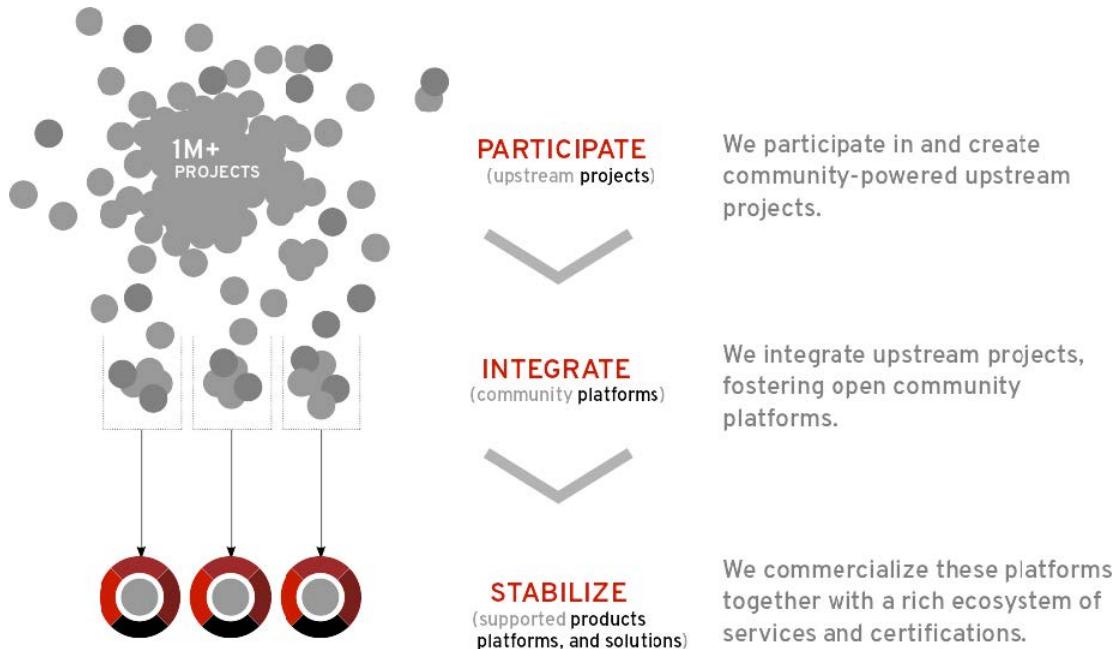
Network

A group or system of interconnected people or things.

Network Automation

Configuration, management, and operation of a group or system of interconnected people or things performed with minimal or reduced human intervention.

From Community to the Enterprise



Ansible and Red Hat Ansible Engine

An open source automation platform:

- Ansible (or Ansible Core) is the community-managed upstream project for the execution engine

Chapter1 | Deploying Ansible

- Red Hat Ansible Engine is Red Hat's commercially supported offering for Ansible <https://access.redhat.com/ansible-top-support-policies>

Red Hat Ansible Tower and AWX

Centralized management, logging, and control of Ansible at scale:

- Red Hat Ansible Tower is Red Hat's commercially supported management platform, which helps larger teams and deployments manage Ansible
 - Browser-based visual dashboard and API
 - Role-based access control and secure protection of credentials and other secrets
 - Central logging of jobs run and their results
 - Support for standard job templates and scheduled job execution
 - Many other features
- AWX is the community-based project on which Ansible Tower is based

Red Hat Ansible Automation for Networking

A support offering that includes Ansible Engine, Ansible Tower, and network module support.

Ansible Network Modules

- Developed, maintained, supported, and tested by Red Hat.
 - 140+ supported modules and growing.*
 - Red Hat supports and fixes problems.
 - The networking modules are included with Red Hat Ansible Engine, but this provides full support from Red Hat.
- * - Take note of supported platforms

Currently Included Systems

- Arista EOS
- Cisco IOS
- Cisco IOS XR
- Cisco NX-OS
- Juniper Junos
- Open vSwitch
- VyOS

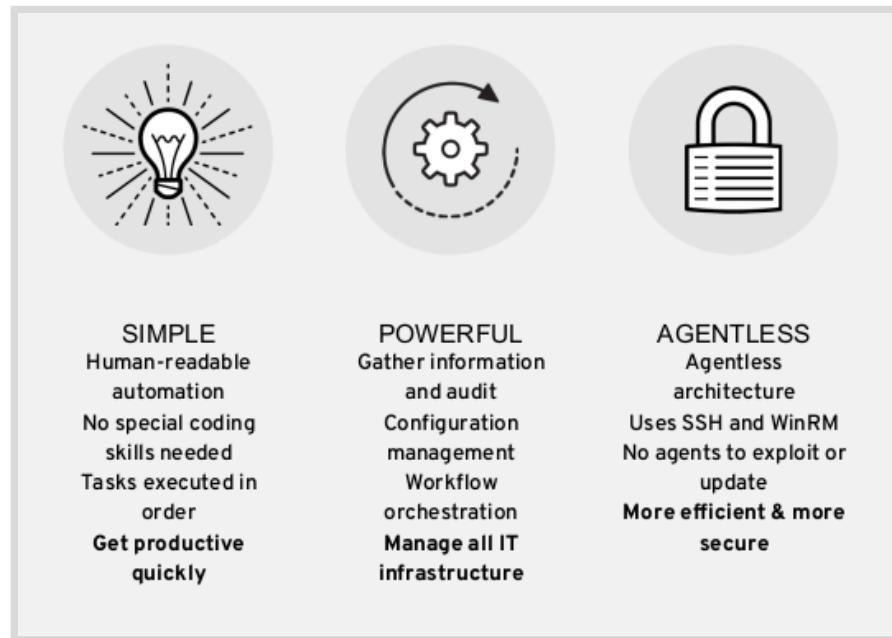
What is Ansible?

A very popular open source automation platform.

When the goal is to make the network more secure, and at the same time more agile and adaptable, the result is Ansible: a Python-based, command-line engine that interprets and

executes human readable, text-based playbooks in YAML format. Playbooks contain one or more plays that perform tasks in sequence.

Ansible Is...



Types of Nodes

Ansible supports two different types of nodes.

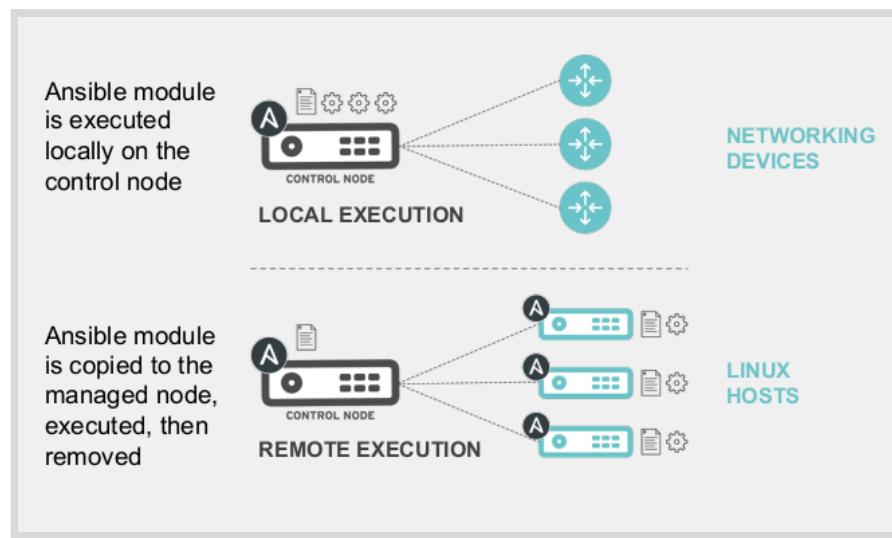
Control Node

Where Ansible ad hoc commands and plays run from.

Managed Nodes (Hosts)

Nodes that can be managed by Ansible. This includes the control node (as `localhost`).

How Does Network Automation Work?





References

The Ansible Project Documentation - All Things Ansible

<http://docs.ansible.com>

Official Sets of Installation Procedures

http://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html

Control Node Requirements

http://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html?#control-machine-requirements

Managed Node Requirements

http://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html?#managed-node-requirements

An Introduction to Network Automation With Ansible

http://docs.ansible.com/ansible/latest/network/getting_started/index.html

Ansible for Network Automation

<http://docs.ansible.com/ansible/latest/network/index.html>

Ansible Modules

http://docs.ansible.com/ansible/latest/modules/modules_by_category.html

Orientation to Lab Activities

Objectives

After completing this section, you should be able to:

- Describe the basic classroom lab environment.
- Outline the story being told in the hands-on activity scenarios throughout this course.

About the Story

The hands-on activities in this course are written to tell the story of an organization with an evolving network, **example.com**.

The guided exercises and lab activities in the first part of the course are written as if the original network administrator for the organization has prepared training for you to learn how to use Ansible to manage the company's network. Later exercises are intended to represent how you could use Ansible to perform increasingly complex network administration tasks and to migrate that network to more sophisticated configurations.

The remainder of this section provides a detailed overview of the classroom environment that you will use to perform these hands-on activities, and of the story told in those activities.

Introducing example.com

Like most businesses, **example.com** has had good years, and less good ones.

- The original founders of **example.com** were Joe and Denise. Very early on, they asked a friend named Jasper to join **example.com** and manage the network. Marsha manages production application servers.
- The growth and development of **example.com** is described in terms of five stages: start-up, expansion, consolidation, break-up, and adjustment. These will be described in more detail as the story of **example.com** unfolds.
- At each stage, the Production Services Network has been redesigned in order to better reflect the needs of the company at that time.

Designing Infrastructure Traffic

Jasper expects that in the future, **example.com** might want to change the way the Production Services Network is provisioned at layer 3.

- That can be relatively easy to do when your management connection to devices happens out of band.
- It's more challenging when your own packets are affected by changes.

The separation of the network infrastructure into production services and management is an example of in-band/out-of-band design. Production network traffic happens in the bandwidth of the Production Services Network, and management traffic happens out of the bandwidth associated with it.

Implementing a Management Network

In one of his first actions at **example.com**, Jasper created a management network. You should assume the management network exists, and does not change.

- The layer 3 address of the management network is **172.25.250.0/24**.
- On the management network are these systems:
 - A server running Red Hat Ansible Tower, which provides an optional web-based user interface and central management system for Ansible (**tower.lab.example.com**).
 - A utility server running Gitea, an open source Git service (**utility.lab.example.com**, also known as **git.lab.example.com**).
 - A management workstation running Red Hat Enterprise Linux (**workstation.lab.example.com**).

Accommodating Servers

At **example.com**, Marsha is in charge of servers.

- Interfaces exist on servers that are reserved for management network use, but these interfaces are currently turned off.
- In the future, **example.com** plans to have direct access all the way to servers by way of the management network.

At the present time, assume that servers can only be reached by way of the in-band network. In order for servers to be reachable, the Production Services Network (the lab network) must be working.

Management Machines

All student systems in the Management Network have a standard user account, **student**, which has the password **student**. The **root** password on all student systems is **redhat**.

Classroom Machines in the Management Network

Machine name	IP address	Role
workstation.lab.example.com	172.25.250.254	Graphical workstation used for administration.
utility.lab.example.com	172.25.250.8	Host used for Git repository
tower.lab.example.com	172.25.250.9	Host used for Ansible Tower server.

The URL of the student's Git repository on the Management Network is <http://git.lab.example.com:3000/student>.

Visualizing example.com Networks

Here is the schematic representation of the relationship between the Production Services and Management Networks at **example.com**.

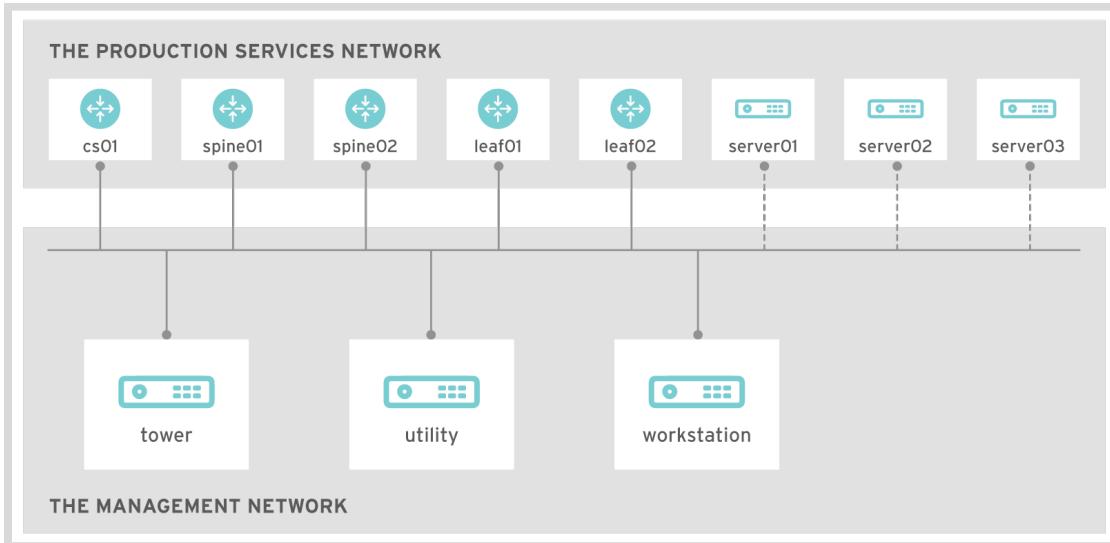


Figure 1.4: The Management Network topology

The topology of the Production Services Network connections are not shown in the previous diagram.

Identify Network Architecture

Jasper has been tasked by founders Joe and Denise to design, implement, and manage a production network that will grow with **example.com**.

He is familiar with the conventional three-tier network architecture: access layer, aggregation/distribution layer, and core. Lately, he has been reading accounts of the new leaf and spine architecture, and he is interested in exploring that at **example.com**.

Redesigning the Network

During their consolidation phase, **example.com** will have an opportunity to implement a data center with four network devices. The data center part of the Consolidation scenario is an example of true spine and leaf architecture, in the sense that every low-tier device (leaf layer) is connected to each of the top-tier devices (spine layer) in a full-mesh topology.

Other internetwork scenarios for **example.com** model changes that reflect shifting business needs. The leaf and spine naming convention is retained for devices, but other than the Consolidation scenario, roles and interconnections do not necessarily provide a good example of spine and leaf architecture. The Break Up scenario, for instance, resembles a hub and spoke topology.

Inspecting the Apparatus

In real life, outside the **example.com** narrative, a particular set of layer 2 interconnections exist. This underlying apparatus makes it possible to implement various forms of the **example.com** Production Services Network at layer 3:

- We must have access to sufficiently many virtual machines of appropriate types (VMs).
- To support the possibility of modeling many different scenarios in this course and in the future, more interconnections at layer 2 are preferred: a mesh is best.

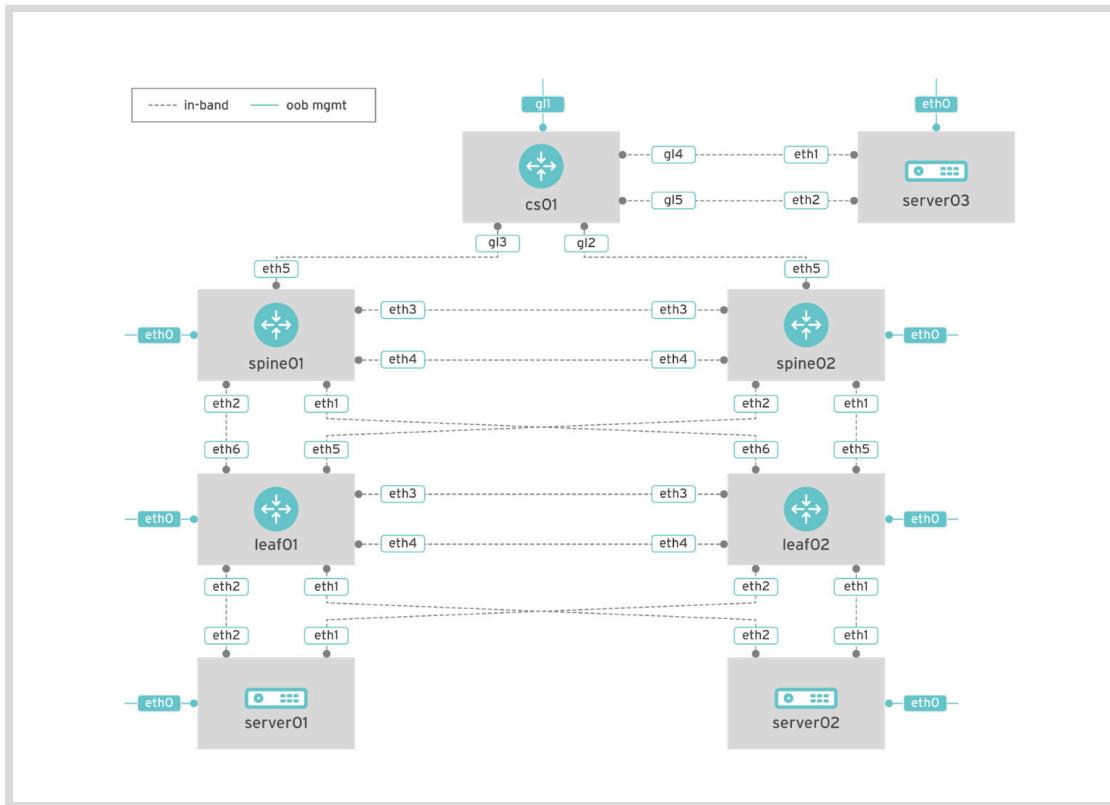


Figure 1.5: Production Services Network interconnections at layer 2

Interfaces labeled “oob mgmt” in the previous diagram are connected to the management network.

Adapting to Changing Requirements

As **example.com** goes through different stages or phases of development, the Production Services Network is redesigned to keep pace with changing business requirements.

Start-up

cloud-based, one server and a cloud services router.

Expansion

two separate locations (corporate plus satellite/branch office), in addition to cloud services.

Consolidation

a data center with a two-tier network; redundant devices at spine and leaf layers, plus connections to the cloud services router.

Break-up

a single corporate entity has been broken up into four different entities; three as distinct autonomous systems, plus a managed services provider.

Adjustment

an adjustment is made that reduces brittleness inherent in the original break-up design.

Start-up

The Start-up phase network of **example.com**.

Chapter 1 | Deploying Ansible

This single-router network models a simple external, internal separation between public and private networks. During this phase both router and server were hosted in the cloud. The diagram illustrates that physical location is immaterial; the same diagram could apply to a residence.

Notice that the interface **Gi1** on **cs01** is playing the role of an in-band internet connection, even though in the classroom environment it is actually connection to the management network at layer 2. During the start-up phase, Jasper was accessing the **example.com** organization's sole router for management purposes by way of the internet. (For the sake of illustration, what is actually the real-life management network is considered to be playing the role of the **example.com** organization's internet.)

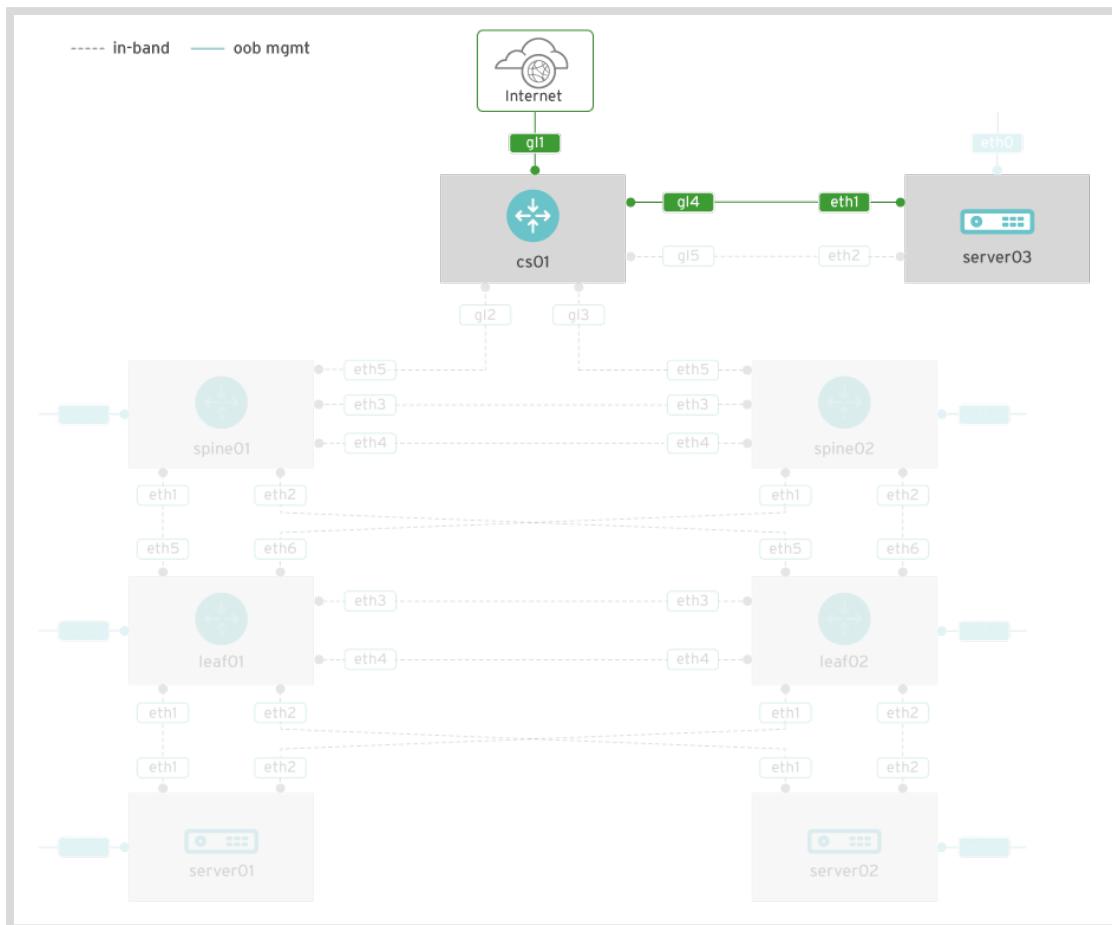


Figure 1.6: Start-up phase

Expansion

The Expansion phase network of **example.com**.

This models a scenario in which **example.com** has moved into an office. The **spine01** router is at the office. The network device **leaf01** is positioned where one expects a switch. It could be playing the role of a layer 3 switch, but with just the single server it fails to model fan-in/fan-out or port density.

The **cs01** router is representing a device that is still hosted with a cloud service provider.

Pretend that the **eth5** interface on **spine01** and the **Gi2** interface on **cs01** are connections to the networks of ISPs, even though in the actual lab environment they are directly connected to each other.

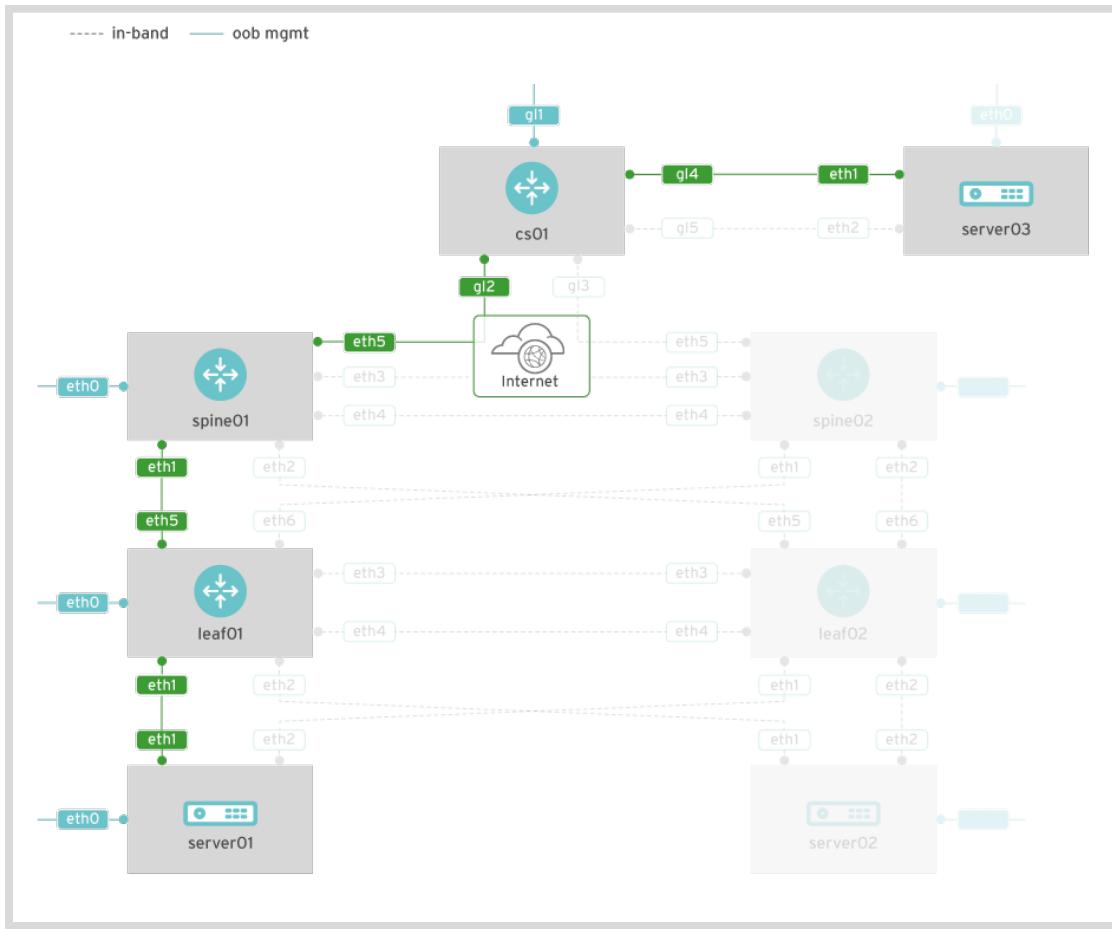


Figure 1.7: Expansion phase

Consolidation

The Consolidation phase network of **example.com**.

In this scenario there are two locations: a data center and the original router and server hosted in the cloud.

The data center portion of this scenario is an example of true spine and leaf architecture, in the sense that every lower-tier device (leaf layer) is connected to each of the top-tier devices (spine layer) in a full-mesh topology.

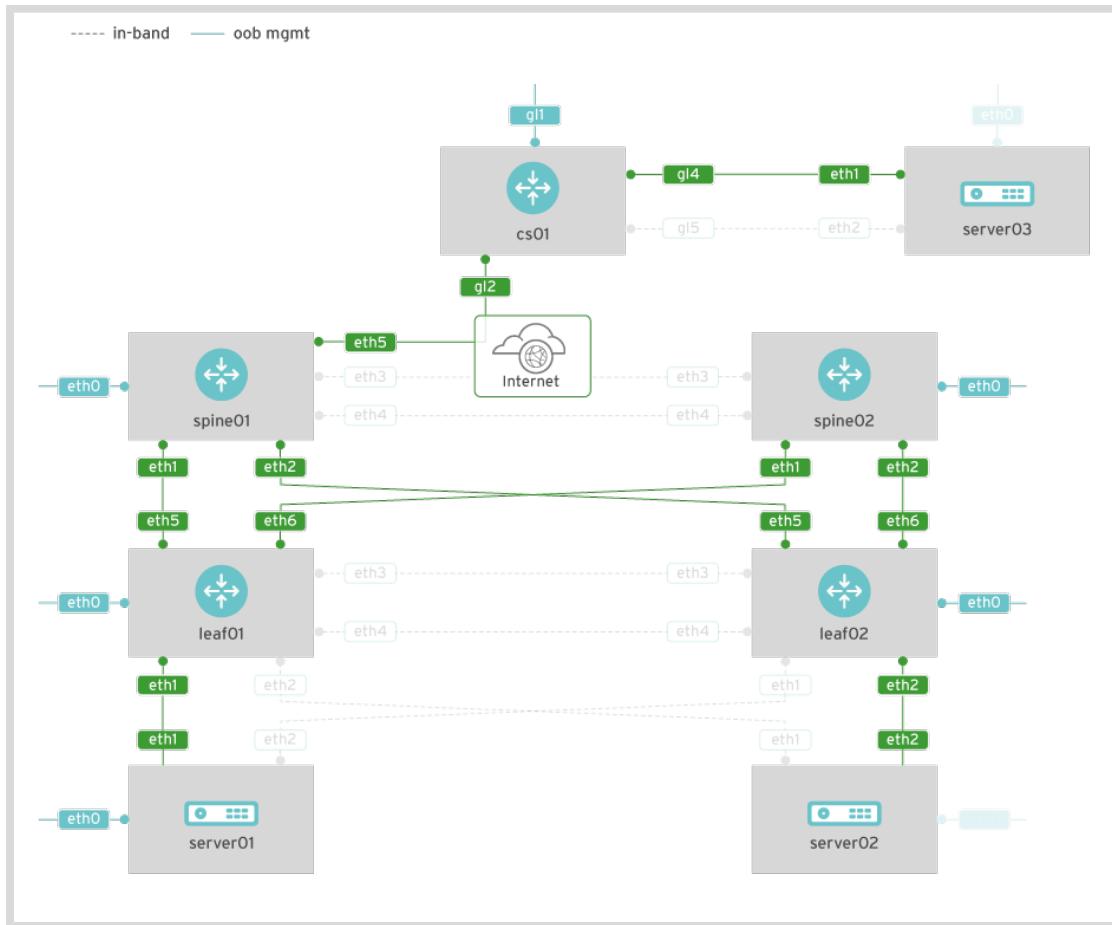


Figure 1.8: Consolidation phase

Break-up

The Break-up phase network of **example.com**.

This models the transition away from a single organization, into three separate organizations. A fourth organization is providing managed services.

The three organizations have separate governance and separate networks. Each is considered an autonomous system, even though one service provider manages the networks for all three organizations.

Even though they are now three separate organizations with their own independent networks, they have entered into an agreement to share routing information, which is communicated across AS boundaries by eBGP peers.

For this scenario, pretend that the **eth5** interface of **spine02** and the **Gi3** interface of **cs01** are now connected through the networks of ISPs, even though in the actual lab environment they are directly connected to each other. (This is just like the relationship between **eth5** on **spine01** and **Gi2** on **cs01** in the scenario and in the actual lab environment.)

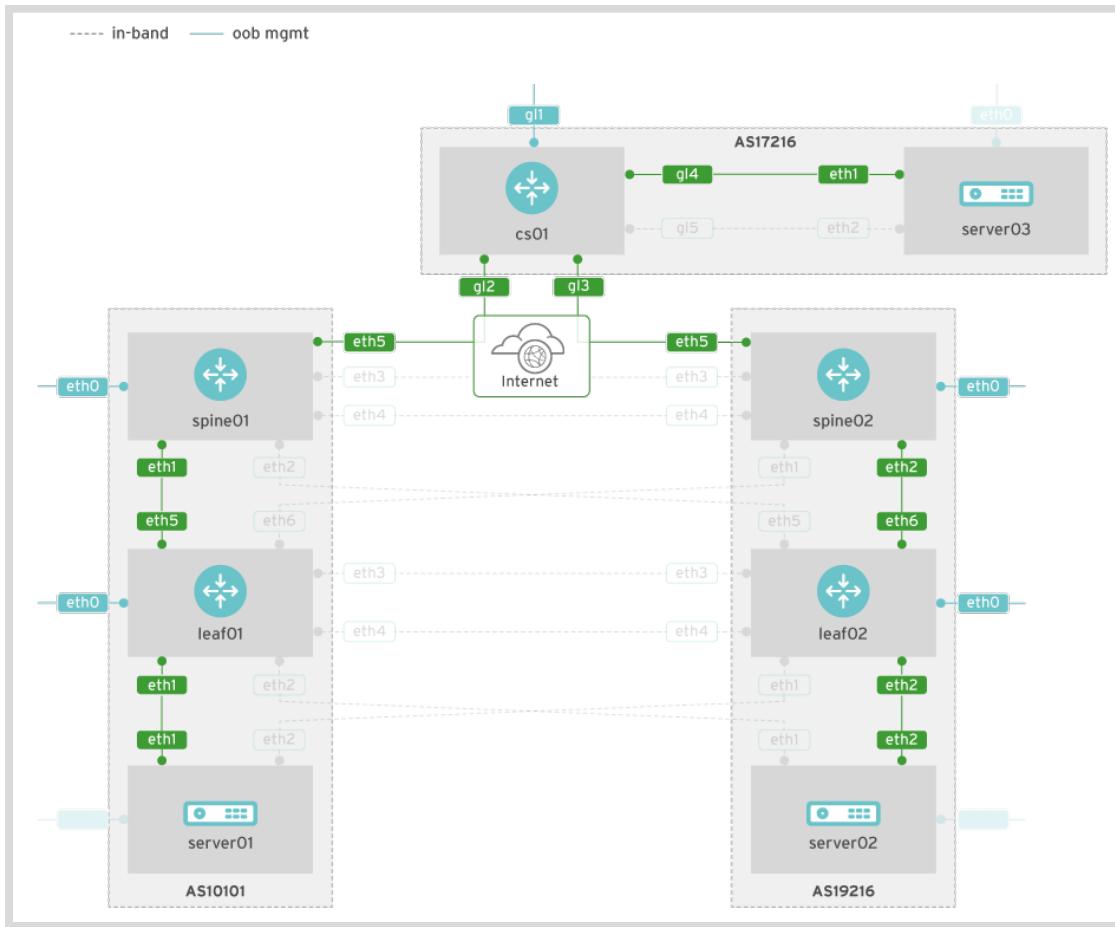


Figure 1.9: Break-up phase

Adjustment

The Adjustment phase network of example.com.

This model eliminates an inherent brittleness in the original Break-up scenario.

With the original Break-up phase internetwork, all traffic between AS10101 and AS19216 was routed through **cs01**. Not only did this add an extra hop, but **cs01** is potentially a single point of failure. Provisioning a direct link between AS10101 and AS19216 improves stability and performance.

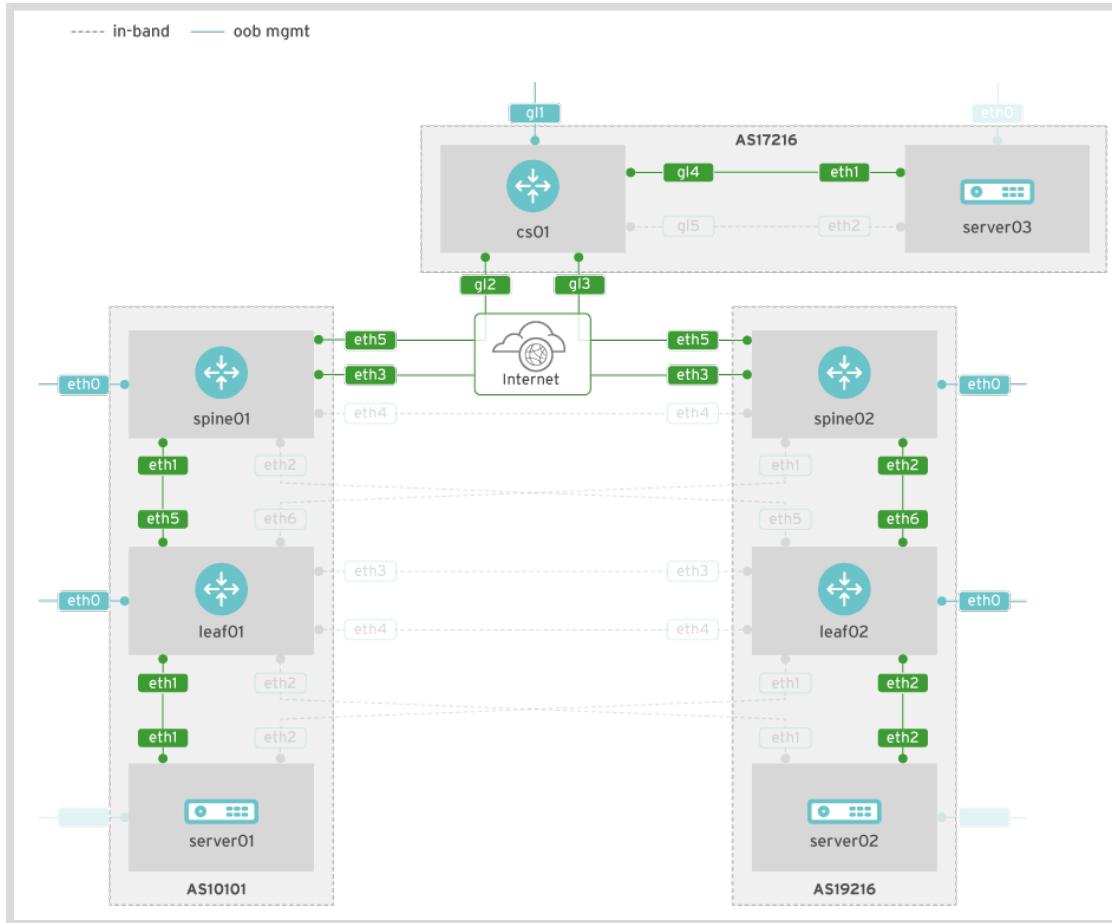


Figure 1.10: Adjustment phase

Operating Power Tools Safely

Jasper is ultimately responsible for managing networks at **example.com**.

- He knows mastery of tools empowers us to achieve much more, in a shorter period of time, than we could otherwise.
- He wants those who are granted administrative level privileges to use their tools skillfully and well.

At **example.com**, the power tools are Red Hat Ansible Engine and Red Hat Ansible Tower.

Completing the Hands-on Activities

Jasper devised a series of lessons illustrating features and functionality of Red Hat Ansible Engine and Red Hat Ansible Tower.

That series of lessons is organized in four parts, or chapters:

- **Deploying Ansible**
- **Running Commands and Plays**
- **Parameterizing Automation**
- **Administering Ansible**

Those who master these four challenges graduate to a fresh set of challenges that focus on networks and networking. Participants will help **example.com** shape and reshape its Production Services Network in response to changing business requirements.

Installing Ansible

Objectives

After completing this section, you should be able to:

- Install *Red Hat Ansible Engine*.
- List the programs installed with *Ansible*.

Sample Installation Procedure

This is an example of how to install Red Hat Ansible Engine on a Red Hat Enterprise Linux control node.

```
[root@host ~]# subscription-manager repos --enable=rhel-7-server-ansible-2-rpms  
[root@host ~]# yum install ansible
```

- Any Red Hat Enterprise Linux subscription can use this to install Ansible, with limited support scope.
- If you have an official Red Hat Ansible Engine support subscription, use **subscription-manager** to attach the control node to the pool containing it. See <https://access.redhat.com/articles/3174981>

Learning About Installed Plug-ins

Plug-ins are pieces of code that augment the core functionality of Ansible.

The **ansible-doc** tool displays information about installed Ansible plug-ins.

```
[user@host ~]$ ansible-doc [-l|-s] [options] [-t <plugin type>] [plugin]
```

Use the list option (**-l**) in conjunction with **-t TYPE** to list plug-ins of a given type. When **-l** is used without a type, it defaults to type **module**. Valid types are **cache**, **callback**, **connection**, **inventory**, **lookup**, **shell**, **module**, **strategy**, and **vars**.

```
[user@host ~]$ ansible-doc -t connection -l
```

To find information about a plug-in that is not of type module, you must give its type.

```
[user@host ~]$ ansible-doc -t connection network_cli
```

Ansible Program Files

These Ansible command line programs provide important functionality.

ansible

Run ad hoc Ansible commands.

ansible-config

View, edit, and manage Ansible configuration.

ansible-doc

Ansible plug-in documentation.

ansible-inventory

Display or dump the configured inventory.

ansible-playbook

Run an Ansible Playbook, executing tasks on targeted hosts.

► Guided Exercise

Installing Ansible on the Control Node

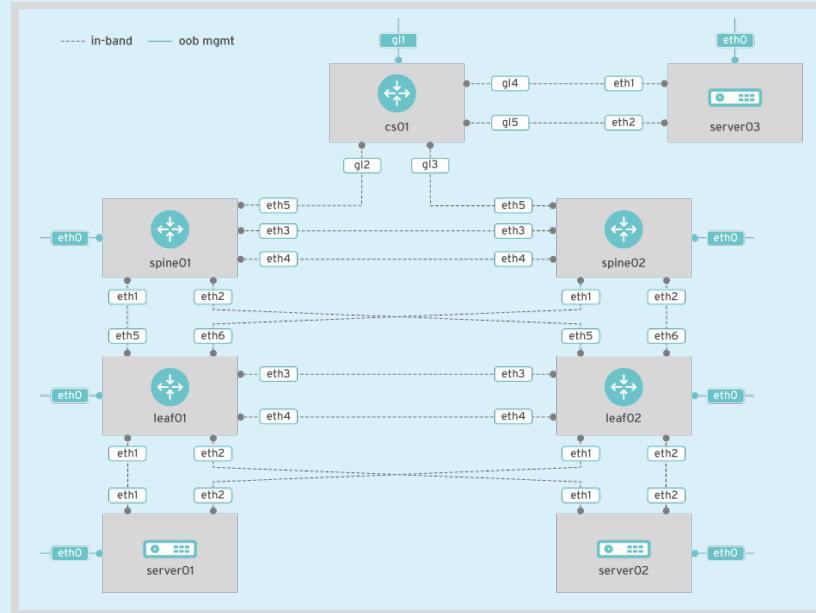


Figure 1.11: Lab network: layer 2 connectivity

Which system acts as the control node at **example.com**? The management workstation is your base of operations on the **example.com** management network. It makes sense that this system will be a control node. There can be more than one control node, but it makes sense for a number of reasons to keep the number of control nodes small enough to be manageable. Red Hat Ansible Tower has features that make it easier to safely and securely share a control node.

In order to automate the management of networking devices with Ansible, it is necessary to first install Ansible.

In this exercise, you will install Ansible on the control node, which in this case is the student **workstation** machine.

Outcomes

You should be able to:

- Install Ansible.
- Verify that the desired version was installed.

Before You Begin

Open a terminal window on the **workstation** machine. Note that you may need to enable opening pop-up windows in order for the **workstation** session to begin. Log in to the workstation VM as user **student** with password **student**. If you are ever in need of passwords, they are all listed in *Appendix B, Connection and Authentication Variables*.

Chapter1 | Deploying Ansible

- ▶ 1. Install Ansible. This installation installs a python library (**python-netaddr**) to support the advanced use of IP addresses later in the exercise.

```
[student@workstation ~]$ sudo yum install ansible python-netaddr
```

- ▶ 2. Verify that version 2.5 or later is installed.

```
[student@workstation ~]$ ansible --version
ansible 2.5.5
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/student/.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, Feb 20 2018, 09:19:12) [GCC 4.8.5 20150623 (Red
  Hat 4.8.5-28)]
```

This concludes the guided exercise.

► Guided Exercise

Identifying Resources for Installed Plug-ins

The effectiveness of knowledge workers in the technology sector is directly proportional to how well informed they are. To use Red Hat Ansible Engine as effectively as possible, it is best to be well informed. The **ansible-doc** command is a key that unlocks many capabilities that might otherwise remain hidden to you.

In this exercise, you will learn how to get more information about Ansible plug-ins that are available locally.

Outcomes

You should be able to:

- Look up features, functionality, and syntax for a plug-in you intend to use, such as the **vyos_config** module.
- Find out which plug-ins you can use by Ansible topic, such as **inventory**.
- Look up Ansible modules you can use that match a given pattern that, for instance, match a network OS type.

Before You Begin

Open a terminal window on the **workstation** machine.

- 1. Consult the command invocation help with the **--help** option for the **ansible-doc** command. This is a reliable place to start whenever you are interested in learning more about a relatively unfamiliar command.

```
[student@workstation ~]$ ansible-doc --help
Usage: ansible-doc [-l|-F|-s] [options] [-t <plugin type>] [plugin]

plugin documentation tool

Options:
  -a, --all           **For internal testing only** Show documentation for
                      all plugins.
  -h, --help          show this help message and exit
  -l, --list          List available plugins
  -F, --list_files   Show plugin names and their source files without
                      summaries (implies --list)
  -M MODULE_PATH, --module-path=MODULE_PATH
                      prepend colon-separated path(s) to module library
                      (default=[u'/root/.ansible/plugins/modules',
                      u'/usr/share/ansible/plugins/modules'])
  -s, --snippet       Show playbook snippet for specified plugin(s)
  -t TYPE, --type=TYPE Choose which plugin type (defaults to "module")
  -v, --verbose       verbose mode (-vvv for more, -vvvv to enable)
```

```
--version           connection debugging)
                   show program's version number and exit

See man pages for Ansible CLI options or website for tutorials
https://docs.ansible.com
```

From the **--help** option, you can find out which options are available to explore. What do you think the **--snippet** option would show you? If you try it with no arguments, it says "**ERROR! Incorrect options passed**," because it is expecting the name of a plug-in.

Use **--list (-l)** to get a list of available plug-ins. The **--type=TYPE (-t TYPE)** option displays a list of plug-ins of a given type. What types of plug-ins are there? You see from the "Developing Plug-ins" page at the Ansible Documentation website that the list includes callback, connection, inventory, lookup, vars, filter, and test as different types of plug-ins. Fortunately, if you try to use invalid data, the **ansible-doc** command provides more information.

```
[student@workstation ~]$ ansible-doc -t filter vyos_command
Usage: ansible-doc [-l|-F|-s] [options] [-t <plugin type> [plugin]]

ansible-doc: error: option -t: invalid choice: u'filter' (choose from 'cache',
'callback', 'connection', 'inventory', 'lookup', 'shell', 'module', 'strategy',
'vars')
```

► 2. Type **ansible-doc network_cli**.

```
[student@workstation ~]$ ansible-doc network_cli
[WARNING]: module network_cli not found in:
/home/student/.ansible/plugins/modules:/usr/share/ansible/plugins/modules:/usr/
lib/python2.7/site-packages/ansible/modules
```

That seems disappointing. Is there no information available from **ansible-doc** about **network_cli**? This is because **ansible-doc** looks up documentation for modules by default, and **network_cli** is a connection plug-in.

► 3. Type **ansible-doc -t connection network_cli**.

```
[student@workstation ~]$ ansible-doc -t connection network_cli
> NETWORK_CLI (/usr/lib/python2.7/site-packages/ansible/plugins/connection/
network_cli.py)

This connection plugin provides a connection to remote devices over
the SSH and implements a CLI shell. This connection plugin is typically
used by network devices for sending and receiving CLI commands to network
devices.
...output omitted...
```

Scroll down using the space bar to view more information. Press **q** to end the session. There is a lot of information listed here, for instance, describing potentially useful options and how to set them (shown under **set_via**).

How did you get access to all of this information? It was the **-t TYPE** option (**--type=TYPE**). How can you get more information about which plug-ins of a particular type exist? The **-t** option, fortunately, can be combined with the **-l** one (**--list**).

- 4. What do you think **strategy** plug-ins are? Type **ansible-doc -t strategy -l**.

```
[student@workstation ~]$ ansible-doc -t strategy -l
debug    Executes tasks in interactive debug session.
free     Executes tasks on each host independently
linear   Executes tasks in a linear fashion
```

- 5. Type **ansible-doc -t connection -l**. This is the list of plug-ins of type **connection**. Because **netconf** appears in this list, for instance, you know you can type **ansible-doc -t connection netconf** to view information about the **netconf** connection plug-in.

```
[student@workstation ~]$ ansible-doc -t connection -l
buildah      Interact with an existing buildah container
chroot       Interact with local chroot
docker        Run tasks in docker containers
funcd        Use funcd to connect to target
iocage        Run tasks in iocage jails
jail         Run tasks in jails
kubectl      Execute tasks in pods running on Kubernetes.
libvirt_lxc  Run tasks in lxc containers via libvirt
local        execute on controller
lxc          Run tasks in lxc containers via lxc python library
lxrd         Run tasks in lxc containers via lxc CLI
netconf      Provides a persistent connection using the netconf protocol
network_cli  Use network_cli to run command on network appliances
oc           Execute tasks in pods running on OpenShift.
paramiko_ssh Run tasks via python ssh (paramiko)
persistent   Use a persistent unix socket for connection
saltstack    Allow ansible to piggyback on salt minions
ssh          connect via ssh client binary
winrm        Run tasks over Microsoft's WinRM
zone         Run tasks in a zone instance
```

- 6. Type **ansible-doc -t inventory -l**. This displays a list of plug-ins that can be used to obtain or construct inventory information from a particular source or source format, or return inventory information in a particular format.

```
[student@workstation ~]$ ansible-doc -t inventory -l
advanced_host_list Parses a 'host list' with ranges
auto            Loads and executes an inventory plugin specified in a YAML
                config
aws_ec2         ec2 inventory source
constructed     Uses Jinja2 to construct vars and groups based on existing
                inventory.
host_list       Parses a 'host list' string
ini             Uses an Ansible INI file as inventory source.
k8s             Kubernetes (K8s) inventory source
openshift       OpenShift inventory source
```

openstack	OpenStack inventory source
script	Executes an inventory script that returns JSON
virtualbox	virtualbox inventory source
yaml	Uses a specifically YAML file as inventory source.

- 7. Type **man ansible-doc** and see if it reports who originally wrote Ansible.

```
[student@workstation ~]$ man ansible-doc
...output omitted...
AUTHOR
Ansible was originally written by Michael DeHaan.
```

This concludes the guided exercise.

Defining Ansible's Scope

Objectives

After completing this section, you should be able to identify the data files used to define Ansible's scope.

Ansible File and Data Formats

It helps to be familiar with the file and data formats used by Ansible. Here is a list of formats, broken down with sublists describing where each format is used:

- INI format
 - The Ansible configuration file, **ansible.cfg**
 - Inventory files (inventory files may also use YAML format)
- YAML
 - Playbook files
 - Inventory files, optionally
 - Other included files, such as role files
- JSON
 - The **ansible-playbook** program returns variable data in JSON form

Creating Ansible Inventories

Objectives

After completing this section, you should be able to:

- Create an Ansible host inventory.
- Inspect the contents of the inventory using the **ansible-inventory** program.

The Ansible Host Inventory

The Ansible host inventory is the source of truth for Ansible managed resources on the network. It includes all managed hosts: routers, switches, firewalls, load balancers, and other network appliances; servers, workstations, desktops, or mobile devices such as tablets or phones.

You can use the inventory to assign managed hosts to groups. And custom variables for hosts and groups can be defined by way of the inventory.

Hosts can be in multiple groups. Groups can be defined to describe many different ways in which it makes sense to map Ansible ad hoc commands or plays to hosts. This makes it possible to apply actions to multiple hosts without having to explicitly specify each one.

Would you like to apply a change to all network devices located in a particular virtual private cloud? If they all belong to a group named **vpc01_net**, use that group name in the ad hoc command or playbook.

Which Inventory File?

Ansible uses command options, the configuration file, or the file system to determine which inventory file to use.

It checks these in this order and uses the first file it finds.

1. Command option, such as **ansible-playbook -i filepath**
2. The **ansible.cfg** configuration file:

```
[defaults]
inventory = /some/other/filepath
```

3. The default locations: **/etc/ansible/hosts**

Inventory File Formats

The inventory file can be in one of many formats, depending on the inventory plug-ins you have. The most common formats are INI and YAML.

INI Format	YAML Format
<pre>[spines] spine01 spine02 [leafs] leaf01 leaf02</pre>	<pre>all: children: spines: hosts: spine01 spine02 leafs: hosts: leaf01 leaf02</pre>

Defining Hosts With Ranges

Ranges, which match all the values from START to END inclusively, can be used in inventory files to define hosts:

```
[START:END]!
```

- **192.168.[4:7].[0:255]** matches all IPv4 addresses in the 192.168.4.0/22 network (192.168.4.0 through 192.168.7.255)
- **srvr[01:20].example.com** matches all hosts named **srvr01.example.com** through **srvr20.example.com**. If leading zeros are included in numeric ranges, they are used in the pattern. This does not match **srvr1.example.com** but does match **srvr07.example.com**.
- **[a:c].dns.example.com** matches hosts named **a.dns.example.com**, **b.dns.example.com**, and **c.dns.example.com**.
- **2001:db8::[a:f]** matches all IPv6 addresses from **2001:db8::a** through **2001:db8::f**.

Nesting Host Groups

The **children** keyword can be used to create nested groups.

INI Format	YAML Format
<pre>[spines] spine[01:02] [leafs] leaf[01:02] [network:children] spines leafs</pre>	<pre>all: children: spines: hosts: spine[01:02] leafs: hosts: leaf[01:02] network: children: spines leafs</pre>

Special Host Groups

There are two special host groups: all and ungrouped.

Ungrouped

Ansible automatically puts hosts that do not have a group into the **ungrouped** group.

All

Every host in the inventory automatically belongs to the special group named **all**.

Inspecting the Inventory

Use the **ansible-inventory** command to inspect the inventory. Here are some useful options:

```
[user@host ~]$ ansible-inventory --list
[user@host ~]$ ansible-inventory --graph
[user@host ~]$ ansible-inventory --graph group
[user@host ~]$ ansible-inventory --host host
```

The **ansible** command accepts *host patterns*. The **--list-hosts** option provides a quick way to show how a host pattern resolves to a host or list of hosts.

```
[user@host ~]$ ansible -i inventory --list-hosts spine*
hosts (2):
    spine01
    spine02
```

► Guided Exercise

Creating Host Inventories

How well do you know your network and the resources you manage? You probably already know these quite well. It may seem counterproductive to be required to create yet another place to store and manage information about hosts. Yes, but consider: (1) the inventory plays a foundational role in the ecosystem of Red Hat Ansible Engine tools, (2) text files structured in INI or YAML format can be extremely effective at modeling relationships simply (with host groups and groups of groups, for instance; or by providing easy ways to associate variables with hosts and groups), and (3) static, file-based inventories illustrate important principles, but dynamic inventories unleash the ability to potentially tap into any suitably crafted API, program, or data store providing host and group information.

At **example.com**, which hosts do we put into our inventory?

In the early, start-up years of **example.com**, there were just two managed hosts: **cs01** and **server03**. That makes for a simple hosts inventory. To make matters interesting, this exercise builds an inventory that corresponds to **example.com** where it is today: eight managed network devices and servers. See *Appendix A, Table of Lab Network Hosts and Groups* for the full list.

In this exercise, you will create a static Ansible hosts inventory for the Lab Environment.

Outcomes

You should be able to:

- Create a hosts inventory file for the Lab Environment.
- Verify the syntax and contents of the inventory by inspecting it with the **ansible-inventory** tool.

Before You Begin

Open a terminal window on the **workstation** machine. This exercise involves editing a file (the **inventory** file). Vim and Atom are available on the **workstation** machine. For a Vim refresher, see *Appendix C, Editing Files with Vim*.

- 1. Create a hosts inventory file for the Lab Environment. In this guide, you will use the **cat** command to show the target contents of a file that needs editing. You do not need to use the **cat** command yourself; instead this indicates that a session with your preferred text editor is required. Using your preferred text editor, create your first version of a static inventory by creating a file named **inventory** containing this text:

```
[student@workstation ~]$ cat inventory
[leafs]
leaf01
leaf02

[spines]
spine01
```

```
spine02

[cloud-services]
cs01

[servers]
server01
server02
server03
```

- 2. Verify the syntax and contents of the inventory you created:

```
[student@workstation ~]$ ansible-inventory -i inventory --graph leafs
@leafs:
  |--leaf01
  |--leaf02
```

- 3. Explore your inventory using the **ansible** command with host patterns:

```
[student@workstation ~]$ ansible -i inventory --list-hosts 'cs*'
hosts (1):
  cs01
```

- 4. Use ranges to simplify the way group members are specified in your inventory:

```
[student@workstation ~]$ cat inventory
[leafs]
leaf[01:02]

[spines]
spine[01:02]

[cloud-services]
cs01

[servers]
server[01:03]
```

- 5. Create a group named **network** containing the groups **spines**, **leafs**, and **cloud-services**:

```
[student@workstation ~]$ cat inventory
[leafs]
leaf[01:02]

[spines]
spine[01:02]

[cloud-services]
cs01
```

Chapter 1 | Deploying Ansible

```
[network:children]
spines
leafs
cloud-services

[servers]
server[01:03]
```

- 6. To make things easier later, create explicit groups that classify your network devices by network OS: one named **ios** that contains the **cs01** host, and one named **vyos** that contains the **spines** and **leafs** groups:

```
[student@workstation ~]$ cat inventory
[leafs]
leaf[01:02]

[spines]
spine[01:02]

[cloud-services]
cs01

[ios:children]
cloud-services

[vyos:children]
spines
leafs

[network:children]
vyos
ios

[servers]
server[01:03]
```

- 7. Note that groups **all** and **ungrouped** are created automatically.

```
[student@workstation ~]$ ansible-inventory -i inventory --graph all
@all:
  |--@network:
  |  |--@ios:
  |  |  |--@cloud-services:
  |  |  |  |--cs01
  |  |  |--@vyos:
  |  |  |  |--leafs:
  |  |  |  |  |--leaf01
  |  |  |  |  |--leaf02
  |  |  |  |--@spines:
  |  |  |  |  |--spine01
  |  |  |  |  |--spine02
  |  |--@servers:
```

```
| |--server01  
| |--server02  
| |--server03  
| --@ungrouped:
```

This concludes the guided exercise.

Configuring Ansible

Objectives

After completing this section, you should be able to:

- Recognize valid locations for the Ansible configuration file.
- Describe how to determine which Ansible configuration file will be used.
- Customize the connection method, authentication details, privilege escalation, and performance.

The Ansible Configuration

Ansible settings are stored in its configuration file.

- Ansible chooses which configuration file to use from several possible locations on the control node.
- The Ansible installation provides the default configuration file `/etc/ansible/ansible.cfg`. Settings and their default values are described in this file using comments.
- A sample configuration file that documents all settings with comments is also available on the web:
<https://raw.githubusercontent.com/ansible/ansible/devel/examples/ansible.cfg>



Note

Misspelled section headings, such as `[default]`, are ignored. The correct heading is `[defaults]`.

Which Configuration File?

Ansible uses the environment and the file system to determine which configuration file to use.

It searches these in this order and uses the first file it finds (it only uses one configuration file):

1. The one specified by the `ANSIBLE_CONFIG` environment variable, if set
2. The current working directory: `./ansible.cfg`
3. A hidden file in the user's home directory: `~/.ansible.cfg`
4. The system's default configuration file: `/etc/ansible/ansible.cfg`

Recommended Practice

You do not have to do things a particular way, but these work well in many contexts.

Use Version Control

Use a Version Control System (VCS) to manage Ansible Playbooks as projects.

Store Ansible Configuration with VCS Project

Store a copy of the `ansible.cfg` file at the top level of the project.

Configure the Hosts Inventory Location

In the `[defaults]` section, use `inventory = filepath` to specify the inventory file location.

Which Configuration File?

If you are uncertain about which configuration file is being used, run this command:

```
[user@host ~]$ ansible --version
ansible 2.5.0
config file = /etc/ansible/ansible.cfg
configured module search path = [u'/root/.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, Aug 4 2017, 00:39:18) [GCC 4.8.5 20150623 (Red
Hat 4.8.5-16)]
```

Managing Settings

The Ansible configuration file uses the familiar INI format. Comments are prefixed by either the number (#) or semicolon (;).

Settings are defined using key-value pairs that fall under one of these groups:

- `defaults`
- `privilege_escalation`
- `paramiko_connection`
- `ssh_connection`
- `accelerate`
- `Selinux`

The sections that contain settings most likely to be useful in terms of customization are `defaults` and `privilege_escalation`.

Key Configuration Information

Key information that Ansible relies on to connect to hosts includes:

- Where the inventory is that lists the managed hosts and host groups.
- Which connection protocol to use to communicate with the managed hosts.
- Whether the protocol uses its standard port or a non-standard one, and if not the standard one, which port to use.
- Which identity to use to access managed hosts.
- Whether privilege escalation is needed when using an unprivileged user to access a managed host.

- Whether to prompt for an SSH password or sudo password to log in or gain privileges.

The Configuration File and Effective Settings

Customization of **ansible.cfg** establishes settings that apply when not otherwise overridden by being set elsewhere in ways that have higher precedence.

- The **ansible.cfg** file establishes a reasonable set of default behaviors.
- The recommended practice is to create an **ansible.cfg** file in a directory from which you run Ansible commands, or where playbooks are located. When an **ansible.cfg** file exists in the playbook directory, it is used instead of the configuration file in the default location (**/etc/ansible/ansible.cfg**).

Commonly Modified Settings

The table shown below shows commonly modified directives in **ansible.cfg**.

Setting	Description
inventory	The location of the Ansible inventory.
remote_user	The remote user account used to establish connections to managed hosts.
ask_pass	Prompt for a password to use when connecting as the remote user.
become	Enable or disable privilege escalation for operations on managed hosts.
become_method	The privilege escalation method to use on managed hosts.
become_user	The user account to escalate privileges to on managed hosts.
become_ask_pass	Defines whether privilege escalation on managed hosts should prompt for a password.
gathering	Set to explicit to disable gathering by default.

Sample Custom Configuration Files

The following **ansible.cfg** configuration file defaults to always prompting for the password used to access devices.

```
[defaults]
inventory = inventory
ask_pass = True
host_key_checking = False
[persistent_connection]
# custom timeout values to accommodate slow 1000V VM
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

The following **ansible.cfg** configuration file uses a Vault password file, which must be suitably protected by file-system permissions.

```
[defaults]
inventory = inventory
ask_pass = True
host_key_checking = False
```

Configuring Authentication Details

By default, Ansible connects to managed hosts using SSH, and defaults to using the same user name on remote systems as the local user running Ansible commands. To use a different identity, set the **remote_user** parameter to the desired user name.

If the local user running Ansible has private SSH keys configured that allow them to authenticate as the remote user on the managed hosts, Ansible automatically logs in. If that is not the case, you can configure Ansible to prompt the local user for the password used by the remote user by setting the directive **ask_pass = true**.

Escalating Privilege Levels

Privilege escalation settings are configured in the **[privilege_escalation]** section.

To enable privilege escalation by default, set the directive **become = true** in the configuration file.

Other privilege escalation settings include **become_method**, which defaults to **sudo**, and **become_user**, which defaults to **root**. If **become_method** requires providing a password, set **become_ask_pass = true** in the configuration file.

Ansible 2.5 added support for **become** to be used to enter enable mode (Privileged EXEC mode) on network devices that support it.

You must set the host connection type to connection: **network_cli** to use **become** for privilege escalation on network devices. Ansible 2.5.3 supports **become** for privilege escalation on **eos**, **ios**, and **nxos**.

Connection Settings: Smart or Not?

Connection decisions: SSH or not? If SSH, smart or not?

Smart (Default)

Ansible determines the most efficient way to use SSH to connect to managed hosts. For instance, Ansible uses SSH multiplexing if it can be used.

Explicit SSH Connection Plug-in

Ansible can be configured to use a particular SSH connection plug-in.

Non-SSH Connection Plug-in

Ansible can be configured to use a particular non-SSH connection plug-in.

This is important for managed network devices or Microsoft Windows systems.

Using the Local Connection Method

The special connection type **local** is available for running ad hoc commands and plays locally on the Ansible control node.

When a **localhost** entry is not listed in your inventory file, Ansible defines **localhost** implicitly so that ad hoc commands and playbooks target the control node. Ansible defaults to the **local** connection type when using the implicit **localhost** definition.

The **local** connection type ignores the **remote_user** setting and runs commands directly on the local system. If privilege escalation is used, it runs **sudo** from the user account that ran the Ansible command, not **remote_user**.

Before Ansible 2.5, network modules used **local** connections to run tasks on the control node that modified remote network devices.

Connecting to Network Devices

The netconf and network_cli connection plug-ins connect to network devices. These were introduced NEW in version 2.5.

The **NETWORK_CLI** plug-in sends and receives network device CLI commands.

http://docs.ansible.com/ansible/2.5/plugins/connection/network_cli.html

```
$ ansible-doc -t connection network_cli
```

The **NETCONF** plug-in provides a persistent connection using the netconf protocol.

<http://docs.ansible.com/ansible/2.5/plugins/connection/netconf.html>

```
$ ansible-doc -t connection netconf
```

These new connection plug-ins allow network automation plays to be written in the same style as plays used with servers.

Using Netconf or Network_Cli

Which should I use?

The target platforms (the network devices you will be managing) must be supported by the connection method you use to manage them.

Network_Cli

This connection plug-in provides a connection to remote devices over the SSH and implements a CLI shell. It is typically used by network devices for sending and receiving CLI commands to network devices. This connection method supports a broad variety of network devices, including the IOS and VyOS devices in the Lab Environment.

Netconf

This connection plug-in provides a persistent connection to remote devices over the SSH NETCONF subsystem. It is typically used by network devices for sending and receiving RPC calls

over NETCONF. This plug-in requires ncclient to be installed on the local Ansible controller. It does not currently support the network devices in the Lab Environment.

Connection Method and Play Performance

Ansible uses connection plug-ins to connect to managed hosts.

Use `ansible-doc -t connection -l` to see a list of available connection plug-ins. The important ones for network automation are `network_cli` and `netconf`. The `netconf_cli` connection plug-in currently supports the widest range of platforms. The netconf plug-in provides a connection to devices over the SSH NETCONF subsystem.

Different connection plug-ins have different performance characteristics.

How Ansible connects to hosts affects how long it takes to execute a play.

Plays typically run faster using plug-ins that support persistent connections compared to those that create and delete connections for each task.

More Play Performance Optimization

When downloading resources such as packages, use local mirrors or caches whenever possible.

The `forks` parameter determines how many hosts Ansible applies a play to in parallel. By default, Ansible sets the `forks` parameter conservatively to 5.

Whenever the number of hosts to which a play is being applied is smaller than the `fork` value, Ansible only spins up as many `ansible-playbook` processes as there are hosts in the target set.

Serializing How Hosts are Mapped to Plays

The value of the `serial` setting is used to specify how many hosts run all the way through a play before proceeding to the next set of that many hosts.

With `serial: 5`, for instance, Ansible runs five hosts all the way through the play, then runs the next five hosts all the way through the play, until it runs out of hosts targeted by the play.

If you have 100 routers, and want to only work on ten at a time, use `serial: 10`. The play is applied to ten hosts. When done with the first ten, the play is applied to the next ten hosts, all the way through the play, until it runs out of targeted hosts.

By default, `serial` is set to 0, which runs all targeted hosts through the play in parallel.

Forking Tasks for Hosts

With the `forks` setting, all the hosts in the play complete a task before moving to the next task in the play.

With `forks: 5`, for instance, Ansible talks to 5 hosts for each task at a time in parallel, and has all hosts in the play complete that task before moving to the next task in the play.

By default, `forks` is set to 5.

Combining Serial and Forks

With `forks: M` and `serial: N`, Ansible talks to `M` hosts at a time for each task, and runs them all the way through the play in groups of `N`.

Upon completing the play with the first **N** hosts, it picks **N** more hosts from the target and work through the play, doing each task across **M** hosts at a time in parallel.

► Guided Exercise

Configuring Ansible

Default settings provide safe choices for a broad range of conditions. You might be surprised, though, at how much you can improve your experience by tailoring settings to suit your particular situation.

In this exercise, you will configure Ansible.

Outcomes

You should be able to:

- Determine which configuration file Ansible uses by default.
- Customize the configuration file.
- Configure the connection method.
- Configure authentication details.
- Verify the configuration and troubleshoot any problems.

Before You Begin

This exercise requires a hosts inventory file that contains a **vyos** group, as in *Guided Exercise: Creating Host Inventories*.

Open a terminal window on the **workstation** machine.

► 1. Determine which configuration file Ansible uses by default.

- 1.1. Execute the **ansible** command with the **--version** option, and review the value of the **config file** setting:

```
[student@workstation ~]$ ansible --version
ansible 2.5.5
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/student/.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, Feb 20 2018, 09:19:12) [GCC 4.8.5 20150623 (Red
  Hat 4.8.5-28)]
```

► 2. Customize the configuration file.

- 2.1. Take a few minutes to familiarize yourself with the contents of the **/etc/ansible/ansible.cfg** file.
- 2.2. Identify, within **/etc/ansible/ansible.cfg**, which settings you would like to customize. Create an **ansible.cfg** file in the local directory. Add to it the settings you identified for customization, customized for your needs.

```
[student@workstation ~]$ cat ansible.cfg
[defaults]
host_key_checking = False
# use the local inventory file without having to use -i
inventory = inventory
# prompt us for the password so we do not have to use -k
ask_pass = True
# do not gather facts unless we ask for them
gathering = explicit

[persistent_connection]
# avoid timing out when configuring slow IOS VM
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

Later on in this course, you learn how to use Ansible Vault to encrypt sensitive data such as passwords. For the time being, whenever running ad hoc commands with the **ansible** command, or playbooks with **ansible-playbook**, Ansible prompts you to provide it interactively. The **ask_pass = True** setting in our local **ansible.cfg** file makes this happen even when **-k** or **--ask-pass** is not used.

- 2.3. Repeat the **ansible --version** command in the directory where your new **ansible.cfg** file is located. The output from the command should indicate that your local **ansible.cfg** file is now being used:

```
[student@workstation ~]$ ansible --version
ansible 2.5.0
  config file = /home/student/ansible.cfg
  configured module search path = [u'/home/student/.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, Feb 20 2018, 09:19:12) [GCC 4.8.5 20150623 (Red
  Hat 4.8.5-28)]
```

- ▶ 3. Configure the connection method.

What is the best way to set the connection method? It is very often the case that the same connection method is used to connect to all of the hosts that belong to a given group. When that is the case, it makes sense to set the connection method at the group level. Create a directory named **group_vars/** to hold files containing group variables, if it does not already exist.

```
[student@workstation ~]$ mkdir group_vars
```

Ansible supports connection methods **network_cli** and **NETCONF** for connecting to network devices. The **NETCONF** method has powerful capabilities, but not all platforms support it yet. You know that the **network_cli** connection method works with all network devices found in the Lab Network, so create a file named **network** in the **group_vars/** directory that looks like this:

```
[student@workstation ~]$ cat group_vars/network  
ansible_connection: network_cli
```

As long as this directory is found where your playbook is located (the playbook-level group variables) or where your inventory file lives (the inventory-level group variables), Ansible should find it and automatically load variables for groups.

► 4. Configure authentication details.

These credentials are used for accessing devices in the Lab Network:

OS/NOS	Ansible Group	Credentials
VyOS	spines, leafs	user: vyos, password: vyos
IOS	cloud	user: admin, password: student

Create these group files with contents as shown:

```
[student@workstation ~]$ cat group_vars/vyos  
ansible_network_os: vyos  
ansible_user: vyos  
[student@workstation ~]$ cat group_vars/ios  
ansible_network_os: ios  
ansible_user: admin
```

► 5. Verify the configuration and troubleshoot any problems.

- 5.1. Verify that the variables are set correctly for members of the **vyos** group. This is an early example of the power of ad hoc commands, which are introduced in the next chapter. This ad hoc command uses the **debug** module to print the value of the **ansible_user** variable for each host in the **vyos** host group (the **vyos** host group from the hosts inventory file you created). Do not be alarmed if the hosts provide their output in a different order than the output below. The SSH password for the VyOS devices is **vyos**.

```
[student@workstation ~]$ ansible -m debug -a "var=ansible_user" vyos  
SSH password: vyos  
leaf01 | SUCCESS => {  
    "ansible_user": "vyos"  
}  
leaf02 | SUCCESS => {  
    "ansible_user": "vyos"  
}  
spine01 | SUCCESS => {  
    "ansible_user": "vyos"  
}  
spine02 | SUCCESS => {  
    "ansible_user": "vyos"  
}
```

- 5.2. Verify that the variables are set correctly for members of the **ios** group. The SSH password for the IOS devices is **student**.

```
[student@workstation ~]$ ansible -m debug -a "var=ansible_user" ios
SSH password: student
cs01 | SUCCESS => {
    "ansible_user": "admin"
}
```

- 5.3. If all is set correctly, it ought to be possible to use an ad hoc command with the Ansible **ping** module to verify connectivity. This module indicates whether Ansible commands can be run on hosts. It happens to be named the same as the familiar network tool that performs ICMP echo requests, but the name is all they share. The Ansible **ping** module has nothing to do with ICMP.



Note

Because VyOS devices and IOS devices are using different SSH passwords, and we have not yet learned how to use Red Hat Ansible Vault to encrypt group variables, you should either test connectivity to devices individually, or use groups of devices that all use the same SSH password.

- 5.3.1. Verify connectivity to IOS devices. The SSH password for IOS devices is **student**.

```
[student@workstation ~]$ ansible -m ping ios
SSH password: student
cs01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

- 5.3.2. Verify connectivity to VyOS devices. The SSH password is **vyos** for VyOS devices.

```
[student@workstation ~]$ ansible -m ping vyos
SSH password: vyos
leaf02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
spine01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
leaf01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
spine02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

- 5.4. Download the **verify-access.yml** playbook from **materials.example.com**. In this example, **wget** is a standard tool for downloading files at the command line by way of the HTTP protocol. The backslash (\) is the shell line continuation character. The greater than symbol (>) is how the shell indicates that this is a continuation of a line. The line continuation notation is used when the command is too long to fit conveniently inside the box used on the page for displaying a command. When copying a command that uses line continuation notation, remove the secondary prompts that are represented as greater-than symbols.

```
[student@workstation ~]$ wget \
> http://materials.example.com/playbooks/verify-access.yml
```

- 5.5. The playbook you downloaded verifies that connectivity and authentication are configured correctly. It skips servers located inside the Lab Network. Those are not expected to be accessible until the Lab Network has been configured and is routing traffic correctly.

```
[student@workstation ~]$ cat verify-access.yml
---
- name: a play that verifies access to members of host group 'all'
  hosts: network
  gather_facts: no

  tasks:

    - name: "verify access to {{ inventory_hostname }}"
      ping:
        register: ping_response

    - name: debug
      debug:
        msg: "{{ ping_response }}"

    - name: assert that the response contains the string 'pong'
      assert:
        that: ping_response.ping == "pong"
```

- 5.6. Use the **ansible-playbook** command to perform the play in the **verify-access.yml** playbook. Note that because VyOS devices and IOS devices are using different SSH passwords, and you have not yet learned how to use Red Hat Ansible Vault to encrypt group variables, use the **-l SUBSET (-l SUBSET)** option to limit the scope of the playbook to targets that all use the same SSH password. The SSH password for IOS devices is **student**, and it is **vyos** for VyOS devices.

```
[student@workstation ~]$ ansible-playbook -l ios verify-access.yml
SSH password: student

PLAY [a play that verifies access to members of host group 'all'] *****

TASK [verify access to cs01] *****
ok: [cs01]

TASK [debug] *****
```

Chapter 1 | Deploying Ansible

```
ok: [cs01] => {
    "msg": {
        "changed": false,
        "failed": false,
        "ping": "pong"
    }
}

TASK [assert that the response contains the string 'pong'] ****
ok: [cs01] => {
    "changed": false,
    "msg": "All assertions passed"
}

PLAY RECAP ****
cs01 : ok=3    changed=0    unreachable=0    failed=0
```

- 5.7. Download the **show-current-access-vars.yml** playbook from materials.example.com:

```
[student@workstation ~]$ wget \
> http://materials.example.com/playbooks/show-current-access-vars.yml
```

- 5.8. This playbook you downloaded displays the values of important connection and authentication variables.

```
[student@workstation ~]$ cat show-current-access-vars.yml
---
- name: a play that exposes the current access vars
  hosts: network
  gather_facts: no

  tasks:
  - name: show the value of key variables
    debug:
      msg: >
        host: {{ inventory_hostname }},
        con: {{ ansible_connection }},
        nos: {{ ansible_network_os }},
        user: {{ ansible_user }},
        pass: {{ ansible_ssh_pass }}
```

- 5.9. Use the **ansible-playbook** command to perform the play in the **show-current-access-vars.yml** playbook. Because VyOS devices and IOS devices are using different SSH passwords, and we have not yet learned how to use Red Hat Ansible Vault to encrypt group variables, use the **--limit=SUBSET (-l SUBSET)** option to limit the scope of the playbook to targets that all use the same SSH password. The SSH password for IOS devices is **student**, and it is **vyos** for VyOS devices.

```
[student@workstation ~]$ ansible-playbook -l vyos show-current-access-vars.yml
SSH password: vyos
```

```
PLAY [a play that exposes the current access vars] ****
```

```
TASK [show the value of key variables] ****
ok: [spine01] => {
    "msg": "host: spine01, con: network_cli, nos: vyos, user: vyos, pass: vyos\n"
}
ok: [leaf01] => {
    "msg": "host: leaf01, con: network_cli, nos: vyos, user: vyos, pass: vyos\n"
}
ok: [spine02] => {
    "msg": "host: spine02, con: network_cli, nos: vyos, user: vyos, pass: vyos\n"
}
ok: [leaf02] => {
    "msg": "host: leaf02, con: network_cli, nos: vyos, user: vyos, pass: vyos\n"
}

PLAY RECAP ****
leaf01                  : ok=1    changed=0    unreachable=0    failed=0
leaf02                  : ok=1    changed=0    unreachable=0    failed=0
spine01                 : ok=1    changed=0    unreachable=0    failed=0
spine02                 : ok=1    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

▶ Lab

Deploying Ansible

In this lab, you will create a `~/proj/` directory containing an `ansible.cfg` file, a static `inventory` file, and a `group_vars/` directory. In the `group_vars/` directory, create group files that set values for variables that make it possible to connect and authenticate automatically to devices.

Outcomes

You should be able to create a simple Ansible project directory that contains a static hosts inventory, an Ansible configuration file, and a `group_vars/` directory with files that sets variables needed to automate the connection and authentication process.

Before You Begin

Open a terminal window on the `workstation` VM.

Lab Network Managed Resources

Host	OS/NOS	Credentials
spine01	VyOS	user: vyos , password: vyos
spine02	VyOS	user: vyos , password: vyos
leaf01	VyOS	user: vyos , password: vyos
leaf02	VyOS	user: vyos , password: vyos
cs01	IOS	user: admin , password: student

Instructions

1. Create and change into the `~/proj/` project directory.
2. Create an Ansible hosts `inventory` file suitable for managing devices shown in the *Lab Network Managed Resources* table.
3. Verify the inventory.
4. Create a local `ansible.cfg` file that contains your customizations.
5. Create a `~/proj/group_vars` directory to contain further customizations.
6. Populate the `group_vars/` directory with group files that set the connection method and authentication details (`ansible_user`).
7. Check your work.

This concludes the lab.

► Solution

Deploying Ansible

In this lab, you will create a `~/proj/` directory containing an `ansible.cfg` file, a static `inventory` file, and a `group_vars/` directory. In the `group_vars/` directory, create group files that set values for variables that make it possible to connect and authenticate automatically to devices.

Outcomes

You should be able to create a simple Ansible project directory that contains a static hosts inventory, an Ansible configuration file, and a `group_vars/` directory with files that sets variables needed to automate the connection and authentication process.

Before You Begin

Open a terminal window on the **workstation** VM.

Lab Network Managed Resources

Host	OS/NOS	Credentials
spine01	VyOS	user: vyos , password: vyos
spine02	VyOS	user: vyos , password: vyos
leaf01	VyOS	user: vyos , password: vyos
leaf02	VyOS	user: vyos , password: vyos
cs01	IOS	user: admin , password: student

Instructions

1. Create and change into the `~/proj/` project directory.

```
[student@workstation ~]$ mkdir ~/proj
[student@workstation ~]$ cd ~/proj
```

2. Create an Ansible hosts `inventory` file suitable for managing devices shown in the *Lab Network Managed Resources* table.

Use your favorite text editor to create the `inventory` file. Ensure it has contents similar to the following:

```
[leafs]
leaf[01:02]
```

```
[spines]
spine[01:02]
```

```
[cloud-services]
cs01

[ios:children]
cloud-services

[vyos:children]
spines
leafs

[network:children]
vyos
ios

[servers]
server[01:03]
```

3. Verify the inventory.

```
[student@workstation proj]$ ansible-inventory -i inventory --graph
@all:
  |--@network:
  |  |--@ios:
  |  |  |--@cloud-services:
  |  |  |  |--cs01
  |  |  |--@vyos:
  |  |  |  |--@leafs:
  |  |  |  |  |--leaf01
  |  |  |  |  |--leaf02
  |  |  |  |--@spines:
  |  |  |  |  |--spine01
  |  |  |  |  |--spine02
  |--@servers:
  |  |--server01
  |  |--server02
  |  |--server03
  -@ungrouped:
```

4. Create a local **ansible.cfg** file that contains your customizations.

Use your favorite text editor to create an **ansible.cfg** file with contents similar to the following:

```
[defaults]
host_key_checking = False
inventory = inventory
ask_pass = True
gathering = explicit

[persistent_connection]
```

Chapter1 | Deploying Ansible

```
# avoid timing out when configuring slow 102 VM
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

5. Create a **~/proj/group_vars** directory to contain further customizations.

```
[student@workstation proj]$ mkdir ~/proj/group_vars
```

6. Populate the **group_vars/** directory with group files that set the connection method and authentication details (**ansible_user**).

Use your favorite text editor to create **group_vars** files for the **network**, **vyos**, and **ios** host groups. Create them with the following contents:

```
[student@workstation proj]$ cat group_vars/network
ansible_connection: network_cli
[student@workstation proj]$ cat group_vars/vyos
ansible_network_os: vyos
ansible_user: vyos
[student@workstation proj]$ cat group_vars/ios
ansible_network_os: ios
ansible_user: admin
```

7. Check your work.

```
[student@workstation proj]$ ansible -m debug -a "var=ansible_user" vyos
SSH password: vyos
leaf01 | SUCCESS -> {
    "ansible_user": "vyos"
}
leaf02 | SUCCESS -> {
    "ansible_user": "vyos"
}
spine01 | SUCCESS -> {
    "ansible_user": "vyos"
}
spine02 | SUCCESS -> {
    "ansible_user": "vyos"
}

[student@workstation proj]$ ansible -m ping vyos
SSH password: vyos
leaf01 | SUCCESS -> {
    "changed": "false"
    "ping": "pong"
}
leaf02 | SUCCESS -> {
    "changed": "false"
    "ping": "pong"
}
spine01 | SUCCESS -> {
    "changed": "false"
    "ping": "pong"
}
```

```
    }
    spine02 | SUCCESS -> {
        "changed": "false"
        "ping": "pong"
    }
```

A hosts inventory, configuration file, and group variables supporting this particular inventory form a collection that provides an operational foundation for future automation projects. This would be a good time to commit and push it to a Git repository.

This concludes the lab.

Chapter 2

Running Commands and Plays

Goal

Run automated tasks on devices using plays and ad hoc commands.

Objectives

- Run ad hoc commands to execute single, one-time tasks.
- Write playbooks, run plays with **ansible-playbook**, and interpret the resulting output.
- Build more complex plays that include multiple tasks.
- Write playbooks that include multiple plays.

Sections

- Executing Ad Hoc Commands (and Guided Exercise)
- Preparing Ansible Playbooks (and Guided Exercise)
- Building a Play with Multiple Tasks (and Guided Exercise)
- Composing Playbooks with Multiple Plays (and Guided Exercise)

Lab

Running Commands and Plays

Executing Ad Hoc Commands

Objectives

After completing this section, you should be able to run ad hoc commands to execute single, one-time tasks.

Introducing Ad Hoc Commands

An ad hoc command is a single, manually run Ansible task that you want to perform quickly and do not need to save to run again later.

This is the form of an ad hoc command:

```
ansible host-pattern -m module [-a 'module arguments'] [-i inventory]
```

When module arguments are enclosed within single or double quote marks, the space that follows the **-a** option and the arguments is optional.

Running Ad Hoc Commands

When are ad hoc commands useful? It depends on the module.

To enable the **netconf** service on a platform that supports it, run the following command:

```
[user@host ~]$ ansible -m junos_netconf host-identifier
```

To test Ansible connectivity and authentication to a managed resource (not ICMP), run the following command:

```
[user@host ~]$ ansible -m ping host-identifier
```

To determine if you can reach an IP address from a managed resource (using ICMP), run the following command:

```
[user@host ~]$ ansible -m ios_ping host-identifier -a "dest=ip-address"
```

You can pass arguments to modules with the **-a** option.

```
[user@host ~]$ ansible -m vyos_command -a "commands='command'" host-identifier
```

More Examples of Ad Hoc Commands

To gather facts about an IOS network device, run the following command:

```
[user@host ~]$ ansible -m ios_facts ios-device-host-identifier
```

To gather facts about a JunOS device, run the following command:

```
[user@host ~]$ ansible -m junos_facts junos-device-host-identifier
```

► Guided Exercise

Executing Ad Hoc Commands

In this exercise, you will execute ad hoc commands on multiple managed hosts.

Outcomes

You should be able to execute a command on a group of hosts on ad hoc basis.

Before You Begin

You should have set the value of variables to support connection and authentication, as described in the Deploying Ansible exercise. The rest of the exercises in this section have the same dependency.

Open a terminal window as the **student** user on the **workstation** VM.

Instructions

- ▶ 1. Change to the `~/proj/` directory created in Lab 1.
- ▶ 2. Verify that the **cs01** device can ping the **tower** machine (**172.25.250.9**).

```
[student@workstation proj]$ ansible -m ios_ping -a "dest=172.25.250.9" cs01
SSH password: student
cs01 | SUCCESS => {
    "changed": false,
    "commands": [
        "ping 172.25.250.9"
    ],
    "packet_loss": "20%",
    "packets_rx": 4,
    "packets_tx": 5,
    "rtt": {
        "avg": 2,
        "max": 4,
        "min": 1
    }
}
```

- ▶ 3. Use the **traceroute** command to inspect the path between **cs01** and the management workstation.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='traceroute 172.25.250.254 probe 2 timeout 2'" cs01
SSH password: student
cs01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "Tracing route to 172.25.250.254 over a maximum of 2 hops\n"
    ]
}
```

```
"Type escape sequence to abort.\nTracing the route to 172.25.250.254\nVRF
info:
(vrf in name/id, vrf out name/id)\n 1 172.25.250.254 4 msec 1 msec"
],
"stdout_lines": [
  [
    "Type escape sequence to abort.",
    "Tracing the route to 172.25.250.254",
    "VRF info: (vrf in name/id, vrf out name/id)",
    " 1 172.25.250.254 4 msec 1 msec"
  ]
]
}
```

- ▶ 4. An extremely useful command for VyOS devices is **sh conf com**, which displays the running configuration. You will see other ways of obtaining the running configuration (**vyos_conf** module with **backup=yes**, **vyos_facts** with **subset=all**).

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh conf com'" spine01
SSH password: vyos
spine01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "...output omitted..."
  ]
}
```

This concludes the guided exercise.

Preparing Ansible Playbooks

Objectives

After completing this section, you should be able to:

- Create a simple playbook.
- Check playbook syntax with **ansible-playbook --syntax-check**.

Tasks

A task is the application of a module to perform a specific unit of work. An example of a task in a play is shown below.

```
tasks:  
  - name: Backup configuration  
    ios_config:  
      backup: yes
```

Plays and Playbooks

A play is a sequence of tasks to be applied, in order, to one or more hosts, whereas a playbook is a YAML file containing one or more plays.

Visualizing Plays and Playbooks

An example of a play in a playbook is shown below.

```
---  
- name: backup router configurations  
  hosts: routers  
  connection: network_cli  
  gather_facts: no  
  
  tasks:  
    - name: gather ios_facts  
      ios_facts:  
        register: version  
  
    - debug:  
      msg: "{{version}}"  
  
    - name: Backup configuration  
      ios_config:  
        backup: yes
```

Interpreting YAML

YAML is a human-friendly language that concisely represents objects as data. Additional characteristics of YAML include:

- Files start with three dashes (---) that mark the start of the document.
- Comments begin with the pound sign (#).
- Indentation is significant, and spaces must be used rather than tabs.
- Elements at the same level (items in the same list, for instance) must have the same indentation.
- Children of an item are indented more than the parent.

YAML Object Types Using Block Style

Unquoted scalar

```
Network
```

Sequence of scalars

```
- 10.0.0.1
- 192.168.0.1
- 172.16.0.1
```

Mapping scalars to scalars

```
IPv4: 10.10.17.42/24
IPv6: "fd42:e5a1:ef5d:6030:0:0:0:2/64"
```

Mapping scalars to sequences

```
rtr01:
  - GigabitEthernet1
  - GigabitEthernet2

rtr02:
  - eth0
  - eth1
```

Sequence of mappings

```
- name: GigabitEthernet2
  ipv4: 172.16.2.2/30
  ipv6: "fd42:e5a1:ef5d:6030:0:0:0:2/64"

- name: eth1
  ipv4: 10.10.10.1/30
  ipv6: "fdbcbda:8486:7118:0:0:0:1/64"
```

Mapping of mappings

```

vyos:
  ansible_network_os: vyos
  ansible_user: vyos

ios:
  ansible_network_os: ios
  ansible_user: admin

```

YAML Object Types Using Flow Style

Sequence of scalars

```
[ GigabitEthernet1, GigabitEthernet2, GigabitEthernet3 ]
```

Mapping scalars to scalars

```
{ name: eth1, ipv4: 10.10.5.1/30, ipv6: "fdb5:4b4e:4574:c6bb:0:0:0:1/64" }
```

Sequence of sequences

```

- [name, ipv4, ipv6]
- [GigabitEthernet1, 172.16.2.2/30, "fdb5:4b4e:4574:c6bb:0:0:0:1/64"]
- [GigabitEthernet2, 10.10.5.2/30, "fdb5:4b4e:4574:c6bb:0:0:0:2/64"]

```

Sequence of mappings

```

- { name: GigabitEthernet2, ipv4: 172.16.2.2/30 }
- { name: GigabitEthernet4, ipv4: 172.16.10.1/30 }

```

Mapping of mappings

```

GigabitEthernet1: {ipv4: 10.0.0.1/30, ipv6: "fdb5:4b4e:4574:c6bb::1/64"}
GigabitEthernet2: {
  ipv4: 192.168.5.1/30,
  ipv6: "fdea:230f:c3cf:c287:0:0:0:1/64"
}

```

Encoding Plays in YAML

How are *Ansible* plays encoded in YAML?

- A playbook is a list of one or more plays.
- Each play is a hash/dictionary; that is to say, a YAML sequence of **key:value** mappings.
- A play must include **host** and **tasks** mappings, and may include a **name** mapping, as well as other mappings depending on which plug-ins are being used.
- Plays can import or include files, other playbooks, task lists, and so on.

Mapping the Terrain

Which **key:value** pairs are used to define plays in playbooks?

- Plays can be named or anonymous. It is useful to name your plays. For example, **name:playname**. Ideally, a name communicates clearly the purpose of a play.
- Plays take a list of tasks and apply them to a list of hosts or host groups.

```
---
- name: a play that backs up configs
  hosts:
    - routers

  tasks:
    - name: backup the running config
      ios_config:
        backup: yes
```

Choosing a Module: ping or ios_ping

ping

- A trivial test module that always returns pong on successful contact.
- This is not an ICMP ping, and requires Python on the remote node.
- It usually does not make sense in playbooks, but is useful from **/usr/bin/ansible** to verify the ability to log in and that a usable Python is configured.

ios_ping

- Tests reachability using ping from network device to a remote destination using available routes. This module is specific to devices running the IOS network operating system.

Running ios_ping Ad Hoc Command

Tests reachability by pinging from an IOS network device to a remote destination.

```
[user@host ~]$ ansible -m ios_ping host-identifier -a "dest=ip-address"
```

Using ios_ping in a Playbook

Tests reachability, in a playbook, by pinging from a network device to a remote destination.

```
---
- name: a reachability test
  hosts: rtr1

  tasks:
    - name: "test reachability to rtr1"
      ios_ping:
        dest: rtr1
```

► Guided Exercise

Converting an Ad Hoc Command to a Play

The ability to convert an ad hoc command into a play is a skill that serves as a bridge from the simplicity of the command line to the power of the playbook.

In this exercise, you will take the ad hoc command used in *Guided Exercise: Executing Ad Hoc Commands*, convert it into a simple play in a playbook, and run it using the **ansible-playbook** command.

Outcomes

You should be able to:

- Translate an ad hoc command into a simple play in a playbook.
- Verify the YAML syntax of the playbook.
- Run the play.

Before You Begin

This exercise assumes you know how to set the value of variables to support connection and authentication, as described in the *Lab: Deploying Ansible* exercise.

Open a terminal window on the **workstation** VM. If you use the Vim text editor, configure it to automatically translate the **Tab** key into two spaces. White space is significant to YAML, and it expects files to be structured in increments of two spaces. Adding this line to **~/.vimrc** may make that easier:

```
autocmd FileType yaml setlocal ai ts=2 sw=2 et
```

For an overview of how to use the Vim text editor, see *Appendix C, Editing Files with Vim*.

Instructions

- 1. Change to the **~/proj/** directory created in Lab 1.
- 2. Create a file named **iosping1.yml** with the following content:

```
---
- name: A reachability test
  hosts: cs01

  tasks:

    - name: Test reachability to 172.25.250.9
      ios_ping:
        dest: 172.25.250.9
```

It is assumed that the **[defaults]** section of your **ansible.cfg** file contains the line **gathering = explicit**. If it does not, you should include the line **gather: False** at the top of the play: after hosts and before tasks, for instance.

- 3. Check the syntax of the playbook you created.

```
[student@workstation proj]$ ansible-playbook --syntax-check iosping1.yml  
playbook: iosping1.yml
```

- 4. Use the **ansible-playbook** command to run the play in your playbook. The SSH password for **cs01** is **student**.

```
[student@workstation proj]$ ansible-playbook iosping1.yml  
SSH password: student  
  
PLAY [A reachability test] ****  
  
TASK [Test reachability to 172.25.250.9] ****  
ok: [cs01]  
  
PLAY RECAP ****  
cs01 : ok=1    changed=0    unreachable=0    failed=0
```

- 5. Use a text editor to create a file named **vyos-sh-conf1.yml** with the following content:

```
---  
- name: Show the running config of a VyOS system  
  hosts: spine01  
  
  tasks:  
  
    - name: execute the command  
      vyos_command:  
        commands:  
          - sh conf com  
        register: result  
  
    - name: show result  
      debug:  
        var: result
```

- 6. Use the **ansible-playbook** command to run the play in your playbook. The SSH password for **spine01** is **vyos**.

```
[student@workstation proj]$ ansible-playbook vyos-sh-conf1.yml  
SSH password: vyos  
...output omitted
```

This concludes the guided exercise.

Building a Play with Multiple Tasks

Objectives

After completing this section, you should be able to build more complex plays that include multiple tasks.

Setting the Connection Method

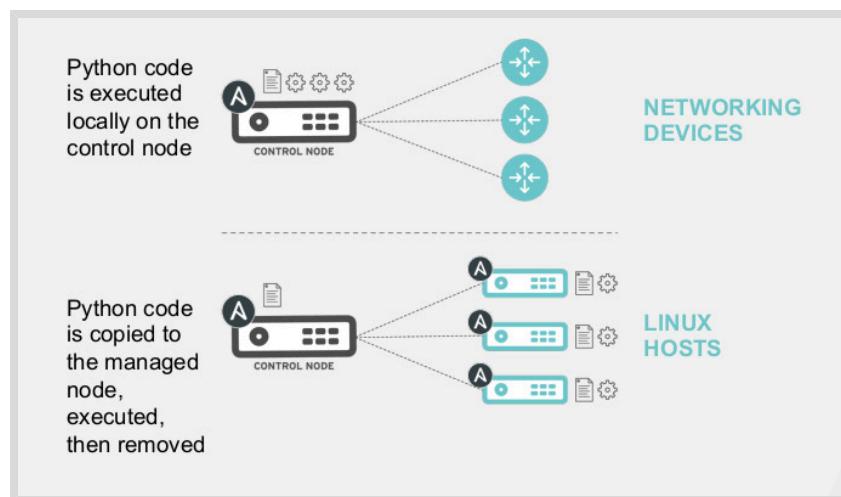
Use **connection: method** to set the connection method in a playbook. Privilege escalation can be customized using **become**, **become_user**, and **become_method**.

```
---
- name: a simple playbook with one play and one task
  hosts: spine1
  connection: network_cli
  remote_user: admin
  become: yes
  become_method: enable
  vars:
    destination: 10.10.10.4

  tasks:
    - name: "test reachability to {{ destination }}"
      ios_ping:
        dest: "{{ destination }}"
```

Local Execution of Python Code

The connection methods for connecting to networking devices execute Ansible python code locally on the control node.



Long and Short Mapping Format Method

Mapping Shorthand or Long Form

For consistent formatting and improved readability, the recommended practice is to use the **key : value** syntax for mappings rather than the **key=value** shorthand.

The **key : value** form also facilitates fine-grained, line-oriented syntax error alerting.

The following is valid, but not recommended:

```
- name: print a message
  debug: msg="Welcome to the example.com network"
```

The preferred format is as follows:

```
- name: print a message
  debug:
    msg: "This is the example.com network"
```

Long and Short Sequence Format

Sequence Shorthand or Long Form

For similar reasons, using the longer - **item** block style rather than the shorter [**item**, ...] flow style is recommended practice for representing sequences.

The following is valid, but not recommended:

```
vars:
  interfaces: [ GigabitEthernet1, GigabitEthernet2, GigabitEthernet3 ]
```

The preferred format is as follows:

```
vars:
  interfaces:
    - GigabitEthernet1
    - GigabitEthernet2
    - GigabitEthernet3
```

Notice that there are no commas after list elements in the preferred style.

Context Makes a Difference

In certain contexts, flow style may be preferred.

Sequences of Mappings

Flow style can be an effective way of promoting clarity, by conveying how elements in a collection are grouped together as a unit. This is frequently the case with sequences of mappings; with roles, for instance, or when loops or aggregation are used.

```

roles:
- { role: git_server, git_project: project }
- { role: git_server, git_project: ipa-client-register }

---
- hosts: routers
gather_facts: no
vars:
  interface_data:
    cs01:
      - { name: Loopback1, ipv4: 172.16.0.1/32 }
      - { name: GigabitEthernet2, ipv4: 172.16.2.2/30 }
      - { name: GigabitEthernet3, ipv4: 172.16.5.2/30 }
      - { name: GigabitEthernet4, ipv4: 172.16.10.1/30 }
  tasks:
    - name: set IP address on IOS device
      ios_l3_interface:
        aggregate: "{{ interface_data[inventory_hostname] }}"
        when: ansible_network_os == 'ios'

```

Multiline Strings

Look for opportunities to break long strings into multiline ones. For example:

```

vars:
  line: this is a very long line that seems to go on and on and on

```

The preferred format is shown below:

```

vars:
  line: >
    this is a very long line
    that seems to go on and on
    and on and on and on

```

Both of those are identical strings as far as YAML is concerned.

Assessing What Happened

The **ansible-playbook** command displays color-coded information about what is happening as plays are running and executing tasks on hosts.

Color Key

Color	Text	Meaning
green	ok	Normal, successful operation
yellow	changed	Host was changed by task
red	error	Error or failure

You can register the output of commands associated with a task, and conditionally perform later tasks based on the status of earlier ones (changed, ok, or error).

Understanding Play Output

The **ansible-playbook** command displays output in JSON form.

Play

Information about which hosts were targeted by the play

Task

Success or failure of task for host, plus any output from task

Gathering Facts

(unless suppressed): Success or failure of fact gathering for host

Play Recap

For each host, the number of tasks resulting in ok, changed, unreachable, and failed

Executing Ansible Playbooks

The **ansible-playbook** program runs the plays found in playbooks and much more. This is why it is good to name plays and tasks to clearly communicate their purpose.

List the hosts to which a playbook would be applied:

```
[user@host ~]$ ansible-playbook --list-hosts playbook-file
```

List the tasks contained in a playbook:

```
[user@host ~]$ ansible-playbook --list-tasks playbook-file
```

Verify the syntax of a playbook:

```
[user@host ~]$ ansible-playbook --syntax-check playbook-file
```

Return information about changes that would be made, but do not make them ("dry run"):

```
[user@host ~]$ ansible-playbook --check playbook-file
```

Power Plays

The first playbook example illustrated how to take an ad hoc command and convert it to a simple playbook. Plays do not begin to show their potential, however, until they do multiple tasks.

```
---
- name: play that sets interface descriptions
  hosts: cloud-services
  connection: network_cli
  tasks:
    - name: backup running config
      ios_config:
```

```
backup: yes

- name: label outside interface
  ios_interface:
    name: GigabitEthernet2
    description: outside

- name: label inside interface
  ios_interface:
    name: GigabitEthernet4
    description: inside
```

► Guided Exercise

Building a Play with Multiple Tasks

In this exercise, you will build a play with multiple tasks.

Outcomes

You should be able to:

- Construct a playbook that uses multiple tasks.
- Verify the YAML syntax of the playbook.
- Run the play.

Before You Begin

It is assumed you know how to set the value of variables to support connection and authentication, as described in the *Lab: Deploying Ansible* exercise.

Open a terminal window as the **student** user on the **workstation** VM.

- ▶ 1. Change to the `~/proj/` directory created in Lab 1.
- ▶ 2. Use your preferred text editor to create a file called `ios-checksys1.yml` with the following contents:

```
---
- name: back up config and look at device health indicators on ios devices
  hosts: ios

  tasks:
    - name: backup the device configuration
      ios_config:
        backup: yes

    - name: look at device health indicators
      ios_command:
        commands:
          # this provides hostname and uptime
          - sh ver | include uptime
          - sh ip domain
          - sh clock
          - sh ip name-server
          - sh proc mem | include Total
      register: results

    - name: show results
```

```
debug:
  msg: "{{ item }}"
loop: "{{ results.stdout_lines }}"

```

YAML is extremely strict about formatting, so pay special attention to indentation. Use two spaces (not tabs) when indenting lines. If you configured Vim as specified in the preceding exercise, then **Tab** indents by two spaces.

It is considered good practice to back up the running configuration for network devices often. This playbook displays several key system indicators (uptime, domain name, clock time, and so forth), and also makes a backup copy of the current (running) configuration.

The **register** module associates the output of a command with a variable. In this example the variable name is **results**. The **debug** module displays the **stdout** property of the **results** variable. This play also introduces a **loop** concept, which is explored in more depth later.

- ▶ 3. Check the syntax of the playbook you created.

```
[student@workstation proj]$ ansible-playbook --syntax-check ios-checksys1.yml
playbook: ios-checksys1.yml
```

- ▶ 4. Use the **ansible-playbook** command to run the play in your playbook:

```
[student@workstation proj]$ ansible-playbook ios-checksys1.yml
SSH password: student

PLAY [back up config and look at device health indicators on ios devices] *****

TASK [backup the device configuration] *****
ok: [cs01]

TASK [look at device health indicators] *****
ok: [cs01]

TASK [show results] *****
ok: [cs01] => (item=None) => {
    "msg": [
        "cs01 uptime is 2 hours, 38 minutes"
    ]
}
ok: [cs01] => (item=None) => {
    "msg": [
        "lab.example.com"
    ]
}
ok: [cs01] => (item=None) => {
    "msg": [
        "**19:06:11.781 UTC Sun Jul 26 2020"
    ]
}
ok: [cs01] => (item=None) => {
    "msg": [
        "255.255.255.255"
    ]
}
```

```
        ]
    }
ok: [cs01] => (item=None) => {
    "msg": [
        "Processor Pool Total: 2092339280 Used: 333676544 Free: 1758662736",
        " lsmpi_io Pool Total: 3149400 Used: 3148568 Free:          832",
        "                                336777680 Total"
    ]
}

PLAY RECAP ****
cs01           : ok=3    changed=0   unreachable=0   failed=0
```

This concludes the guided exercise.

Composing Playbooks with Multiple Plays

Objectives

After completing this section, you should be able to write playbooks that include multiple plays.

Larger Playbooks

Playbooks can run multiple plays. That means you can apply a different succession of tasks to different hosts.

This playbook contains two plays: one that configures **foo** hosts with the domain name `foo.com` and another that configures **bar** hosts with `bar.com`.

```
---
- name: a play that sets the domain name to foo.com
  hosts: foo
  vars:
    domain_name: foo
  tasks:
    - name: "set domain name to {{ domain_name }}"
      ios_config:
        lines:
          - "ip domain-name {{ domain_name }}"

- name: a play that sets the domain name to bar.com
  hosts: bar
  vars:
    domain_name: bar
  tasks:
    - name: "set domain name to {{ domain_name }}"
      ios_config:
        lines:
          - "ip domain-name {{ domain_name }}"
```

Using Ansible Playbook Options

As a general rule, use command-line options available with the **ansible-playbook** command:

- When checking the syntax of a playbook (the **--syntax-check** option) or predicting some of the changes that may occur (the **--check** or **-C** option).
- When exploring the possibilities: testing or trying out alternative settings.
- When troubleshooting, to see if different settings would help.
- When it is necessary to deviate from your usual settings to deal with outliers; to connect to a remote host not ordinarily managed by your team, for instance.

Checking the syntax of playbooks you edit, before running them, is recommended. If a different setting involves something you do on a regular basis, consider editing **ansible.cfg** to modify

the default value, or use a group or host variable to associate the value with a group or host. If it only applies to a particular play or task, use a variable at the play or task level to explicitly set it there.

Overriding the Connection Settings

Options available with the **ansible-playbook** command modify the connection settings used when that command is executed:

- Use the **--ask-pass** (or **-k**) option when you want to be prompted to provide the password used to access a device.
- If you need to access a device that requires a non-default private key, you can use the **--key-file=PRIVATE_KEY_FILE** option to explicitly specify it.
- To connect as a different user, use the **--user=REMOTE_USER** (or **-u REMOTE_USER**) option.
- If you are troubleshooting attempts to access a remote system, and you want to try a different connection type, use **--connection=CONNECTION** (or **-c CONNECTION**).
- If you are experiencing difficulties connecting, and you want to test whether increasing the connection timeout would help, use the **--timeout=TIMEOUT** (or **-T TIMEOUT**) option.

Overriding Privilege Escalation

The **ansible-playbook** command provides options that modify privilege escalation.

- If a password is required to escalate privilege, and you want to be prompted to provide it, use the **--ask-become-pass** (or **-K**) option.
- To perform tasks using the **become** method of privilege escalation, use the **--become** (or **-b**) option. The default **become** method is **sudo**.
- To specify a different become method, use the **--become-method=BECOME_METHOD** option. Valid choices are **sudo**, **su**, **enable**, **pbrun**, **pfexec**, **doas**, **dzdo**, **ksu**, **runas**, and **pmrunk**.
- To run privileged operations as a different user, use the **--become-user=BECOME_USER** option (default is **root**).

Customizing Playbook Execution

The **ansible-playbook** command has other options for modifying many aspects of playbook behavior, some of which are described below. Use **ansible-playbook --help** to see all available options.

- To explicitly specify a particular inventory file or dynamic inventory program, use the **--inventory=INVENTORY** (or **-i INVENTORY**) option.
- To limit the hosts targeted by the plays in a playbook to a specified subset of what the hosts field of the plays stipulates, use the **--limit=SUBSET** (or **-l SUBSET**) option.
- To incorporate additional data into a playbook, for example to customize or extend functionality in ways not supported by predefined variables, use the **--extra-vars=EXTRA_VARS** (or **-e EXTRA_VARS**) option.

Specifying Hosts with Patterns

Host patterns tell Ansible which inventory hosts to target for a play or ad hoc command.

Chapter 2 | Running Commands and Plays

It is easier and more flexible to adapt patterns to control the hosts that a play targets than to embed conditional logic in playbooks. Note that the **--list-hosts** option displays which hosts are effectively targeted with a pattern.

Using a host identifier as the selection pattern:

```
[user@host ~]$ ansible ftp -i myinventory --list-hosts
hosts (1):
  ftp
```

Using a group identifier as the selection pattern:

```
[user@host ~]$ ansible spines -i myinventory --list-hosts
hosts (2):
  spine01
  spine02
```

Wildcards and Logic in Patterns

Using a wildcard in the selection pattern:

```
[user@host ~]$ ansible 'server0*' -i myinventory --list-hosts
hosts (2):
  server01
  server02
```

Using logical negation in the selection pattern:

```
[user@host ~]$ ansible 'datacenter,!servers' -i myinventory --list-hosts
hosts (4):
  leaf01
  leaf02
  ...
  ...
```

Using a logical conjunction (intersection) as the selection pattern:

```
[user@host ~]$ ansible 'servers,&web' -i myinventory --list-hosts
hosts (1):
  web
```

Overlap Between Hosts and Groups

Exercise care when using patterns to target hosts. If the pattern matches one or more host identifiers and also matches one or more group identifiers, the result might not be what you intended.

Consider the following inventory snippet:

```
[servers]
server01
server02
accounting01
www01
```

The pattern **serv*** matches the group name **servers**, and therefore includes the **accounting01** and **www01** hosts, which might not be what you intended:

```
[user@host ~]$ ansible 'serv*' -i inventory --list-hosts
hosts (4):
server01
server02
accounting01
www01
```

► Guided Exercise

Composing Playbooks with Multiple Plays

In this exercise, you will compose a playbook that contains multiple plays. Each play will target a different host. Different targets will have different login passwords. This course has not covered how to use Ansible Vault to encrypt files or variables containing sensitive data yet, so the `--extra-vars` option is used to pass two different passwords to two different plays in the same playbook.

Outcomes

You should be able to:

- Compose a playbook that has multiple plays.
- Verify the YAML syntax of the playbook.
- Run the plays.

Before You Begin

It is assumed you know how to set the value of variables to support connection and authentication, as described in the *Lab: Deploying Ansible* exercise.

Open a terminal window on the **workstation** VM.

- ▶ 1. Change to the `~/proj/` directory created in Lab 1.
- ▶ 2. Create a file called `multi-vendor-backup.yml` with the following contents:

```
---
- name: back up config from a VyOS device
  hosts: spine01
  vars:
    ansible_password: "{{ vyos_pass }}"

  tasks:
    - name: back up config
      vyos_config:
        backup: yes

- name: back up config from an IOS device
  hosts: cs01
  vars:
    ansible_password: "{{ ios_pass }}"

  tasks:
```

```
- name: back up config
  ios_config:
    backup: yes
```

Different Ansible modules are used to back up network devices that run different operating systems. The **ios_backup** module backs up the running configuration for devices that run IOS and the **vyos_backup** module backs up the configuration for devices that run VyOS. How do you make sure that one way of doing a task applies to one set of hosts and a different way is used when dealing with different hosts? One way is to use different plays.

This playbook illustrates the fact that a single playbook can hold multiple plays, each of which maps a particular set of tasks to a set of hosts.

- 3. Check the syntax of the playbook you created:

```
[student@workstation proj]$ ansible-playbook --syntax-check multi-vendor-
backup.yml
```

```
Playbook: multi-vendor-backup.yml
```

- 4. Use the **ansible-playbook** command to perform the plays in the **multi-vendor-backup.yml** playbook. Circumvent the restriction of using one group at a time by using the **--extra-vars** option to pass in the SSH passwords needed. You can provide any password at the SSH prompt; it is not used.

```
[student@workstation proj]$ ansible-playbook \
> -e 'vyos_pass=vyos ios_pass=student' multi-vendor-backup.yml
SSH password: anything

PLAY [back up config from a VyOS device] ****
TASK [back up config] ****
ok: [spine01]

PLAY [back up config from an IOS device] ****
TASK [back up config] ****
ok: [cs01]

PLAY RECAP ****
cs01 : ok=1    changed=0    unreachable=0   failed=0
spine01 : ok=1    changed=0    unreachable=0   failed=0
```

This concludes the guided exercise.

▶ Lab

Running Commands and Plays

In this lab, you will execute ad hoc commands and run plays.

Outcomes

You should be able to execute ad hoc commands and run plays in playbooks.

Before You Begin

This exercise assumes you know how to set the value of variables to support connection and authentication, as described in the *Lab: Running Commands and Plays* exercise.

Open a terminal window on the **workstation** VM and change to your **~/proj/** directory. The **~/proj/** directory should contain a suitable hosts inventory file. It should also include group variables files under **group_vars/** that set variables that play key roles in connecting and authenticating to remote hosts:

- **ansible_connection: network_cli** in **group_vars/network**
- **ansible_network_os: vyos** and **ansible_user: vyos** in **group_vars/vyos**
- **ansible_network_os: ios**, **ansible_user: admin** in **group_vars/ios**

The SSH password for the **cs01** device is **student**.

Instructions

1. Execute ad hoc commands to find out if the **utility** and **tower** machines are reachable from **cs01**. The IPv4 addresses of **utility** and **tower** are **172.25.250.8** and **172.25.250.9**. Hint: **cs01** is an IOS device, so investigate the **ios_ping** module.
2. Create a playbook in your Ansible project directory, named **iosping2.yml**, that does what you did with ad hoc commands in the previous step.
3. Run the play in your playbook using the **ansible-playbook** command. Provide **student** when prompted for a password.
4. Create a playbook that checks key system indicators, but for all devices on the network. What are “key system indicators?” They are the commands that you would use on the command line of a network device to check its health status. All of those can be automated as tasks within Ansible Playbooks. Most can probably be executed remotely as an Ansible ad hoc command. For convenience and as an illustration, it is suggested to display at least the host name, uptime, domain name, and system time. Use the **vyos_pass** and **ios_pass** extra command-line variables method, as shown earlier in this document, to set the **ansible_pass** variable to different values.
5. Run the play in your playbook using the **ansible-playbook** command.

This concludes the lab.

► Solution

Running Commands and Plays

In this lab, you will execute ad hoc commands and run plays.

Outcomes

You should be able to execute ad hoc commands and run plays in playbooks.

Before You Begin

This exercise assumes you know how to set the value of variables to support connection and authentication, as described in the *Lab: Running Commands and Plays* exercise.

Open a terminal window on the **workstation** VM and change to your **~/proj/** directory. The **~/proj/** directory should contain a suitable hosts inventory file. It should also include group variables files under **group_vars/** that set variables that play key roles in connecting and authenticating to remote hosts:

- **ansible_connection: network_cli** in **group_vars/network**
- **ansible_network_os: vyos** and **ansible_user: vyos** in **group_vars/vyos**
- **ansible_network_os: ios**, **ansible_user: admin** in **group_vars/ios**

The SSH password for the **cs01** device is **student**.

Instructions

1. Execute ad hoc commands to find out if the **utility** and **tower** machines are reachable from **cs01**. The IPv4 addresses of **utility** and **tower** are **172.25.250.8** and **172.25.250.9**. Hint: **cs01** is an IOS device, so investigate the **ios_ping** module.

```
[student@workstation ~]$ ansible -m ios_ping -a "dest=172.25.250.8" cs01
SSH password: student
...output omitted...
[student@workstation ~]$ ansible -m ios_ping -a "dest=172.25.250.9" cs01
SSH password: student
...output omitted...
```

2. Create a playbook in your Ansible project directory, named **iosping2.yml**, that does what you did with ad hoc commands in the previous step.

Create a file named **iosping2.yml** with content similar to the following:

```
---
- name: A reachability test
  hosts: cs01

  tasks:
    - name: Test reachability to 172.25.250.8
```

```

ios_ping:
  dest: 172.25.250.8

- name: Test reachability to 172.25.250.9
  ios_ping:
    dest: 172.25.250.9

```

3. Run the play in your playbook using the **ansible-playbook** command. Provide **student** when prompted for a password.

```
[student@workstation proj]$ ansible-playbook iosping2.yml
SSH password: student
...output omitted...
```

4. Create a playbook that checks key system indicators, but for all devices on the network. What are “key system indicators?” They are the commands that you would use on the command line of a network device to check its health status. All of those can be automated as tasks within Ansible Playbooks. Most can probably be executed remotely as an Ansible ad hoc command. For convenience and as an illustration, it is suggested to display at least the host name, uptime, domain name, and system time. Use the **vyos_pass** and **ios_pass** extra command-line variables method, as shown earlier in this document, to set the **ansible_pass** variable to different values.

Create a file named **multi-vendor-syscheck.yml** with content similar to the following:

```

---
- name: back up config and inspect health on vyos
hosts: vyos
vars:
  ansible_password: "{{ vyos_pass }}"

tasks:
  - name: backup config
    vyos_command:
      commands:
        - sh host name
        - sh system uptime
        - sh host domain
        - sh host date
        - sh host os
    register: results

  - name: show results
    debug:
      var: results.stdout

- name: back up config and inspect health on ios
hosts: ios
vars:
  ansible_password: "{{ ios_pass }}"

tasks:
  - name: backup config

```

```
ios_config:  
  backup: yes  
  
  - name: look at system elements  
    ios_command:  
      commands:  
        - sh ver | include uptime  
        - sh ip domain  
        - sh clock  
        - sh ip name-server  
    register: results  
  
  - name: show results  
    debug:  
      var: results.stdout
```

5. Run the play in your playbook using the **ansible-playbook** command.

```
[student@workstation ~]$ ansible-playbook \  
> -e 'vyos_pass=vyos ios_pass=student' multi-vendor-syscheck.yml  
SSH password: anything  
...output omitted...
```

This concludes the lab.

Chapter 3

Parameterizing Automation

Goal

Perform complex tasks with loops, variables, conditions, roles, and templates.

Objectives

- Define variables and use them in conditionals to control tasks.
- Parameterize playbooks and deploy customized files using Jinja2 templates.

Sections

- Defining Variables (and Guided Exercise)
- Controlling Tasks with Loops and Conditions (and Guided Exercise)
- Transforming Variable Data with Filters (and Guided Exercise)
- Working with Roles (and Guided Exercise)
- Customizing Data with Jinja2 Templates (and Guided Exercises)

Lab

Parameterizing Automation

Defining Variables

Objectives

After completing this section, you should be able to:

- Manage variables in Ansible projects.
- Explain variable precedence.

Naming Variables

Variable names must start with a letter and may only contain letters, numbers, and underscores.

The following are valid Ansible variable names:

- network
- Net7Work
- net_work

The following are *not* valid Ansible variable names:

- net-work
- 7network
- net@work

Defining Variables

Where and how variables are defined affects their scope and precedence.

Description	Scope
<i>Extra variables</i> , set on the command line: <code>-e name=value</code>	global
" <code>vars_files</code> " variables, set in the vars_files block in the play	play
<i>Play variables</i> , set in the vars block in the play	play
<i>Host facts</i> , typically gathered when play starts	host
<i>Host variables</i> , set in host_vars/host.yml	host
<i>Group variables</i> , set in group_vars/group.yml	host
<i>Role variables</i> , set in roles/rolename/defaults/main.yml	role

This is a simplified view of the relative precedence of different kinds of variables. There are many more types. Extra variables override **vars_files** variables, which override play variables, which override host facts, and so forth.

Using Variables in Playbooks

Ansible variable substitution uses the Jinja2 templating system. Variable names enclosed in curly braces are replaced during evaluation with values.

```
---
- name: trace the path from source (remote host) to trace_me
  hosts: cs01
  gather_facts: no

  vars:
    trace_me: 172.25.250.254

  tasks:
    - name: trace the path from {{ inventory_hostname }} to {{ trace_me }}
      ios_command:
        commands:
          - traceroute {{ trace_me }}
      register: path

    - name: show the path from {{ inventory_hostname }} to {{ trace_me }}
      debug:
        msg: "{{ path }}"
```

Note: A YAML/Jinja2 Quirk

Ordinarily, YAML syntax does not care whether strings are quoted or not. The double braces used by Jinja2, though, are similar enough to a YAML construct for notating mappings to confuse the YAML parser when the variable is the first element of a value.

```
debug:
  msg: {{ output_from_show_command }}
^ here
We could be wrong, but this one looks like it might be an issue with
missing quotes. Always quote template expression brackets when they
start a value. For instance:
with_items:
- {{ foo }}
Should be written as:
with_items:
- "{{ foo }}"
```

If the value starts with a brace, quote the string. Avoid unwarranted quoting, which could cause quotes to be interpreted as data.

Discovering Facts About Hosts

The connection modules used to connect to networking devices run Ansible's built-in, Python-based, fact-gathering system on the control node (localhost). You see that this is the case if you run an ad hoc command using the **setup** module: the data returned refers to the control node (localhost), not the network device.

```
[user@host ~]$ ansible -m setup network-device-host-inventory-identifier
```

When network devices are targeted by plays, turn off Ansible's built-in fact gathering. This disables fact gathering in a play:

```
- hosts: network-devices  
gather_facts: no
```

Facts are available for networking devices by way of platform specific ***os_facts** modules, such as **ios_facts**, **iosxr_facts**, **nxos_facts**, **vyos_facts**, **junos_facts**, and so forth.

Magic Variables

Ansible provides some predefined variables that are automatically set. Some of the better known magic variables are listed below. For the full list, see http://docs.ansible.com/ansible/playbooks_variables.html [http://docs.ansible.com/ansible/playbooks_variables.html]

Variable name	Description
hostvars	Contains the variables for managed hosts, and can be used to get the values for another managed host's variables. It does not include the managed host's facts if they have not been gathered yet for that host.
group_names	Lists all groups the current managed host is in.
groups	Lists all groups and hosts in the inventory.
inventory_hostname	Contains the host name for the current managed host as configured in the inventory. This may be different from the host name reported by facts for various reasons.

Introspection with Debug Module

The **debug** module prints messages while plays are executing. It can be useful for debugging variables or expressions.

```
- debug:  
msg: >  
  "{{ansible_net_hostname}} is {{ansible_net_model}}  
  running {{ansible_net_version}}"
```

The **debug** module could be used, for instance, to see values assumed by magic variables in a particular context in playbooks created for this purpose.

```
- debug:  
msg: "{{ role_names }} and {{ vars }}"
```

Registering Variables

The **register** keyword is used in the context of a playbook task. It saves the result of a task to a variable. Like facts, registered variables have host scope.

```
---
- name: >
  play for ios devices
  that uses commands
  to examine indicators
  hosts: ios
  gather_facts: no
  tasks:
    - name: look at indicators
      ios_command:
        commands:
          - sh ver | include uptime
          - sh ip domain
          - sh clock
          - sh ip name-server
          - sh proc mem | include Total
      register: results

    - name: show results
      debug:
        var: results.stdout
```

```
PLAY [play for ios devices that uses commands to examine indicators] ****
*****
TASK [look at indicators] ****
ok: [cs01]

TASK [show results] ****
ok: [cs01] => {
    "results.stdout": [
        "cs01 uptime is 3 hours, 12 minutes",
        "lab.example.com",
        "**01:09:01.866 UTC Sun Jun 3 2018",
        "255.255.255.255",
        "Processor Pool Total: 2202704640 Used: 250994304 Free:",
        "1951710336\n lsmpi_io Pool Total: 6295128 Used: 6294296 Free:",
        "832\n                                257260704 Total"
    ]
}

PLAY RECAP ****
cs01 : ok=2  changed=0  unreachable=0  failed=0
```

Read the Manual on *OS_FACTS

You are probably already familiar with the CLI for a given network OS, but ***os_facts** modules provide easy access to important facts. This example is for the **ios_facts** module, where interesting facts show up listed under # **hardware**.

```
$ ansible-doc ios_facts | sed -n '/^# hardware/,/^$/p'
# hardware
ansible_net_filesystems:
```

```

description: All file system names available on the device
returned: when hardware is configured
type: list
ansible_net_memfree_mb:
description: The available free memory on the remote device in Mb
returned: when hardware is configured
type: int
ansible_net_memtotal_mb:
description: The total memory on the remote device in Mb
returned: when hardware is configured
type: int

```

Facts Modules are Self-registering

The ***os_facts** modules handle variable registration for you. This example uses the **hardware** subset that you learned about from **ansible-doc ios_facts**.

```

---
- name: >
  play for ios devices
  that uses ios_facts
hosts: ios
gather_facts: no
tasks:
- name: look at some facts
  ios_facts:
    gather_subset:
      - hardware

- name: show results
  debug:
    msg:
      - "filesystems: {{ ansible_net_filesystems }}"
      - "free mem MB: {{ ansible_net_memfree_mb }}"
      - "mem total MB: {{ ansible_net_memtotal_mb }}"

```

```

PLAY [play for ios devices that uses ios_facts] ***

TASK [look at some facts] ****
ok: [cs01]

TASK [show results] ****
ok: [cs01] => {
  "msg": [
    "filesystems: [u'bootflash:']",
    "free mem MB: 1905270",
    "mem total MB: 2151078"
  ]
}

PLAY RECAP ****
cs01 : ok=2 changed=0 unreachable=0 failed=0

```

► Guided Exercise

Defining and Using Variables

In this exercise, you will define and use variables in a playbook.

Outcomes

You should be able to:

- Define variables in a playbook.
- Create various tasks that include defined variables.
- Execute an ad hoc command that displays output from the **setup** module when the connection method is **network_cli** and examine the results.
- Employ a Jinja2 template to generate magic variable data and examine the results.
- Create a playbook containing a play that uses a variable explicitly set using the **--extra-vars (-e)** option.
- Run the play you created using the **-e** option (**--extra-vars**).
- Review the various kinds of variables and how they are set.

Before You Begin

Open a terminal window on the **workstation** VM and change to your **~/proj/** directory.

Instructions

- 1. Create an **iosping3.yml** file by copying the **iosping2.yml** playbook you created in Lab2 and replacing the IP addresses with a pair of variables named **utility_ipv4** and **tower_ipv4**. These may be defined as **play variables** in a **vars** block at the top of a play. A variable that is set at the top of a play is a play variable. Play variables are defined (in scope) only within the play at the top of which they are set.

```
[student@workstation proj]$ cp iosping2.yml iosping3.yml
[student@workstation proj]$ cat iosping3.yml
---
- name: A reachability test
  hosts: cs01
  vars:
    utility_ipv4: 172.25.250.8
    tower_ipv4: 172.25.250.9

  tasks:
    - name: "Test reachability to utility: {{ utility_ipv4 }}"
      ios_ping:
        dest: "{{ utility_ipv4 }}"
```

```
- name: "Test reachability to tower: {{ tower_ipv4 }}"
  ios_ping:
    dest: "{{ tower_ipv4 }}"
```

- 2. Verify the syntax of the playbook and run the play. The IOS device password is **student**.

```
[student@workstation proj]$ ansible-playbook --syntax-check iosping3.yml
[student@workstation proj]$ ansible-playbook iosping3.yml
```

- 3. What happens when a variable is defined as a play variable (that is, set in a **vars** block at the top of a play) and we set the value of the same variable on the command line?

```
[student@workstation proj]$ ansible-playbook \
> iosping3.yml -e 'utility_ipv4=172.25.250.254'
SSH password: student

PLAY [A reachability test] ****
TASK [Test reachability to utility: 172.25.250.254] ****
ok: [cs01]

TASK [Test reachability to tower: 172.25.250.9] ****
ok: [cs01]

PLAY RECAP ****
cs01 : ok=2    changed=0    unreachable=0    failed=0
```

It does the right thing by treating the value set in the playbook file as a default and overriding it with the value specified in the command line with the **-e** option.

- 4. Create an **iosping4.yml** playbook that has one play, and defines the variable **dest** as a play variable set to **172.25.250.9**. Use the internal variable **inventory_hostname** to provide more information when the play runs. The resulting playbook should have the following contents:

```
---
- name: A reachability test
  hosts: cs01
  vars:
    dest: 172.25.250.9

  tasks:
    - name: "test reachability from {{ inventory_hostname }} to {{ dest }}"
      ios_ping:
        dest: "{{ dest }}"
```

- ▶ 5. Verify the syntax and run the play at least twice. The first time, let it use the default value. Then run it again and set the value of **dest** using the **-e** command-line option:

```
[student@workstation proj]$ ansible-playbook --syntax-check iosping4.yml  
[student@workstation proj]$ ansible-playbook iosping4.yml  
[student@workstation proj]$ ansible-playbook iosping4.yml -e 'dest=172.25.250.254'
```

- ▶ 6. Create an **iosping5.yml** playbook that sets the value of **dest** using **vars_file** instead of the **vars** block. The new playbook will perform the same task as the previous one.

The point is to demonstrate two of the many ways in which variables can be defined: either within the playbook itself in a **vars** block at the top of a play (as you did when you defined **dest** in **iosping4.yml**), or loaded from a file using **vars_files** (as you will see with **iosping5.yml**, using **vars/myvars.yml**).

There are valid use cases for doing it both ways. Define it in the playbook for simplicity, when you want everything contained in a single file. Define data using **vars** files when so much data must be defined that it seems awkward or unwieldy to put it in the playbook.

- 6.1. Create a **vars/** directory:

```
[student@workstation proj]$ mkdir vars
```

- 6.2. Create a text file in this directory named **myvars.yml** and set the value of **dest** in this file:

```
[student@workstation proj]$ echo 'dest: 172.25.250.9' > vars/myvars.yml
```

- 6.3. Copy **iosping4.yml** to **iosping5.yml**:

```
[student@workstation proj]$ cp iosping4.yml iosping5.yml
```

- 6.4. Replace the **vars** block with a **vars_files** statement. The file should appear as follows:

```
---  
- name: A reachability test  
  hosts: cs01  
  vars_files:  
    - vars/myvars.yml  
  
  tasks:  
  
    - name: "test reachability from {{ inventory_hostname }} to {{ dest }}"  
      ios_ping:  
        dest: "{{ dest }}"
```

- 6.5. Check the syntax, then run the play. Test it both with the default value and using the **-e** option to set the value of **dest** to something different.

- 6.5.1. Without using the **--extra-vars (-e)** option:

```
[student@workstation proj]$ ansible-playbook iosping5.yml
SSH password: student

PLAY [A reachability test] ****
TASK [Test reachability from cs01 to 172.25.250.9] ****
ok: [cs01]

PLAY RECAP ****
cs01 : ok=1    changed=0    unreachable=0    failed=0
```

6.5.2. Using the **--extra-vars (-e)** option to set **dest**:

```
[student@workstation proj]$ ansible-playbook \
> -e 'dest=172.25.250.8' iosping5.yml
SSH password: student

PLAY [A reachability test] ****
TASK [Test reachability from cs01 to 172.25.250.8] ****
ok: [cs01]

PLAY RECAP ****
cs01 : ok=1    changed=0    unreachable=0    failed=0
```

- 7. Execute an ad hoc command that displays output from the **setup** module when the connection method is **network_cli** and examine the results. Type **ansible -m setup spine01 | more** and browse through the data returned. You can use **ansible-doc setup** to learn more about the **setup** module, or visit the **setup** module page at the Ansible documentation website. The connection method for **spine01** should be set to **network_cli**. Notice that when you browse through the data, all or nearly all of it is about the local host, not **spine01**. Remember that the password for VyOS devices is **vyos**.

When people are seeking a way to display all of the magic variables and facts about a host that are available, they are often directed to the **setup** module. This exercise demonstrates that the **setup** module, when used in conjunction with network devices, returns facts pertaining to the local host (the control node), not the network device. You must use a module from the ***os_facts** family to gather facts from network devices. For VyOS devices, for instance, use the **vyos_facts** module.

- 8. Employ a Jinja2 template to generate magic variable data and examine the results.

- 8.1. Create a **j2** subdirectory if it does not already exist:

```
[student@workstation proj]$ mkdir j2
```

- 8.2. Create a Jinja2 template named **j2/magic.j2** that contains the following content:

```
hostvars:
{{ hostvars | to_nice_yaml }}
```

```
groups:  
  {{ groups | to_nice_yaml }}  
  
group_names:  
  {{ group_names | to_nice_yaml }}  
  
inventory_hostname:  
  {{ inventory_hostname | to_nice_yaml }}  
  
play_hosts:  
  {{ play_hosts | to_nice_yaml }}
```

- 8.3. Create a playbook, named **magic.yml**, that uses the **template** module with the **j2/magic.j2** template to write a file named **magic.out**. The **magic.yml** content should be similar to the following:

```
---  
- name: write magic variables to a file using a template  
  hosts: cs01  
  
  tasks:  
    - name: do it  
      template:  
        src: j2/magic.j2  
        dest: magic.out
```

- 8.4. Execute the **ansible-playbook** command to run the play in the **magic.yml** playbook. The SSH password for **cs01** is **student**.

```
[student@workstation proj]$ ansible-playbook magic.yml  
SSH password: student  
  
PLAY [write magic variables to a file using a template] *****  
  
TASK [do it] *****  
changed: [cs01]  
  
PLAY RECAP *****  
cs01 : ok=1    changed=1    unreachable=0    failed=0
```

- 8.5. Examine the contents of the **magic.out** file. A range of data is available by way of these preexisting variables.

```
[student@workstation proj]$ less magic.out
```

- ▶ 9. Create a playbook containing a play that uses a variable explicitly set using the **--extra-vars (-e)** option.

- 9.1. Create a playbook named **ios-tracert1.yml** that performs the **traceroute** IOS command using an extra variable named **dest**. Include the following content in your playbook:

```
---
- name: >
  illustrate the use of the
  --extra-vars (-e) option by
  using the ios_command module
  and running the traceroute command
  when destination is {{ dest }}
hosts: cs01

tasks:

- name: traceroute to dest
  ios_command:
    commands:
      - traceroute {{ dest }} probe 2 timeout 2
  register: result

- name: show result
  debug:
    var: result.stdout_lines
```

- 9.2. Run the play you created using the **-e** option (**--extra-vars**).

```
[student@workstation proj]$ ansible-playbook -e 'dest=172.25.250.254' ios-
tracert1.yml
SSH password: student

PLAY [illustrate the use of the --extra-vars (-e) option by using the ios_command
module by running the traceroute command when destination is 172.25.250.254] ***

TASK [traceroute to 172.25.250.254] ****
ok: [cs01]

TASK [show result] ****
ok: [cs01] => {
    "result.stdout_lines": [
        [
            "Type escape sequence to abort.",
            "Tracing the route to 172.25.250.254",
            "VRF info: (vrf in name/id, vrf out name/id)",
            "  1 172.25.250.254 3 msec 2 msec"
        ]
    ]
}

PLAY RECAP ****
cs01 : ok=2    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

Controlling Tasks with Loops and Conditions

Objectives

After completing this section, you should be able to:

- Implement iteration (loops) in plays to apply tasks to collections of items.
- Build conditions in plays to control when tasks perform actions.
- Customize how plays respond to error codes using conditions.
- Combine conditions and loops to create tasks that are both powerful and flexible.

Simple Conditions

The **when** statement is used to run a task only when a condition is satisfied. A simple condition is whether a Boolean variable is true or false. The **when** statement in the example below causes the task to run only if the **show_interfaces** variable is true:

The indentation of the **when** statement matters. The **when** statement is not a module variable, so it must be outside the module at the top level of the task.

A task is a YAML mapping, and the **when** statement is a key in the task, like the task's name and the module it uses. A common convention places **when** statements after the task's name and the module (and module arguments).

```
---
- name: illustrating a simple condition
  hosts: cs01
  vars:
    show_interfaces: True

  tasks:
    - name: conditionally issue command
      vyos_command:
        commands:
          - show interfaces
      register: command_output
      when: show_interfaces == True

    - name: show command output
      debug:
        msg: "{{ command_output }}"
      when: show_interfaces == True
```

Compound when Statements

A single, compound **when** statement can evaluate multiple conditions. Use the keywords **and** or **or** to form a compound **when** statement. Multiple conditions that all need to be true (a logical 'and') can also be specified as a list, as in this example.

```

---
- name: a play for ios devices that uses ios_facts
  hosts: ios
  gather_facts: no

  tasks:
    - name: look at some facts
      ios_facts:
        gather_subset:
          - hardware

  - name: show results
    debug:
      msg:
        - "model: {{ ansible_net_model }}"
        - "version: {{ ansible_net_version }}"

  when:
    - ansible_net_model == 'CSR1000V'
    - ansible_net_version is version('03.14', '>=')

```

```

PLAY [a play for ios devices that uses ios_facts] ***

TASK [look at some facts] ****
ok: [cs01]

TASK [show results] ****
ok: [cs01] => {
  "msg": [
    "model: CSR1000V",
    "version: 03.14.01.S"
  ]
}

PLAY RECAP ****
cs01    : ok=2 changed=0  unreachable=0  failed=0

```

Programmatic Style

Conditionals are a construct found in programming languages, and so it is tempting to use conditionals in Ansible in a similar way, as in the example below:

```

---
- hosts: spines

  tasks:
    - name: build list of interfaces
      vyos_command:
        commands: show interfaces | grep eth | cut -d' ' -f1
      register: interfaces

    - name: set eth0 description
      vyos_config:

```

```

lines:
  - set interface ethernet eth0 description 'Outside'
loop: "{{ interfaces.stdout_lines }}"
when: item == "eth0"

```

Ansible Style

When you find yourself using the **when** statement, consider whether there might be a simpler alternative. There might be a way to make use of host groups, for instance, to limit the scope of tasks. The condition might not be necessary.

```

---
- name: configure interface descriptions on spine devices
  hosts: spines

  tasks:
    - name: configure a spine device
      vyos_config:
        lines:
          - set interface ethernet eth0 description 'Outside'
          - set interface ethernet eth1 description 'Inside'

```

Applying Tasks to Objects

Ansible provides many directives that serve to control and direct repeated application of a task to individual objects in the context of collections of objects. This is a sample.

Directive	Description
loop	Iterates a task over a list of items. List elements may be mappings.
with_nested	Takes an outer list of two or more inner lists and runs a loop that acts upon an array of items. The array elements are referred to as item.0 , item.1 , and so forth.
with_sequence	Generates a sequence of items in increasing numerical order. Can take start and end arguments that have a decimal, octal, or hexadecimal integer value.
with_dict	Takes a hash (a dictionary) and loops through its elements, returning item.key and item.value .

Simple Loops

A simple loop uses the **loop** directive to iterate a task over a list of items.

```

- name: issue show interface command for each interface in list
  hosts: cs01
  gather_facts: no
  vars:
    interface_list: [ GigabitEthernet1, GigabitEthernet2 ]

  tasks:
    - name: show each interface

```

```

ios_command:
  commands:
    - show interface {{ item }}
register: results
loop: "{{ interface_list }}"

- name: show results
  debug:
    msg: "{{ results }}"

```

Nested Loops

The **with_nested** directive causes the task to iterate over a list of lists. This provides access within the body of the task to an array of items, one from each list.

```

---
- name: illustrating nested loops
  hosts: spine01
  gather_facts: no
  vars:
    source_interfaces: [ eth0, eth1 ]
    dest_addresses: [ 172.25.250.254, 172.25.250.195 ]

  tasks:
    - name: with each destination, source ping from each interface
      vyos_command:
        commands:
          - ping {{ item.0 }} interface {{ item.1 }} count 2
      register: ping_output
      with_nested:
        - "{{ dest_addresses }}"
        - "{{ source_interfaces }}"

    - name: show ping_output
      debug:
        msg: "{{ ping_output }}"

```

Looping Over Dict With Conditions

Display the target interface, but only if it is UP.

```

---
- name: >
  show us the target interface
  but only if it is UP

  hosts: ios
  gather_facts: no
  vars:
    target_interface: GigabitEthernet4

  tasks:

```

```

- name: gather facts
  ios_facts:
    gather_subset: all
    when: ansible_network_os == 'ios'

- name: display facts
  # 'ansible_net_interfaces' fact is a dict of interfaces
  # with interface name as key
  debug:
    msg: "{{ item }}"
  with_dict: "{{ ansible_net_interfaces }}"
  when:
    # Multiple conditions that all need to be true
    # (a logical 'and') can be specified as a list
    - item.key == target_interface
    - item.value['operstatus'] == 'up'

```

```

PLAY [show us the target interface but only if it is UP] *****

TASK [debug] *****
ok: [cs01] => (item=None) => {
  "msg": {
    "key": "GigabitEthernet4",
    "value": {
      "bandwidth": 1000000,
      "description": null,
      "duplex": "Full",
      "ipv4": [
        {
          "address": "172.16.10.1",
          "subnet": "30"
        }
      ],
      "ipv6": [
        {
          "address": "FDFB:EDE0:BDF2:C094::1",
          "subnet": "FDFB:EDE0:BDF2:C094::/64"
        }
      ],
      "lineprotocol": "up",
      "macaddress": "2cc2.600c.e636",
      "mediatype": "RJ45",
      "mtu": 1500,
      "operstatus": "up",
      "type": "CSR vNIC"
    }
  }
}

PLAY RECAP *****
cs01 : ok=2  changed=0  unreachable=0  failed=0

```

Normal Task Processing

A nonzero command exit code fails its task; it skips the remaining tasks for *this host*.

```
---
- name: >
  an illustration of
  normal task processing

hosts: ios
gather_facts: no
vars:
  intf: Interface GigabitEthernet16

tasks:

- name: a failed task
  # this task fails because
  # no such interface exists
  ios_config:
    lines:
      - shutdown
  parents: "{{ intf }}"

- name: a good task
  debug:
    msg: "this is a good task"
```

```
PLAY [an illustration of normal task processing] ****

TASK [a failed task] ****
An exception occurred during task execution. To see the full traceback, use -
vvv. The error was: cs01(config)#
fatal: [cs01]: FAILED! => {"changed": false, "module_stderr": "Traceback
(most recent call last):\n  File \"/tmp/ansible_Rz9r3l/
ansible_module_ios_config.py\", line 583, in <module>\n      main()\n  File \"/tmp/ansible_Rz9r3l/ansible_module_ios_config.py\", line 512, in
main\n      load_config(module, commands)\n  File \"/tmp/ansible_Rz9r3l/
ansible_modlib.zip/ansible/module_utils/network/ios/ios.py\", line 162, in
load_config\n  File \"/tmp/ansible_Rz9r3l/ansible_modlib.zip/ansible/
module_utils/connection.py\", line 149, in
__rpc__\n    ansible.module_utils.connection.ConnectionError: Interface
GigabitEthernet16\r\n                                          ^\r\n% Invalid input
detected at '^' marker.\r\n  ncs01(config)#\n", "module_stdout": "", "msg":
"MODULE FAILURE", "rc": 1}!
    to retry, use: --limit @/home/student/tmp/chgxpl/ansible-for-
networkautomation/
l2/ios-normal-error-handling.retry

PLAY RECAP ****
cs01                  : ok=0  changed=0  unreachable=0  failed=1
```

The rest of the tasks in the play are skipped for this host because the first task failed.

Ignoring Errors on a Task

```
---
- name: >
  an illustration of
  ignoring errors on a task

  hosts: ios
  gather_facts: no
  vars:
    intf: Interface GigabitEthernet16

  tasks:

    - name: a failed task
      # this task fails because
      # no such interface exists
      ios_config:
        lines:
          - shutdown
        parents: "{{ intf }}"
      ignore_errors: yes

    - name: a good task
      debug:
        msg: "this is a good task"
```

```
PLAY [an illustration of normal task processing] ****
TASK [a failed task] ****
An exception occurred during task execution. To see the full traceback, use -vvv. The error was: cs01(config)#
fatal: [cs01]: FAILED! => {"changed": false, "module_stderr": "Traceback (most recent call last):\n  File \"/tmp/ansible_02hpnL/ansible_module_ios_config.py\", line 583, in <module>\n    main()\n  File \"/tmp/ansible_02hpnL/ansible_module_ios_config.py\", line 512, in\n    main\n      load_config(module, commands)\n  File \"/tmp/ansible_02hpnL/ansible_modlib.zip/ansible/module_utils/network/ios/ios.py\", line 162, in\n    load_config\n  File \"/tmp/ansible_02hpnL/ansible_modlib.zip/ansible/module_utils/connection.py\", line 149, in\n    __rpc__\n      nansible.module_utils.connection.ConnectionError: Interface\n      GigabitEthernet16\r\n                                         ^\r\n                                         Invalid input\n      detected at '^' marker.\r\n      nncs01(config)#\n      n      \"module_stdout\": \"\", \"msg\":\n      n      \"MODULE FAILURE\", \"rc\": 1}\n      n      ...ignoring\n\nTASK [a good task] ****
ok: [cs01] => {
      "msg": "this is a good task"
}
PLAY RECAP ****
cs01 : ok=2  changed=0  unreachable=0  failed=0
```

It does not matter if the play fails to shut down a nonexistent interface, so ignore the error.

► Guided Exercise

Controlling Tasks with Loops and Conditions

In this exercise, you will construct loops and conditionals in Ansible Playbooks.

Outcomes

You should be able to:

- Use an Ansible **loop** in conjunction with conditionals.
- Use the **when** statement to control the flow of task execution in plays.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/proj/` directory.

Instructions

- 1. Create a file named **iosping6.yml** with a single task that loops over a variable named **interesting_destinations**, consisting of a list of IPv4 addresses. The resulting playbook should appear similar to the following:

```
---
- name: A reachability test
hosts: cs01
vars:
  interesting_destinations:
    - 172.25.250.8
    - 172.25.250.9

tasks:
  - name: "test reachability from {{ inventory_hostname }}"
    ios_ping:
      dest: "{{ item }}"
    loop: "{{ interesting_destinations }}"
```

- 2. Verify the syntax and run the play:

```
[student@workstation proj]$ ansible-playbook --syntax-check iosping6.yml
[student@workstation proj]$ ansible-playbook iosping6.yml
```

The required SSH password is **student**.

- 3. The **when** statement can be used to consolidate plays in multivendor playbooks. To keep things relatively simple, we look at how this works with a playbook designed to deal with a single target host at a time.

Create a playbook named **multi-vendor-syscheck2.yml**. The individual tasks are similar to your **multivendor-syscheck.yml** playbook. Modify it so that it contains a single play, and sets the value of hosts based on an extra variable named target. Conditionally run the tasks based on the value of the **ansible_network_os** variable.

```
[student@workstation proj]$ cat multi-vendor-syscheck2.yml
- name: back up config and record device health indicators
  hosts: network

  tasks:
    - name: backup vyos config
      vyos_config:
        backup: yes
      when: ansible_network_os == "vyos"

    - name: look at vyos device health indicators
      vyos_command:
        commands:
          - sh host name
          - sh system uptime
          - sh host domain
          - sh host date
          - sh host os
          - sh sys mem
      register: vyos_result
      when: ansible_network_os == "vyos"

    - name: backup ios config
      ios_config:
        backup: yes
      when: ansible_network_os == "ios"

    - name: look at ios device health indicators
      ios_command:
        commands:
          - sh ver | include uptime
          - sh ip domain
          - sh clock
          - sh ip name-server
          - sh proc mem platform | include System memory
      register: ios_result
      when: ansible_network_os == "ios"

    - name: show results
      debug:
        msg: "{{ item }}"
      loop: "{{ (vyos_result | combine(ios_result)).stdout_lines }}"
```

- 4. Run the new playbook, using the limit option (**-l SUBSET**) as shown here:

```
[student@workstation proj]$ ansible-playbook -l cs01 multi-vendor-syscheck2.yml
```

Or like this:

```
[student@workstation proj]$ ansible-playbook -l vyos multi-vendor-syscheck2.yml
```



Note

Since you have removed the password **extra-vars** from this version, you must provide the appropriate password when running the playbook. Use the **vyos** password for the **spine** and **leaf** VyOS devices. Use the **student** password for the IOS device, **cs01**.

This concludes the guided exercise.

Transforming Variable Data with Filters

Objectives

After completing this section, you should be able to:

- Reformat the data in a variable with a Jinja2 filter.
- Run a task conditionally based on the result of a Jinja2 test.

Filtering Variables and Data

Filters in Ansible are from Jinja2, and are used for transforming data inside a template expression.

Filters are available for a wide variety of use cases.

For example, to transform the CIDR form of an IPv4 address into a host address, you can use the `ipaddr` filter:

```
{{ '192.0.2.1/24' | ipaddr('address') }}
```

This returns the value **192.0.2.1**.

You can use the same Jinja2 filter with a different argument to extract the variable-length subnet mask from that specification:

```
{{ '192.0.2.1/24' | ipaddr('netmask') }}
```

This returns the value **255.255.255.0**. You can use the `prefix` argument with the `ipaddr` filter if you want the CIDR prefix for that subnet mask instead.

Parsing Command Output

A network CLI filter is available that, when used in conjunction with a parser specification, automatically parses a network device CLI command output into structured JSON form.

```
---
- name: a play that shows parsed interface data
  hosts: ios
  gather_facts: no
  vars:
    shint_parser: vars/ios-shipintbr-parser.yml

  tasks:
    - name: issue commands
      ios_command:
        commands:
          - show ip interface brief
      register: results
```

```
- name: show CLI output
  debug:
    msg: "{{ results.stdout[0] | parse_cli(shint_parser) }}"
```

Running Tasks Conditionally by Testing Variables

Jinja2 tests evaluate template expressions and return a True or False result.

You can use these expressions to conditionally run tasks. The **when** directive only runs a task if its condition or list of conditions are True.

```
tasks:
  - name: run a command
    vyos_command:
      commands:
        - show interface ethernet eth2
    register: result
    ignore_errors: True

  - debug:
      msg: "show interface ethernet eth2 failed, interface might not be present"
      when: result is failed
```

In this example, the expression **result is failed** is a Jinja2 test.

► Guided Exercise

Looping Over a Filtered List

In this exercise, you will construct loops and conditionals in Ansible Playbooks.

Outcomes

You should be able to use a filter to loop over items selected from a list.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/proj/` directory.

Instructions

- ▶ 1. Add an **ipv4_addresses** list variable to your **vars/myvars.yml** file:

```
dest: 172.25.250.9
ipv4_addresses:
  - 172.25.250.8
  - 172.25.250.9
  - 172.25.250.51
  - 172.25.250.61
  - 172.25.250.151
  - 172.25.250.161
  - 172.25.250.195
  - 8.8.8.8
```

- ▶ 2. Create a playbook named **iosping7.yml** that uses this file for data, but disregards addresses not associated with the **172.25.250.0/24** subnet. You saw how to use **vars_files** to separate data from code. That makes it possible to maintain the data independently. The available data, however, might be broader in scope than the present task requires. A list of IP addresses, for instance, might contain many addresses that are of no interest with respect to the task at hand. Your playbook should contain the following:

```
---
- name: A reachability test
hosts: cs01
vars_files:
  - vars/myvars.yml

tasks:

- name: "test reachability from {{ inventory_hostname }} to {{ dest }}"
  ios_ping:
    dest: "{{ item }}"
  loop: "{{ ipv4_addresses | select('match', '^172\\.25\\.250\\..*') | list }}"
  when: ansible_network_os == 'ios'
```

- 3. Verify the syntax and run the play:

```
[student@workstation proj]$ ansible-playbook iosping7.yml
SSH password: student

PLAY [A reachability test] ****
TASK [Test reachability from cs01 to 172.25.250.9] ****
ok: [cs01] => (item=172.25.250.8)
ok: [cs01] => (item=172.25.250.9)
ok: [cs01] => (item=172.25.250.51)
ok: [cs01] => (item=172.25.250.61)
ok: [cs01] => (item=172.25.250.151)
ok: [cs01] => (item=172.25.250.161)
ok: [cs01] => (item=172.25.250.195)

PLAY RECAP ****
cs01 : ok=1    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

Working with Roles

Objectives

After completing this section, you should be able to:

- Describe the structure of a role.
- Create a role.
- Execute a play that uses one or more roles.

Creating Reusable Playbooks

You can write a playbook in one very large file (and you might start out learning playbooks this way), but eventually you will want to reuse files and start to organize things. In Ansible, there are three ways to do this: includes, imports, and roles.

What are imports and includes? With Ansible 2.5 you can:

- Import playbooks.
- Include or import task files.
- Include or import roles.

What is the difference between including and importing?

- Import statements are processed when the playbooks are parsed.
- Include statements are processed as they are encountered during the execution of the playbook.

Organizing Playbooks with Roles

You can use roles as an effective way to organize your playbooks.

- Roles allow you to make your Ansible code more reusable, by saving tasks, template files, and variables in a generic way that can be reused by different playbooks and people.
- Roles allows you to better separate the procedure used to configure a device (in the role) from the details of the configuration that is applied (in the play, inventory, and variables).
- A well-written role can be shared safely with others without exposing sensitive information about your configuration.

Installing Roles

Here are two ways of installing roles:

- Provide a **roles/requirements.yml** file that contains links to SCM-based roles. These roles are loaded automatically. This is a highly scalable way to install roles.
- Create roles locally by:
 - Using **ansible-galaxy init** to initialize a directory structure.

- Populating the directory structure with files that implement the role.

Automating Role Installation

A sample **requirements.yml** file is described below.

```
# sample requirements.yml

- src: ssh://git@example.com:8989/ansible-role-network-spine.git
  name: network-spine
  scm: git

- src: ssh://git@example.com:8989/ansible-role-network-leaf.git
  name: network-leaf
  scm: git
```

The **requirements.yml** file controls how roles are located and loaded.

Initializing Roles Locally

Use the command **ansible-galaxy init rolename** to initialize a role named *rolename*.

The **ansible-galaxy init** command does not create the roles directory.

Use the following command sequence to initialize a role skeleton for a role named **myrole**.

```
$ mkdir roles && ansible-galaxy init roles/myrole
```

These commands generate the following directory structure:

```
└── roles
    └── myrole
        ├── defaults
        │   └── main.yml
        ├── files
        ├── handlers
        │   └── main.yml
        ├── meta
        │   └── main.yml
        ├── README.md
        ├── tasks
        │   └── main.yml
        ├── templates
        ├── tests
        │   └── inventory
        │       └── test.yml
        └── vars
            └── main.yml
```

Identifying Role Subdirectories

```
roles/
  rolename/           # this hierarchy represents a "role"
    defaults/
      main.yml        # <-- default lower priority variables for this role
    files/
      bar.txt         # <-- files for use with the copy resource
      foo.sh          # <-- script files for use with the script resource
    handlers/
      main.yml        # <-- handlers file
    meta/
      main.yml        # <-- role dependencies
    tasks/
      main.yml        # <-- tasks file can include smaller files
    templates/
      ntp.conf.j2     # <-- templates end in .j2
    vars/
      main.yml        # <-- variables associated with this role
```

Always provide content in the **meta/main.yml** file.

Identifying Your Role

Two examples of what **meta/main.yml** might look like are shown below. Always fill in the **author** and **description** values.

```
---
galaxy_info:
  author: Bob
  description: Ansible role to deploy myAcmeApp
  company: Bob Inc.
  license: MIT
  min_ansible_version: 2.2
  platforms:
    - name: EL
      versions:
        - all
  categories:
    - cloud
    - web
dependencies: []
```

```
dependencies: []
galaxy_info:
  author: Alice
  company: Alice Inc.
  description: Nethosts, resolver and interfaces
  license: MIT
  min_ansible_version: 2.2.0
  platforms:
    - name: Debian
      versions:
```

```

    - wheezy
    - jessie
    - stretch
categories:
  - system
  - network

```

The main justification for roles is reusability. Failing to provide role metadata severely impairs reusability.

Roles and Variables

Variables can be set on the command line, within the inventory file, within inventory-level **group_vars** and **host_vars** files, within role files, and default or variable files at the role level.

More commonly used ones are shown in boldface. Listed in order of precedence, from low to high:

- role defaults
- inventory file or script group vars
- inventory group_vars/all
- **playbook group_vars/all**
- inventory group_vars/*
- **playbook group_vars/***
- inventory file or script host vars
- inventory host_vars/*
- **playbook host_vars/***
- **host facts**
- **play vars**
- play vars_prompt
- **play vars_files**
- role vars (defined in role/vars/main.yml)
- block vars (only for tasks in block)
- task vars (only for the task)
- role (and include_role) params
- include params
- include_vars
- **set_facts / registered vars**
- **extra vars** (always win precedence)

Familiarize yourself with a few of the more commonly used types of variables and stick with those unless you come across special requirements.

Creating an Ansible Project

When roles are used in conjunction with an Ansible project, put configuration files and playbooks in the project root with roles in a **roles** directory.

```

└── acme-network
    ├── ansible.cfg
    ├── leafs.yml
    └── roles
        ├── common
        ├── leaf
        └── spine
    └── spines.yml

```

This makes it easy to manage the project directory with a VCS such as Git. It can be deployed to any Ansible control node that holds a host inventory locally, and used there with little or no extra operational overhead.

Incorporating Roles into Playbooks

There are two conventions for incorporating roles into playbooks: classic and modern.

The classic way of using roles is with the **roles** directive:

```

---
- hosts: ourhosts
  roles:
    - common
    - role1
    - role2

```

You can also use roles inline with any other tasks using **import_role** or **include_role**:

```

---
- hosts: ourhosts
  tasks:
    - debug:
        msg: "before we run our role"
    - import_role:
        name: role1
    - include_role:
        name: role2
    - debug:
        msg: "after we ran our role"

```

Adapting Playbooks to Use Roles

It is easy to adapt playbooks that do not use roles into ones that do, as shown below:

1. If some sequence of tasks applies to all hosts, create a role named "common" and move those tasks into it.
2. Identify sequences of tasks to be applied to all hosts that play a particular role.

3. Optionally, decompose further by grouping tasks together that work toward a particular end result (a goal or subgoal).
4. With each such sequence of tasks, create a role and move the tasks into it.
5. Adapt the playbook, or create a new playbook, using the original hosts block and replacing the original tasks block with a roles block.

An Example of Adapting a Playbook

An example is provided here that illustrates how to adapt playbooks to use roles.

Create a **spine** role (**roles/spine/tasks/main.yml**) and a playbook that uses it (**spines.yml**).

```
---
- name: configure interface descriptions on spine devices
  hosts: spines

  tasks:
    - name: set interface description
      vyos_interface:
        aggregate:
          - { name: eth0, description: Outside }
          - { name: eth1, description: Inside }
```

Given this simple playbook that does not use roles:

```
---
- name: the spine role
  tasks:
    - name: set interface description
      vyos_interface:
        aggregate:
          - { name: eth0, description: Outside }
          - { name: eth1, description: Inside }
```

```
---
- name: the spines playbook
  hosts: spines
  roles:
    - spine
```

Role Dependencies

Roles can include other roles as dependencies. Dependencies are defined in the **meta/main.yml** file. A role, for instance, that configures an IOS device to support SSH access might also require a **vty_lines** role.

```
# meta/main.yml

dependencies:
  - { role: ssh_access, ssh_version: 2 }!
  - { role: vty_lines, range_start: 0, range_end: 4 }
```

By default, a given role is only added as a dependency to a playbook once. This can be overridden by setting the **allow_duplicates** variable to **yes** in the **meta/main.yml** file.

Order of Execution

When playbooks contain both roles and non-role tasks, tasks in roles normally run before non-role ones. Ansible also lets you override this with the special **pre_tasks** and **post_tasks** sections. You can define one sequence of non-role tasks that execute before roles, and another sequence that executes after roles.

```
---
- hosts: somehost.example.com
  pre_tasks:
    - debug:
        msg: 'before'

  roles:
    - role1
    - role2

  tasks:
    - debug:
        msg: 'implicitly after'

  post_tasks:
    - debug:
        msg: 'explicitly after'
```

In this example, even if the **tasks** block preceded the **roles** block, the "implicitly after" debug message would still run *after* the roles (with roles being called in the classic way).

► Guided Exercise

Creating and Using Roles

In this exercise, you will create and use a role.

Outcomes

You should be able to:

- Define data using a variable named **interfaces**.
- Compose a play that uses the **interfaces** variable to set interface descriptions for members of the **cloud-services** group.
- Convert the original play into a role.
- Create a new play that uses the role.
- Verify that the new play works as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/proj/` directory.

Instructions

► 1. Define data using variables.

Create an **interfaces** variable, associated with the **cloud-services** group. The `group_vars/cloud-services` file should have the following contents:

```
interfaces:
- { name: GigabitEthernet1, description: Management }
- { name: GigabitEthernet2, description: Outside Primary }
- { name: GigabitEthernet3, description: Outside Secondary }
- { name: GigabitEthernet4, description: Server Link1 }
```

► 2. Compose a play that sets interface descriptions for members of the **cloud-services** group.

2.1. Create a playbook named `cs-ifdescr.yml` for members of the **cloud-services** group, containing a play that sets interface descriptions. All members of the **cloud-services** group happen to be IOS devices.

```
---
- name: set interface descriptions for cloud-services devices
hosts: cloud-services
# interface description data for this group
# is set in its group variables file
tasks:
- name: set interface description
```

```

ios_interface:
  aggregate: "{{ interfaces }}"

  - name: get interface descriptions
    ios_command:
      commands:
        - show interfaces | include Description
    register: results

  - name: show results
    debug:
      var: results.stdout

```

2.2. Test the play to determine if it does what it is intended to do:

```

[student@workstation proj]$ ansible-playbook --syntax-check cs-ifdescr.yml

playbook: cs-ifdescr.yml
[student@workstation proj]$ ansible-playbook --check cs-ifdescr.yml
SSH password: student

PLAY [set interface descriptions for cloud-services devices] ****
TASK [set interface description] ****
changed: [cs01]

TASK [get interface descriptions] ****
ok: [cs01]

TASK [show results] ****
ok: [cs01] => {
  "results.stdout": [
    ""
  ]
}

PLAY RECAP ****
cs01 : ok=3    changed=1    unreachable=0    failed=0

```

No results are shown because the `--check` option does not make any changes. It predicts some of the changes that may occur when the playbook is executed.

▶ 3. Convert the original play into a role.

Why take the trouble to convert a working playbook into a role? Here is a partial list of reasons:

- Modularity in service of reusability.
- Converting something that is useful and has value in a specific context (being applied to a particular set of hosts) into something that is generic, and is therefore useful and has value in a much broader range of contexts.
- Converting something that is relatively complicated, and carries with it the baggage of being mapped to a particular host or group of hosts, into something slimmer and more flexible; a named sequence of tasks.

- Creating the possibility of partitioning a large, flat task space into a hierarchical tasks space, consisting of much simpler modular units at each level.
- Converting something relatively untidy into something relatively neat and orderly.

Compare the number of lines in the original playbook to the playbook that uses a role. There are 19 lines, not counting empty lines and comments, in the original playbook. There are just six lines in **cs-ifdescr2.yml**.

3.1. Create an **ios-cs-ifdescr** role directory structure using **ansible-galaxy**:

```
[student@workstation proj]$ mkdir roles && ansible-galaxy init roles/ios-cs-ifdescr
```

3.2. Show all the directories and files created when you initialized the new role:

```
[student@workstation proj]$ tree roles/
roles/
└── ios-cs-ifdescr
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    │   └── main.yml
    ├── meta
    │   └── main.yml
    ├── README.md
    ├── tasks
    │   └── main.yml
    ├── templates
    ├── tests
    │   ├── inventory
    │   └── test.yml
    └── vars
        └── main.yml

9 directories, 8 files
```

3.3. Get the initial information about your new role.

```
[student@workstation proj]$ ansible-galaxy info \
> --offline --roles-path roles ios-cs-ifdescr

Role: ios-cs-ifdescr
      description:
      dependencies: []
      galaxy_info:
          author: your name
          company: your company (optional)
          galaxy_tags: []
          license: license (GPLv2, CC-BY, etc)
          min_ansible_version: 1.2
...output omitted...
```

- 4. Assign the metadata for the role, and take credit for it, by populating the **meta/main.yml** file.

- 4.1. Look over the choices that are available for describing your new role:

```
[student@workstation proj]$ less roles/ios-cs-ifdescr/meta/main.yml
```

At the very least, consider providing values for keys that are not commented out. By specifying role dependencies, you can automatically pull in other roles when using your role. Our role has no dependencies on other roles.

- 4.2. Back up the original **meta/main.yml** file in case you would like to consult it in the future:

```
[student@workstation proj]$ mv roles/ios-cs-ifdescr/meta/main.yml \
> roles/ios-cs-ifdescr/meta/main.yml.orig
```

- 4.3. Create a **meta/main.yml** file that describes the role. Edit the file and assign the following values:

```
---
galaxy_info:
  author: D0457 Student
  description: Interface descriptions for example.com devices
  company: Examples, Ltd.
  license: ASL 2.0
  min_ansible_version: 2.5
  galaxy_tags:
    - acme
    - network
    - interface
  dependencies: []
```

- 5. Populate the **tasks/main.yml** task file in the new role with the tasks from the **cs-ifdescr.yml** playbook:

```
---
# tasks file for ios-cs-ifdescr
- name: set interface description
  ios_interface:
    aggregate: "{{ interfaces }}"
- name: get interface descriptions
  ios_command:
    commands:
      - show interfaces | include Description
  register: results
- name: show results
  debug:
    var: results.stdout
```

- 6. Create a new playbook named **cs-ifdescr2.yml** that uses the role:

```
---
```

```
- name: manages interface descriptions for cloud services
# use the ios-cs-ifdescr role
hosts: cloud-services

roles:
  - ios-cs-ifdescr
```

- 7. Verify that the new play works as desired:

```
[student@workstation proj]$ ansible-playbook cs-ifdescr2.yml
SSH password: student
```

This concludes the guided exercise.

Customizing Data with Jinja2 Templates

Objectives

After completing this section, you should be able to generate configurations by running a playbook binding values to template slots.

Ansible Variables Use Jinja2

Ansible is written in Python, and uses the Jinja2 templating engine to implement variable substitution in playbooks, roles, and other Ansible YAML files. Variable substitution happens on the control node before tasks are sent and performed on the target machine.

```
---
- name: >
  this play gathers facts from device and
  shows inventory_hostname and configured hostname
hosts:
  - router01
  - router02

tasks:
  - name: use ios_facts to gather facts from {{ inventory_hostname }}
    ios_facts:
      gather_subset: all

  - name: show the configured hostname
    debug:
      msg: >
        the configured hostname of {{ inventory_hostname }}
        is {{ ansible_net_hostname }}
```

The Playbook as Jinja2 Interface

Ansible Playbooks can be used as a front-end interface to the Jinja2 template engine. You can take advantage of Jinja2's powerful template capabilities without writing a single line of Python code.

Modeling the network as an autogenerated set of network device configurations promotes consistency and standardization. Many changes can be implemented by updating variable values or adding new ones. This is separation of concerns in action: data elements are managed independently of vendor- and model- specific implementation details, which are encoded in templates.

Name	Format	Description
Variables file	.ini or .yml/.yaml	File containing individual data elements; group_vars/groupname or host_vars/hostname.yml , and so on.

Name	Format	Description
Template file	.j2	Template file containing Jinja2 parameters (variable names), vendor- and model- specific contextual elements, and other boilerplate text.
Ansible playbook	.yml	The controlling file, an Ansible playbook.

Configuring Devices from Templates

To configure network devices using templates:

1. Separate the platform-independent infrastructure data from platform-dependent configuration statement syntax.
2. Put the former into a **vars** file and the latter into a Jinja2 template. By convention, the directory locations for these are **vars/** and **j2/**.
3. Use the **vars_files** directive to load data as play variables.
4. Tasks can use the **src** option with ***os_config** modules to feed global configuration statements to target devices directly from Jinja2 templates.

An example is provided to illustrate how this works.

Scope of Template Configuration

Feeding configuration statements from templates to devices works only with global config statements.

The ***os_config** modules for platforms with hierarchical interfaces have a **parents** option. It is used to apply config statements at the correct context of a command-line interface hierarchy.

Parents only works with the **lines** option, which takes a list of config statements. The **src** option, which can be used with template files, is mutually exclusive with **lines** and **parents**.

```
$ ansible-doc ios_config
...output omitted...
- parents
  The ordered set of parents that uniquely identify the section or
  hierarchy the commands should be checked against. If the parents
  argument is omitted, the commands are checked against the set of top
  level or global commands.
...output omitted...
- src
  Specifies the source path to the file that contains the configuration
  or configuration template to load. The path to the source file can
  either be the full path on the Ansible control host or a relative path
  from the playbook or role root directory. This argument is
  mutually exclusive with 'lines', 'parents'.
```

Configuring from a Template

The configuration of static, local host names on IOS devices is illustrated here.

When a static host name exists on an IOS device, you can use the name instead of the IP address on that device wherever you would use an address: with the **ping** command, with ACLs, and so forth.

Note that these name-to-IP address mappings exist only on devices where they are configured. They are not part of the DNS service.

Static host names are configured with IOS CLI using this syntax:

```
(config)#ip host hostname ipv4_address
```

Identifying Data for Static Host Names

The data to support IOS static host names consists of a list of hash/dicts that map names to IPv4 addresses. Each row represents a host, and each row consists of a **name** field and an **ipv4** field. The file is saved as **vars/hostnames.yml**.

```
hostname_data:  
- { name: spine01, ipv4: 10.0.0.1/32 }  
- { name: spine02, ipv4: 10.0.0.11/32 }  
- { name: leaf01, ipv4: 192.168.0.1/32 }  
- { name: leaf02, ipv4: 192.168.0.2/32 }  
- { name: cs01, ipv4: 172.16.0.1/32 }  
- { name: server01, ipv4: 10.10.10.2/32 }  
- { name: server02, ipv4: 192.168.10.2/32 }  
- { name: server03, ipv4: 172.16.10.2/32 }
```

It is often useful to store IPv4 information using CIDR notation. This makes it possible to extract various forms of information from a single unit of data: network address, subnet mask, machine address, and so forth. It is recommended practice to use that, as illustrated in the previous definition of the **hostname_data** variable.

Building a Template

The data has been defined as a sequence of hashes/dictionaries, named **hostname_data**. Given that data, the template used to generate IOS static host name commands is relatively simple. The file name is saved as **j2/ios-static-hostnames.j2**.

```
{% for host in hostname_data %}  
ip host {{ host.name }} {{ host.ipv4 | ipaddr('address') }}  
{% endfor %}
```

The Jinja2 **for** command is used to loop over individual members of the **hostname_data** sequence as a dict object named host. The **host.name** variable refers to the **name** field of the host object and **host.ipv4** refers to its **ipv4** field.

The ipaddr filter is used, together with the address argument, to obtain an IPv4 address from the original CIDR form of the data.

Viewing the Results

The IOS CLI command that displays static host names is **show hosts**. You can use an Ansible ad hoc command with the **ios_command** module to view the results.

```
$ ansible -m ios_command -a "commands='show hosts'"  
cs01 | SUCCESS => {  
    "changed": false,  
    "stdout_lines": [  
        [  
            "Host", "Port", "Flags", "Age", "Type", "Address(es)",  
            "spine01", "None (perm, OK) 0 IP", "10.0.0.1",  
            "spine02", "None (perm, OK) 0 IP", "10.0.0.11",  
            "leaf01", "None (perm, OK) 0 IP", "192.168.0.1",  
            "leaf02", "None (perm, OK) 0 IP", "192.168.0.2",  
            "cs01", "None (perm, OK) 0 IP", "172.16.0.1",  
            "server01", "None (perm, OK) 0 IP", "10.10.10.2",  
            "server02", "None (perm, OK) 0 IP", "192.168.10.2",  
            "server03", "None (perm, OK) 0 IP", "172.16.10.2"  
        ]  
    ]  
}
```

► Guided Exercise

Generating Config Statements with Jinja2

In this exercise, you will configure an IOS device with static host objects using commands sourced directly from a Jinja2 template.

Outcomes

You should be able to:

- Create a data variable that maps names to IP addresses.
- Create a Jinja2 template that loops over the data, generating an IOS **ip host** command for each row of data.
- Write a play that uses the **template** module and the **local** connection method to write the set of commands generated by your Jinja2 template to a file.
- Write a play that uses the **ios_config** module (connection method **network_cli**) to configure IOS devices directly from the set of commands generated by your Jinja2 template.

Before You Begin

Open a terminal window on the **workstation** VM and change to your **~/proj** directory.

Instructions

- 1. Create a data variable that maps names to IP addresses. Download the **host-objects.yml** file into the **vars** subdirectory:

```
[student@workstation proj]$ mkdir -p vars
[student@workstation proj]$ cd vars
[student@workstation vars]$ wget \
> http://materials.example.com/full/vars/host-objects.yml
[student@workstation vars]$ cd ..
[student@workstation proj]$ cat vars/host-objects.yml
host_object_data:
- { name: spine01, ipv4: 10.0.0.1/32 }
- { name: spine02, ipv4: 10.0.0.11/32 }
- { name: leaf01, ipv4: 192.168.0.1/32 }
- { name: leaf02, ipv4: 192.168.0.2/32 }
- { name: cs01, ipv4: 172.16.0.1/32 }
- { name: server01, ipv4: 10.10.10.2 }
- { name: server02, ipv4: 192.168.10.2 }
- { name: server03, ipv4: 172.16.10.2 }
```

- 2. Create a Jinja2 template that loops over the data, generating an IOS **ip host** command for each row of data. Create a file named **j2/ios-host-objects.j2** with the following content:

```
{% for obj in host_object_data %}
ip host {{ obj.name }} {{ obj.ipv4 | ipaddr('address') }}
{% endfor %}
```

- 3. Write a play that uses the **template** module and connection method **local** to write the set of commands generated by your Jinja2 template to a file. Create a playbook named **j2test-ios-host-objects.yml** that defines a task that dumps text from your Jinja2 template out to a file named **out/ios-host-objects.cmd**. The playbook should look similar to the following:

```
---
- name: build a file containing IOS static host object statements
  hosts: localhost
  connection: local
  vars:
    srcfile: j2/ios-host-objects.j2
    destfile: out/ios-host-objects.cmd
  vars_files:
    - vars/host-objects.yml

  tasks:

    - name: dump output from template to a file
      template:
        src: "{{ srcfile }}"
        dest: "{{ destfile }}"
```

- 4. Before you run the play, create the **out** directory that will hold the file created by the **template** module:

```
[student@workstation proj]$ mkdir out
[student@workstation proj]$ echo 'directory for generated output' > out/README
```

Run the play found in your new playbook. Type any text when prompted for a password, because it is not used.

```
[student@workstation proj]$ ansible-playbook j2test-ios-host-objects.yml
SSH password: anything
```

- 5. View the output file containing the generated list of IOS **ip host** commands:

```
[student@workstation proj]$ cat out/ios-host-objects.cmd
ip host spine01 10.0.0.1
ip host spine02 10.0.0.11
ip host leaf01 192.168.0.1
ip host leaf02 192.168.0.2
ip host cs01 172.16.0.1
ip host server01 10.10.10.2
ip host server02 192.168.10.2
ip host server03 172.16.10.2
```

- 6. Write a play that uses the **ios_config** module (connection method **network_cli**) to configure IOS devices directly from the set of commands generated by your Jinja2 template.

- 6.1. Create a playbook named **ios-host-objects.yml**. It uses the **src** attribute of the **ios_config** module to source the config lines directly from the Jinja2 template: The **ios-host-objects.yml** playbook must include the following:

```
---
- name: build a file containing IOS static host object statements
  hosts: ios
  vars:
    srcfile: j2/ios-host-objects.j2
  vars_files:
    - vars/host-objects.yml

  tasks:
    - name: configure directly from template
      ios_config:
        src: "{{ srcfile }}"
        when: ansible_network_os == 'ios'
```

- 6.2. Run the play, and this time specify **student** as the SSH password.

```
[student@workstation proj]$ ansible-playbook ios-host-objects.yml
SSH password: student
```

- 6.3. Execute an ad hoc command to confirm that the change was successful:

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include ip host'" cs01
SSH password: student
cs01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "ip host cs01 172.16.0.1\nip host leaf01 192.168.0.1\nip
     host leaf02 192.168.0.2\nip host server01 10.10.10.2\nip host
     server02 192.168.10.2\nip host server03 172.16.10.2 ip host
     spine01 10.0.0.1\nip host spine02 10.0.0.11\n"
  ],
  "stdout_lines": [
    [
      "ip host cs01 172.16.0.1",
      "ip host leaf01 192.168.0.1",
      "ip host leaf02 192.168.0.2",
      "ip host server01 10.10.10.2",
      "ip host server02 192.168.10.2",
      "ip host server03 172.16.10.2",
      "ip host spine01 10.0.0.1",
      "ip host spine02 10.0.0.11",
    ]
  ]
}
```

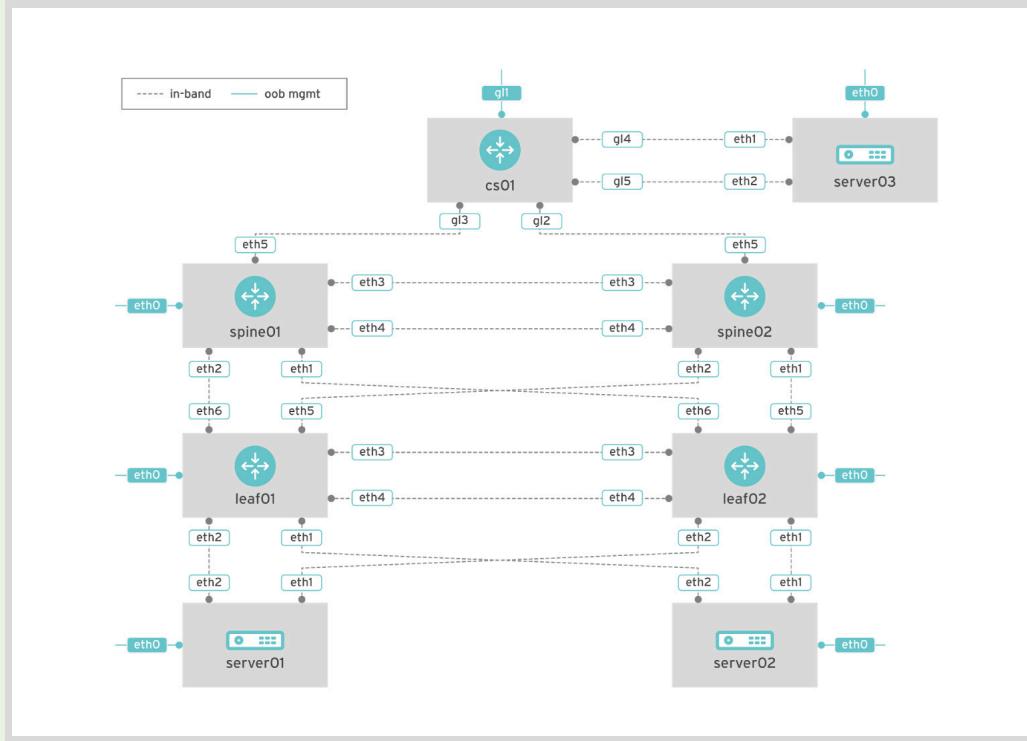
This concludes the guided exercise.

► Lab

Parameterizing Automation

It is important to label interfaces appropriately. This is an opportunity to see how playbooks make use of variables that are set at an appropriate level. In this Lab, interface descriptions are provided by way of group variables. Layer 3 addresses will be mapped to interfaces later.

The interface descriptions used in this lab exercise are based on the following Layer 2 diagram:



Interface Descriptions for Spine Devices

Interface name	Description
eth0	management
eth1	leaf01
eth2	leaf02
eth3	peer-link1
eth4	peer-link2
eth5	cloud-services

Interface Descriptions for Leaf Devices

Interface name	Description
eth0	management
eth1	server01
eth2	server02
eth3	peer-link1
eth4	peer-link2
eth5	spine01
eth5	spine02

In this lab you will write a play that uses roles to apply different sets of interface descriptions to different classes of devices.

Outcomes

You should be able to:

- Define interface data using variables.
- Create a playbook with plays that set interface descriptions for two host groups.
 - One play must use the **spine_interfaces** variable to set interface descriptions for the **spines** group.
 - The other play must use the **leaf_interfaces** variable to set interface descriptions for the **leafs** group.
- Convert each play into a role.
- Create a playbook that uses the new roles.
- Verify that the roles work as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/proj/` directory.

Instructions

1. Define interface data using variables. Define interface data for spine and leaf devices using group variables named **spine_interfaces** and **leaf_interfaces** that map interfaces by name to descriptions.
To save typing, the files can be downloaded using **wget**. Download the **spines** and **leafs** group variable files:

```
[student@workstation proj]$ cd group_vars  
[student@workstation group_vars]$ wget \  
> http://materials.example.com/content/ch3/lab3/group_vars/spines  
[student@workstation group_vars]$ wget \  
> http://materials.example.com/content/ch3/lab3/group_vars/leafs  
[student@workstation group_vars]$ cd ..
```

2. Create a playbook named **spine-leaf-ifdescr.yml** with plays that set interface descriptions for two host groups.
3. Convert the plays into roles named **vyos-spine** and **vyos-leaf**.
 - 3.1. Create appropriate directory structures.
 - 3.2. Describe the new roles and take credit for them by populating their **meta/main.yml** files.
 - 3.3. Reproduce the play tasks as corresponding role tasks.
4. Create a playbook named **spine-leaf-roles.yml** that uses the new roles. It should contain two plays:
 - A play that maps the **vyos-spine** role to **spines**.
 - A play that maps the **vyos-leaf** role to **leafs**.
5. Verify that the roles work as desired.
 - 5.1. Execute **ansible-playbook** with the new playbook to perform the new role-based plays.
 - 5.2. Execute ad hoc commands to verify that all went as expected.

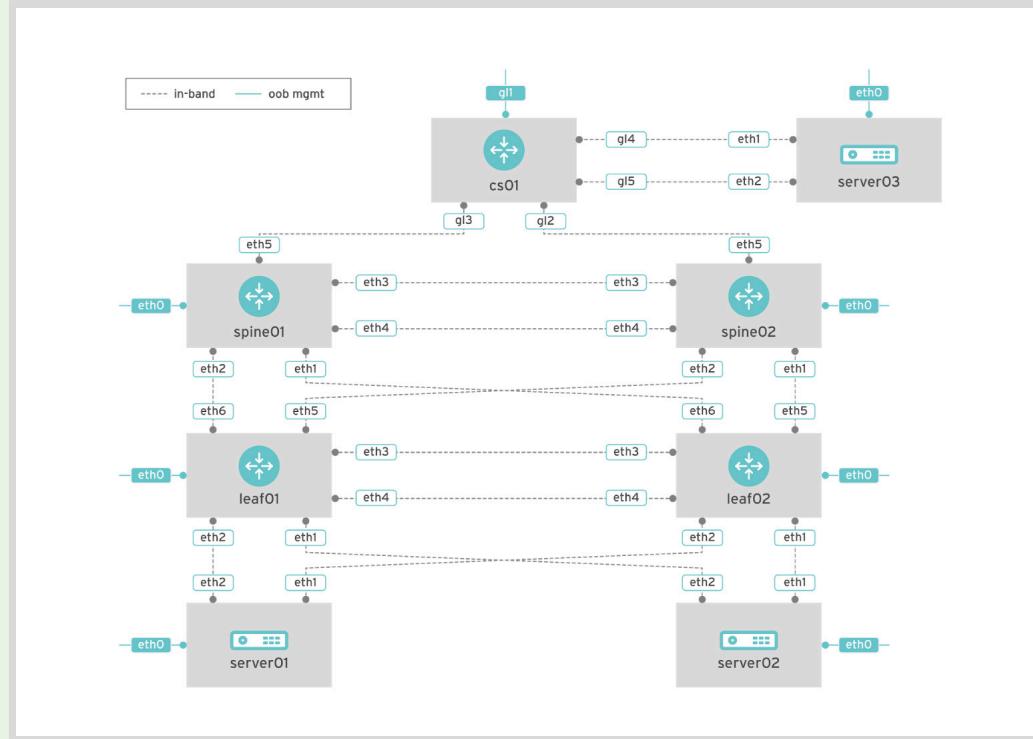
This concludes the lab.

► Solution

Parameterizing Automation

It is important to label interfaces appropriately. This is an opportunity to see how playbooks make use of variables that are set at an appropriate level. In this Lab, interface descriptions are provided by way of group variables. Layer 3 addresses will be mapped to interfaces later.

The interface descriptions used in this lab exercise are based on the following Layer 2 diagram:



Interface Descriptions for Spine Devices

Interface name	Description
eth0	management
eth1	leaf01
eth2	leaf02
eth3	peer-link1
eth4	peer-link2
eth5	cloud-services

Interface Descriptions for Leaf Devices

Interface name	Description
eth0	management
eth1	server01
eth2	server02
eth3	peer-link1
eth4	peer-link2
eth5	spine01
eth5	spine02

In this lab you will write a play that uses roles to apply different sets of interface descriptions to different classes of devices.

Outcomes

You should be able to:

- Define interface data using variables.
- Create a playbook with plays that set interface descriptions for two host groups.
 - One play must use the **spine_interfaces** variable to set interface descriptions for the **spines** group.
 - The other play must use the **leaf_interfaces** variable to set interface descriptions for the **leafs** group.
- Convert each play into a role.
- Create a playbook that uses the new roles.
- Verify that the roles work as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/proj/` directory.

Instructions

1. Define interface data using variables. Define interface data for spine and leaf devices using group variables named **spine_interfaces** and **leaf_interfaces** that map interfaces by name to descriptions.

To save typing, the files can be downloaded using **wget**. Download the **spines** and **leafs** group variable files:

```
[student@workstation proj]$ cd group_vars  
[student@workstation group_vars]$ wget \  
> http://materials.example.com/content/ch3/lab3/group_vars/spines  
[student@workstation group_vars]$ wget \  
> http://materials.example.com/content/ch3/lab3/group_vars/leafs  
[student@workstation group_vars]$ cd ..
```

```
[student@workstation proj]$ cat group_vars/spines  
spine_interfaces:  
- { name: eth0, description: mgmt }  
- { name: eth1, description: leaf01 }  
- { name: eth2, description: leaf02 }  
- { name: eth3, description: peer-link1 }  
- { name: eth4, description: peer-link2 }  
- { name: eth5, description: cloud-services }
```

```
[student@workstation proj]$ cat group_vars/leafs  
leaf_interfaces:  
- { name: eth0, description: mgmt }  
- { name: eth1, description: server01 }  
- { name: eth2, description: server02 }  
- { name: eth3, description: peer-link1 }  
- { name: eth4, description: peer-link2 }  
- { name: eth5, description: spine01 }  
- { name: eth6, description: spine02 }
```

2. Create a playbook named **spine-leaf-ifdescr.yml** with plays that set interface descriptions for two host groups.
 - 2.1. Create a playbook named **spine-leaf-ifdescr.yml** with plays that set interface descriptions for two host groups.
 - A play that uses the **spine_interface** variable to set interface descriptions for the **spines** group.
 - A play that uses the **leaf_interface** variable to set interface descriptions for the **leafs** group.

```
[student@workstation proj]$ cat spine-leaf-ifdescr.yml  
---  
- name: set interface descriptions for spine devices  
  hosts: spines  
  # interface description data is in group variables file  
  
  tasks:  
  
    - name: set interface description  
      vyos_interface:  
        aggregate: "{{ spine_interfaces }}"  
  
    - name: set interface descriptions for leaf devices  
      hosts: leafs
```

```

gather_facts: no
# interface description data is in group variables file

tasks:

- name: set interface description
  vyos_interface:
    aggregate: "{{ leaf_interfaces }}"

```

The **spine** and **leaf** devices use the same authentication credentials, so you could optionally model this using a **when** statement.

```

[student@workstation proj]$ cat spine-leaf-ifdescr2.yml
---
- name: set interface descriptions for spine and leaf devices
  hosts: vyos
  # interface description data is in group variables file

  tasks:

    - name: set interface description
      vyos_interface:
        aggregate: "{{ spine_interfaces }}"
      when: inventory_hostname in groups['spines']

    - name: set interface description
      vyos_interface:
        aggregate: "{{ leaf_interfaces }}"
      when: inventory_hostname in groups['leafs']

```

2.2. Check the plays in the playbook to determine if they do what they are intended to do.

```

[student@workstation proj]$ ansible-playbook --syntax-check spine-leaf-ifdescr.yml
...output omitted...
[student@workstation proj]$ ansible-playbook --check spine-leaf-ifdescr.yml
SSH password: vyos
...output omitted...
[student@workstation proj]$ ansible-playbook --syntax-check spine-leaf-
ifdescr2.yml
...output omitted...
[student@workstation proj]$ ansible-playbook --check spine-leaf-ifdescr2.yml
SSH password: vyos
...output omitted...

```

3. Convert the plays into roles named **vyos-spine** and **vyos-leaf**.

3.1. Create appropriate directory structures.

Create the **~/proj/roles/** directory if it does not already exist:

```
[student@workstation proj]$ mkdir -p roles
```

Create **vyos-spine** and **vyos-leaf** role directory structures in the **~/proj/roles/** directory using **ansible-galaxy**:

```
[student@workstation proj]$ ansible-galaxy init roles/vyos-spine
[student@workstation proj]$ ansible-galaxy init roles/vyos-leaf
```

- 3.2. Describe the new roles and take credit for them by populating their **meta/main.yml** files.

Preserve the original **meta/main.yml** files:

```
[student@workstation proj]$ mv roles/vyos-spine/meta/main.yml \
> roles/vyos-spine/meta/main.yml.orig
[student@workstation proj]$ mv roles/vyos-leaf/meta/main.yml \
> roles/vyos-leaf/meta/main.yml.orig
```

Create new **meta/main.yml** files that describe the roles:

```
[student@workstation proj]$ $ cat roles/vyos-spine/meta/main.yml
---
galaxy_info:
  author: D0457 Student
  description: Interface descriptions for example.com VyOS spines
  company: Example, Ltd.
  license: ASL 2.0
  min_ansible_version: 2.5
  galaxy_tags:
    - acme
    - network
    - interface
dependencies: []

[student@workstation proj]$ $ cat roles/vyos-leaf/meta/main.yml
---
galaxy_info:
  author: D0457 Student
  description: Interface descriptions for example.com VyOS leafs
  company: Example, Ltd.
  license: ASL 2.0
  min_ansible_version: 2.5
  galaxy_tags:
    - acme
    - network
    - interface
dependencies: []
```

- 3.3. Reproduce the play tasks as corresponding role tasks.

Populate the **roles/vyos-spine/tasks/main.yml** file with tasks from the corresponding play:

```
[student@workstation proj]$ cat roles/vyos-spine/tasks/main.yml
---
# tasks file for vyos-spine
- name: set interface description
  vyos_interface:
    aggregate: "{{ spine_interfaces }}"
```

Populate the **roles/vyos-leaf/tasks/main.yml** file with tasks from the corresponding play:

```
[student@workstation proj]$ cat roles/vyos-leaf/tasks/main.yml
---
# tasks file for vyos-leaf
- name: set interface description
  vyos_interface:
    aggregate: "{{ leaf_interfaces }}"
```

4. Create a playbook named **spine-leaf-roles.yml** that uses the new roles. It should contain two plays:
 - A play that maps the **vyos-spine** role to **spines**.
 - A play that maps the **vyos-leaf** role to **leafs**.

```
[student@workstation proj]$ cat spine-leaf-roles.yml
---
- name: Map roles to spines and leafs
  hosts: vyos

  tasks:
    - name: apply vyos-spine role to spines
      include_role:
        name: vyos-spine
      when: inventory_hostname in groups['spines']

    - name: apply vyos-leaf role to leafs
      include_role:
        name: vyos-leaf
      when: inventory_hostname in groups['leafs']
```

5. Verify that the roles work as desired.
 - 5.1. Execute **ansible-playbook** with the new playbook to perform the new role-based plays.

```
[student@workstation proj]$ ansible-playbook spine-leaf-roles.yml
SSH password: vyos
...output omitted...
```

- 5.2. Execute ad hoc commands to verify that all went as expected.

```
[student@workstation proj]$ ansible -m vyos_command -a "commands='sh int'" vyos
SSH password: vyos
...output omitted...
```

This concludes the lab.

Chapter 4

Administering Ansible

Goal

Discuss how Ansible solves administrative challenges faced by enterprises today.

Objectives

- Manage advanced inventories, safeguard information with Ansible Vault.
- Define roles and manage infrastructure using Red Hat Ansible Tower.

Sections

- Ansible in the Enterprise
- Safeguarding Sensitive Data with Ansible Vault (and Guided Exercise)
- Running Plays with Encrypted Data (and Guided Exercise)
- Protecting Resources with Ansible Vault (and Guided Exercise)
- Creating Inventories Using YAML (and Guided Exercise)
- Generating and Using Dynamic Inventories (and Guided Exercise)
- Centrally Running Ansible with Red Hat Ansible Tower (and Guided Exercises)

Lab

Administering Automation

Ansible in the Enterprise

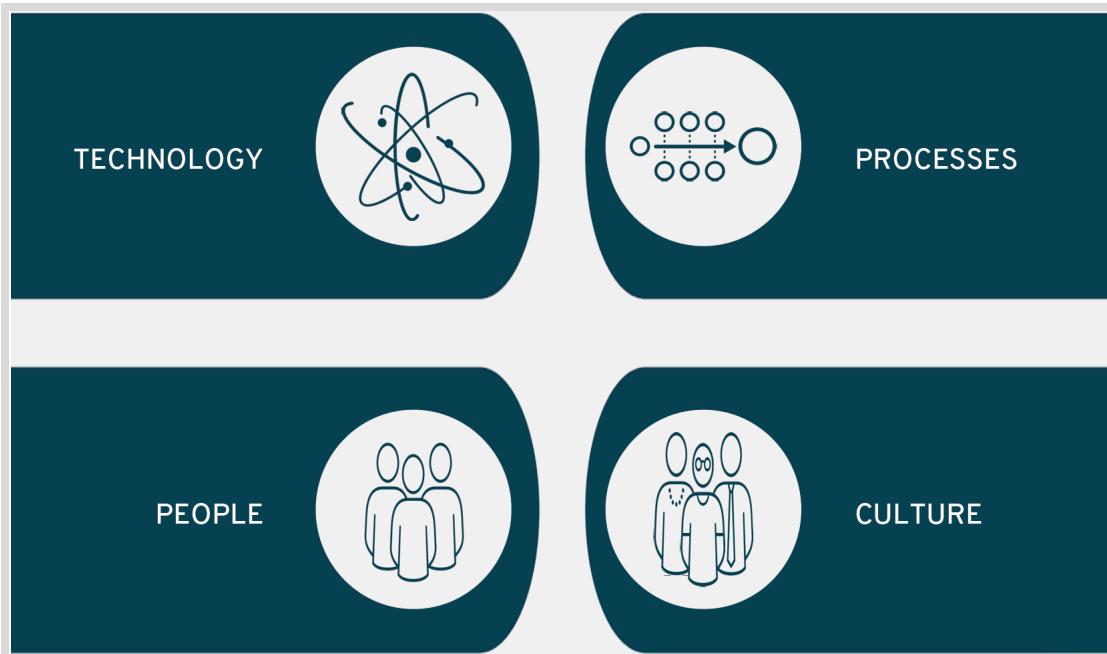
Objectives

After completing this section, you should be able to:

- Manage Ansible projects with version control software.
- Safeguard information with Ansible Vault.
- Use advanced inventory management techniques.
- Manage Ansible resources and infrastructure using Red Hat Ansible Tower.

Scaling Ansible in the Enterprise

Scaling automation successfully in an enterprise involves more than technology.



Defining the Ingredients of Scalability

What characteristics do we expect a scalable automation solution to exhibit?

Some of these depend on processes and procedures or culture. Others involve technology:

- A consistent project architecture: anyone can look at any project and understand how the parts are related to the whole
- A consistent coding style: anyone can look at any project and recognize a familiar coding style
- Secure storage of credentials
- Easy and reliable job scheduling

- An API for programmatic job execution
- Operation accountability, reporting, auditing, and analytics

Identifying Key Actions for Success

What actions make a difference in achieving smooth and successful scalability of automation?

The following actions can help an enterprise scale automation successfully:

- Develop a compelling vision of the role of automation in the enterprise.
- Translate the vision into a sound plan.
- Involve the security team as early as possible.
- Communicate effectively to a team that understands the plan and shares the vision.
- Foster a culture of effective processes and procedures.
- Integrate security practices effectively from first principles: built-in, not bolted-on.
- Capitalize on the potential of automation to improve the security posture in the enterprise. That is, take advantage of available controls to reduce risk exposure.

Applying This to Ansible

How does Ansible fit as a scalable enterprise automation solution?

Some of the ways that Ansible facilitates scalability in an enterprise are described below:

- Ansible is agentless and uses native transports. It integrates with and builds on existing security measures, mechanisms, and safeguards. This makes it the perfect tool for managing network gear, where one rarely has the luxury of installing agent software.
- Ansible Vault securely protects credentials. Protecting credentials used to access and manage infrastructure is of paramount importance.
- Red Hat Ansible Tower makes the job of security professionals easier, by providing a centrally controlled access platform with built-in audit capabilities.
- If access is already carefully controlled with bastion/jump box, multifactor processes, and so forth, there is a relatively simple and seamless transition, because Ansible supports proxy with bastion.
- If access is relatively open and uncontrolled, Ansible is an opportunity for improved control and security.
- Ansible is designed to work well in conjunction with best practices from DevOps, DevNetOps, and DevSecOps.

Best Practices from Dev and Ops

Development

- Version control
- Code review
- Continuous testing

Operations

- Change control and change management
- Provisioning
- Reliability

Structuring Projects for Scalability

Here is a directory structure designed for scalability. It works well with version control software and Red Hat Ansible Tower.

Playbooks are stored at the project root.

```
ansible-project/
└── ansible.cfg
└── .gitignore          ; ignore roles and other things
└── group_vars          ; "play" level group_vars
    └── all.yml
    └── dev.yml
    └── prod.yml
└── library              ; custom modules available across all roles
└── playbook.yml
└── roles
    └── requirements.yml ; keep your roles in separate Git repos, not local
```

Roles provide modularity and reusability and repository-based roles are highly scalable.

Safeguarding Sensitive Data with Ansible Vault

Objectives

After completing this section, you should be able to:

- Create a new encrypted file or encrypt an existing file using Ansible Vault.
- View, edit, or change the password on an existing file encrypted with Ansible Vault.
- Remove encryption from a file that has been encrypted with Ansible Vault.

Identifying Potential Sources

Ansible needs information in order to manage systems. This may include sensitive data such as passwords or private keys (including API keys). When secrets are stored in files as plain text, users with read access to the file system where the files are stored, or to version control systems that do not exclude the files, can see them.

There are at least two ways to store the data more safely:

- Use Ansible Vault, which is included with Ansible and can encrypt and decrypt any structured data file used by Ansible.
- Use a third-party encryption or key management tool or service to store the data.

In this section, you will learn how to use Ansible Vault.

Using Ansible Vault

What can you do with Ansible vault? How do you do it?

Here are some of the things you can do with Ansible Vault:

- Encrypt a string
- Create a new encrypted file
- Encrypt an existing file
- View an encrypted file without opening it for editing
- Edit an encrypted file
- Change the Vault password for an encrypted file
- Permanently decrypt an encrypted file
- Provide the Vault password to a play so it can read data from encrypted files
- Get the Vault password from a password file instead of providing it interactively

Encrypting Strings

To encrypt a string, use this syntax:

```
[user@host ~]$ ansible-vault encrypt_string "a string to be encrypted."  
Vault password: secret
```

Strings encrypted in this manner can be embedded in inventory files and variable files (including group and host variable files). When the Vault password has been provided, Ansible automatically decrypts strings found in such files.

Encrypting New Files with Vault

To create a new encrypted file:

```
[user@host ~]$ ansible-vault create secret.yml  
New Vault password: redhat  
Confirm New Vault password: redhat
```

This prompts you for the new Vault password and then opens a file using the default editor.

Encrypting Existing Files with Vault

To encrypt one or more existing files, use the following command:

```
[user@host ~]$ ansible-vault encrypt secret1.yml secret2.yml  
New Vault password: redhat  
Confirm New Vault password: redhat  
Encryption successful
```

Viewing Encrypted Files

To view an encrypted file without opening it for editing, use the following command:

```
[user@host ~]$ ansible-vault view secret1.yml  
Vault password: redhat
```

Editing Encrypted Files

To edit an existing encrypted file, use the command syntax below. It decrypts the file to a temporary file and lets you edit the file. When saved, it copies the content and removes the temporary file.

```
[user@host ~]$ ansible-vault edit secret.yml  
Vault password: redhat
```



Important

The **edit** subcommand always rewrites the file, so it should only be used when making changes. This can have implications when the file is kept under version control. The **view** subcommand should always be used to see the file's contents without making changes.

Modifying the Encryption Status

To change the Vault password for an encrypted file or files:

```
[user@host ~]$ ansible-vault rekey secret.yml  
Vault password: redhat  
New Vault password: RedHat  
Confirm New Vault password: RedHat  
Rekey successful!
```

To permanently decrypt an encrypted file:

```
[user@host ~]$ ansible-vault decrypt secret1.yml  
Vault password: redhat  
Decryption successful!
```

► Guided Exercise

Safeguarding Sensitive Data with Ansible Vault

In this exercise, you will use Ansible Vault to add encrypted passwords to existing group variables files. Then you will delete these files and replace them with fully encrypted ones. You will subsequently rekey the encrypted files. You will then view the contents of the encrypted files and edit an encrypted file.

Outcomes

You should be able to:

- Encrypt a plain text file and create a new encrypted file.
- Change the Vault password of your encrypted files.
- View the contents of an encrypted file.
- Edit an encrypted file.

Before You Begin

Open a terminal window on the **workstation** VM.

► 1. Prepare a working directory for this guided exercise.

1.1. Create a directory named **ge4-1/** and change into it.

```
[student@workstation ~]$ mkdir ge4-1  
[student@workstation ~]$ cd ge4-1
```

1.2. Download the **ansible.cfg** and **inventory** files.

```
[student@workstation ge4-1]$ wget \  
> http://materials.example.com/content/ch4/ge4-1/ansible.cfg  
[student@workstation ge4-1]$ wget \  
> http://materials.example.com/content/ch4/ge4-1/inventory
```

1.3. Create a **group_vars** subdirectory in the **ge4-1/** directory and change into it.

```
[student@workstation ge4-1]$ mkdir group_vars  
[student@workstation ge4-1]$ cd group_vars
```

1.4. Download the **network**, **ios**, and **vyos** group variables files.

```
[student@workstation group_vars]$ wget \
> http://materials.example.com/content/ch4/ge4-1/group_vars/network
[student@workstation group_vars]$ wget \
> http://materials.example.com/content/ch4/ge4-1/group_vars/ios
[student@workstation group_vars]$ wget \
> http://materials.example.com/content/ch4/ge4-1/group_vars/vyos
```

15. Change back to the parent of the **group_vars** directory.

```
[student@workstation group_vars]$ cd ..
```

16. Verify that when you provide the appropriate SSH password, you can connect and authenticate to IOS and VyOS devices with Ansible.

1.6.1. Use an ad hoc command to run the Ansible **ping** module to verify that you can connect and authenticate to IOS devices. Use the **-k** option so that the **ansible** command prompts for the SSH password. The SSH password for IOS devices is **student**.

```
[student@workstation ge4-1]$ ansible -k -m ping ios
SSH password: student
cs01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

1.6.2. Use an ad hoc command to run the Ansible **ping** module to verify that you can connect and authenticate to VyOS devices. Use the **-k** option so that the **ansible** command prompts for the SSH password. The SSH password for VyOS devices is **vyos**.

```
[student@workstation ge4-1]$ ansible -k -m ping vyos
SSH password: vyos
spine02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
spine01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
leaf02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
leaf01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

- 2. Encrypt a plain text file and create a new encrypted file.

- 2.1. Edit the **group_vars/ios** file and add the **ansible_ssh_pass** variable with the value **student**. It should look like this when you are done:

```
ansible_network_os: ios
ansible_user: admin
ansible_ssh_pass: student
```

- 2.2. Now encrypt the file. Use **foobar** when prompted to provide the Vault password.

```
[student@workstation ge4-1]$ ansible-vault encrypt group_vars/ios
New Vault password: foobar
Confirm New Vault password: foobar
Encryption successful
```

- 2.3. Verify that the file is encrypted.

```
[student@workstation ge4-1]$ cat group_vars/ios
$ANSIBLE_VAULT;1.1;AES256
$ANSIBLE_VAULT;1.1;AES256
39393935323362373335323630613166323066316536373335316365336261373765656331626535
6238623735396634393638623264343761646237626238660a663539333330383738316538326365
363465323032636162353738613263373164386135339343732613838623535337616365663536
6539386535366136300a393839343464343330383839303135393634373466323665616530646361
38386330323732346431313161666331396238356532363730623933356334373765396163633665
37636532326462623937636566303633376635346465633035396437633135353937353166366366
6635373139363834653662636264643833633766239363737623732643863626432663731323562
6262653066316132663631343930653935323136666333065623839663261656234653538366132
6637
```

- 2.4. Delete the **group_vars/vyos** file.

```
[student@workstation ge4-1]$ rm group_vars/vyos
```

- 2.5. Create a new, encrypted, **group-vars/vyos** file. Continue to use **foobar** as the Vault password.

```
[student@workstation ge4-1]$ ansible-vault create group_vars/vyos
New Vault password: foobar
Confirm New Vault password: foobar
```

- 2.6. The file should contain this text:

```
ansible_network_os: vyos
ansible_user: vyos
ansible_ssh_pass: vyos
```

- 2.7. Verify that the file you just created is encrypted.

```
[student@workstation ge4-1]$ cat group_vars/vyos
$ANSIBLE_VAULT;1.1;AES256
35616661613563653666393263323961343263613439333566333565333534353133653538613263
```

```
3432306334353032336631663636383361313861656536610a613235343330653739376365663830  
336134363132346662636562313537396331663834646338665396661363336363835373661  
6538626334323037630a643431393863386634313164393233653431363934393761653437353036  
31393838623262663261316632356432323632653735633264393364626466346263356561383530  
63306261653534323564363039623863633932383566323635336531353531633735616137353430  
3162313630663137306431363438336163646631313066632393365376539636334326632393364  
38373234613030373961
```

- 2.8. Verify that you can now connect and authenticate to both IOS and VyOS devices by providing only the Vault password, which you set to **foobar**.

```
[student@workstation ge4-1]$ ansible --ask-vault-pass -m ping network  
Vault password: foobar  
leaf02 | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}  
spine02 | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}  
cs01 | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}  
spine01 | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}  
leaf01 | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}
```

- 3. Change the Vault password of your encrypted files.

- 3.1. Rekey the **group_vars/ios** file. Set the new Vault password to **redhat**.

```
[student@workstation ge4-1]$ ansible-vault rekey group_vars/ios  
Vault password: foobar  
New Vault password: redhat  
Confirm New Vault password: redhat  
Rekey successful
```

- 3.2. Rekey the **group_vars/vyos** file. Set the new Vault password to **redhat**.

```
[student@workstation ge4-1]$ ansible-vault rekey group_vars/vyos  
Vault password: foobar  
New Vault password: redhat  
Confirm New Vault password: redhat  
Rekey successful
```

- 4. View the contents of an encrypted file.

- 4.1. View the contents of the **group_vars/ios** file. The Vault password should now be **redhat**.

```
[student@workstation ge4-1]$ ansible-vault view group_vars/ios  
Vault password: redhat  
ansible_network_os: ios  
ansible_user: admin  
ansible_ssh_pass: student
```

- 4.2. View the contents of the **group_vars/vyos** file. The Vault password should now be **redhat**.

```
[student@workstation ge4-1]$ ansible-vault view group_vars/vyos  
Vault password: redhat  
ansible_network_os: vyos  
ansible_user: vyos  
ansible_ssh_pass: vyos
```

► 5. Edit an encrypted file.

- 5.1. Add this variable to the **group_vars/ios** file:

```
banner_message: Access is restricted to authorized users only.
```

- 5.2. Edit the **group_vars/ios** file. The Vault password should be **redhat**.

```
[student@workstation ge4-1]$ ansible-vault edit group_vars/ios  
Vault password: redhat
```

- 5.3. Display the file to confirm that the new variable has been successfully added.

```
[student@workstation ge4-1]$ ansible-vault view group_vars/ios  
Vault password: redhat  
ansible_network_os: ios  
ansible_user: admin  
ansible_ssh_pass: student  
banner_message: Access is restricted to authorized users only.
```

This concludes the guided exercise.

Running Plays with Encrypted Data

Objectives

After completing this section, you should be able to run a playbook that uses data or files encrypted with Ansible Vault.

Providing Access to Encrypted Data

To read data from encrypted files, Ansible needs to know the Vault password.

The **--ask-vault-pass** option can be used to prompt for interactive input of the password:

```
[user@host ~]$ ansible-playbook --ask-vault-pass site.yml  
Vault password: redhat
```

The **--vault-password-file** and **--vault-id** options read the Vault password from the Vault password file.



Important

The Vault password file is a plain text file that contains the Vault password as a plain text string stored as a single line. This file is *not* encrypted, so it is vital that it be protected using file permissions or other security measures.

Use the following command to create a Vault password file for the **site.yml** playbook.

```
[user@host ~]$ ansible-playbook --vault-password-file=filename site.yml
```

Protecting the Vault Password File

The Vault password must be provided when accessing files encrypted using Vault. You can be prompted, and provide it interactively, or you can automate the process by creating a Vault password file. A Vault password file stores the Vault password in plain text, so it must be protected by appropriate file system permissions.

Here are some rules for protecting a Vault password file:

- It should be stored in a directory that is outside of the scope of any version control software projects. This prevents it from accidentally having a copy included in project source code.
- The directory where it lives should be owner-only read/write/execute.
- The file itself should be owner-only read/write.

► Guided Exercise

Running Plays with Encrypted Data

The ability to automate operations that would otherwise require manual intervention is a key capability. Working with encrypted data potentially frees processes from blocking on prompts for password entry.

In this exercise, you will use Ansible Vault to encrypt the file containing passwords on the local system and use the encrypted file in a playbook to create users on the **spine01** managed host.

Outcomes

You should be able to use variables defined in the encrypted file when performing a play.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- ▶ 1. Create an encrypted file named **users-secret.yml** in the `~/proj/vars/` directory. This file will define the password variables and store the passwords to be used in the playbook.

Use the associative array variable **newusers** to define users and passwords using the name and pw keys, respectively. Define the **ansibleuser1** user and its **redhat** password. Also define the **ansibleuser2** user and its **Re4H1T** password.

Set the Vault password to **redhat**.

- 1.1. If you do not already have a **vars/** directory, create it now.

```
[student@workstation proj]$ mkdir -p vars
```

- 1.2. Change to the **vars** directory.

```
[student@workstation proj]$ cd vars
```

- 1.3. Create an encrypted file named **users-secret.yml** in **vars/**. Provide a Vault password of **redhat** and confirm it. This will open a file in the default editor.

```
[student@workstation vars]$ ansible-vault create users-secret.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

- 1.4. Add a variable named **newusers**. It should consist of a list of associative arrays, with key/value pairs, where the keys are **name** and **pw**, as illustrated here:

```
---
newusers:
- name: ansibleuser1
  pw: redhat
- name: ansibleuser2
  pw: Re4H1T
```

Save the file.

- ▶ 2. Create a playbook in the `~/proj/` directory named `create-users.yml` that uses the variables defined in the `vars/users-secret.yml` encrypted file.

```
[student@workstation proj]$ cat create-users.yml
---
- name: create users on the spine machines
  hosts: spines
  vars_files:
    - vars/users-secret.yml

  tasks:

    - name: create users
      vyos_user:
        name: "{{ item.name }}"
        configured_password: "{{ item.pw }}"
        state: present
      loop: "{{ newusers }}"
      loop_control:
        label: "{{ item.name }}"
```

- ▶ 3. Verify the syntax of the play contained in your playbook, and then run it. The first command in the following example is on one long line that ends with `create-users.yml`.

```
[student@workstation proj]$ ansible-playbook --ask-vault-pass --syntax-check
  create-users.yml
Vault password: redhat

playbook: create-users.yml
[student@workstation proj]$ ansible-playbook --ask-vault-pass create-users.yml
SSH password: vyos
Vault password: redhat

PLAY [create users on the spine machines] ****
*****
TASK [create users] ****
changed: [spine01] => (item=ansibleuser1)
changed: [spine02] => (item=ansibleuser1)
changed: [spine01] => (item=ansibleuser2)
changed: [spine02] => (item=ansibleuser2)
```

```
PLAY RECAP ****
spine01 : ok=1    changed=1    unreachable=0    failed=0
spine02 : ok=1    changed=1    unreachable=0    failed=0
```

- 4. Execute an ad hoc command to confirm that the users exist.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh sys login users'" spines
SSH password: vyos
spine02 | SUCCESS => {
    "changed": false,
...output omitted...
    "stdout_lines": [
        [
            "Username      Type   Tty      From           Last login",
            "ansibleuser1  vyatta          never logged in",
            "ansibleuser2  vyatta          never logged in",
            "vyos          vyatta  pts/0    172.25.250.254  Thu Jul 30 14:40:16
2020"
        ]
    ]
}
spine01 | SUCCESS => {
    "changed": false,
...output omitted...
    "stdout_lines": [
        [
            "Username      Type   Tty      From           Last login",
            "ansibleuser1  vyatta          never logged in",
            "ansibleuser2  vyatta          never logged in",
            "vyos          vyatta  pts/0    172.25.250.254  Thu Jul 30 14:40:15
2020"
        ]
    ]
}
```

This concludes the guided exercise.

Protecting Resources with Ansible Vault

Objectives

After completing this section, you should be able to run playbooks that reference multiple resources encrypted with different Ansible Vault passwords.

Securing Playbook Resources

Plays cannot access data stored in Vault-encrypted files unless the Vault password is provided. If the password is not provided, this message is displayed: **ERROR: A vault password must be specified to decrypt secret.yml**.

There are two ways to provide the Vault password:

- Use the **--ask-vault-pass** option to prompt for the password interactively.
- Use the **--vault-password-file** or **--vault-id** option or the **ANSIBLE_VAULT_PASSWORD_FILE** environment variable to refer to a Vault password file.



Important

The Vault password file is a plain text file that contains the Vault password as a plain text string stored as a single line. This file is *not* encrypted, so it is vital that it be protected using file permissions or other security measures.

Decrypting Content With Multiple IDs

Prior to Ansible 2.4, all files protected by Ansible Vault used by a playbook had to be encrypted using the same password.

Ansible 2.4 and later support multiple Vault passwords because the **ansible-playbook** command now allows **--vault-id** to appear multiple times.

- If multiple Vault passwords are provided, Ansible attempts to decrypt Vault content by trying each Vault secret in the order they were provided on the command line.
- If the Vault content was encrypted using a **--vault-id** option, then the label of the Vault ID is stored with the Vault content. When Ansible knows the right Vault ID, it tries the matching Vault ID's secret first before trying the rest of the Vault IDs.

Recommended Practices

Incorporating Vault-protected files into Ansible projects.

Wherever you have a variables file (in **group_vars** or **host_vars**, for instance) you could consider replacing that with a **vars** file that contains no sensitive information. You can use an encrypted file named **vault** that holds the sensitive data.

A convenient way of organizing this is to introduce new directories under **group_vars** and **host_vars** that represent individual groups or individual hosts, respectively. Put **vars** files and, where appropriate, **vault** files, inside those directories.

```
└── ansible.cfg
└── group_vars
    └── all
        └── group1
            └── vars
                └── vault
    └── group2
        └── vars
└── host_vars
    └── host1
        └── vars
            └── vault
└── inventory
└── playbook.yml
```

Optimizing Vault Performance

By default, Ansible uses functions from the *python-crypto* package to encrypt and decrypt Ansible Vault files. If there are many encrypted files, decrypting them at startup may cause a perceptible delay.

To speed things up, install the *python-cryptography* package:

```
[user@host ~]$ sudo yum install python-cryptography
```



Note

Note that *python-crypto* and *python-cryptography* are two different packages. The former is installed by default, and is required. The latter is optional.

► Guided Exercise

Protecting Resources with Ansible Vault

In this exercise, you will implement a recommended practice directory structure; one which establishes, within directories containing variables files, an easy way to distinguish between plain text files and encrypted ones.

Outcomes

You should be able to:

- Convert the existing **proj/group_vars/groupname** files into **proj/group_vars/groupname/vars.yml** files.
- Create encrypted **vault.yml** files for the **vyos** and **ios** groups, in which the **ansible_ssh_pass** variable is set.
- Create a playbook in the **proj/** directory designed to demonstrate authentication to both VyOS and IOS machines in the Network Lab by providing only the Vault password.
- Run the play, using **--ask-vault-pass** to request to be prompted for the Vault password.
- Create a Vault password file, using file-system permissions to secure it.
- Configure Ansible locally to automatically use the Vault password file.
- Run the play again, this time without being prompted to supply the Vault password.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Convert the existing **proj/group_vars/groupname** files into **proj/group_vars/groupname/vars.yml** files.

Convert the existing **~/proj/group_vars/** directory structure, which stores group-specific variables in files named **groupname**, to a structure in which the variables are stored in **vars.yml** files located in **group_vars/groupname/** directories.



Important

Do EITHER step 1.1 OR step 1.2, not both.

- 1.1. With each **group_vars/groupname** file:

- 1.1.1. Rename the **group_vars/groupname** file to **group_vars/vars.yml**.

- 1.1.2. Create a directory named **group_vars/groupname/**.

- 1.1.3. Move the **group_vars/vars.yml** file into the **group_vars/groupname/** directory.

```
[student@workstation proj]$ mv group_vars/ios group_vars/vars.yml
[student@workstation proj]$ mkdir group_vars/ios
[student@workstation proj]$ mv group_vars/vars.yml group_vars/ios/
[student@workstation proj]$ mv group_vars/leafs group_vars/vars.yml
[student@workstation proj]$ mkdir group_vars/leafs
[student@workstation proj]$ mv group_vars/vars.yml group_vars/leafs/
[student@workstation proj]$ mv group_vars/network group_vars/vars.yml
[student@workstation proj]$ mkdir group_vars/network
[student@workstation proj]$ mv group_vars/vars.yml group_vars/network/
[student@workstation proj]$ mv group_vars/spines group_vars/vars.yml
[student@workstation proj]$ mkdir group_vars/spines
[student@workstation proj]$ mv group_vars/vars.yml group_vars/spines/
[student@workstation proj]$ mv group_vars/vyos group_vars/vars.yml
[student@workstation proj]$ mkdir group_vars/vyos
[student@workstation proj]$ mv group_vars/vars.yml group_vars/vyos/
```

- 1.2. Alternatively, you can download a shell script and run it. It will transform the directory structure for you.

```
[student@workstation proj]$ wget \
> http://materials.example.com/content/ch4/ge4-3/transform-directory-structure.sh
[student@workstation proj]$ chmod +x transform-directory-structure.sh
[student@workstation proj]$ ./transform-directory-structure.sh
```

- ▶ 2. Create encrypted **vault.yml** files for the **vyos** and **ios** groups, setting the **ansible_ssh_pass** variable.

- 2.1. Create an encrypted **vault.yml** file in the **group_vars/vyos/** directory. It should contain the **ansible_password** variable for VyOS machines in the Network Lab. This password is **vyos**. When prompted for the Vault password (New Vault password:), respond with **redhat**. Define the variable and its value using the familiar key: value form. It should appear as **ansible_password: vyos** in the file as you are creating it. Save the file and quit.

```
[student@workstation proj]$ ansible-vault create group_vars/vyos/vault.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

- 2.2. Confirm the file is indeed encrypted.

```
[student@workstation proj]$ cat group_vars/vyos/vault.yml
$ANSIBLE_VAULT;1.1;AES256
3161333393034346266616434326364636134386136636131663236653365393666376132616637
366663376639393234326239643261633933394643962350a353338643634303331323465306333
3764343666636565326364626563326637643035313162303533663530333233436363131363033
3939616338366531360a613332396232613465613865396464306235373834653861303337373263
63303433663139343237323734343136663962343364653434303337363436373837
```

- 2.3. View the content of the encrypted file to confirm that it is correct.

```
[student@workstation proj]$ ansible-vault view group_vars/vyos/vault.yml  
Vault password: redhat  
ansible_password: vyos
```

- 2.4. Create the encrypted **vault.yml** file in the **group_vars/ios/** directory. It should contain the **ansible_password** variable for **ios** machines in the lab network. This password is **student**. The decrypted file should contain the following content:

```
ansible_password: student
```

Save the file and quit. When prompted for the Vault password, respond with **redhat**.

```
[student@workstation proj]$ ansible-vault create group_vars/ios/vault.yml  
New Vault password: redhat  
Confirm New Vault password: redhat
```

- 2.5. Confirm the file is encrypted.

```
[student@workstation proj]$ cat group_vars/ios/vault.yml  
$ANSIBLE_VAULT;1.1;AES256  
30623734306234313231633730323766616165623464386237383337623164613062323334396261  
3430623662396636326135343632383735663263653264310a333039323639363866323833613639  
64646633623330646637346636386230343463303831346332343139616533353138306631643233  
3163613865616566300a663163643762353030376436366133636162373966636132366534656466  
31306239656432333235373731346430393164663833663666353862313166323065
```

- 2.6. View the content of the encrypted file to confirm that it is correct.

```
[student@workstation proj]$ ansible-vault view group_vars/ios/vault.yml  
Vault password: redhat  
ansible_password: student
```

- 3. Write a playbook that demonstrates authentication to both VyOS and IOS machines in the Network Lab by providing only the Vault password. Create a playbook named **multi-vendor-hostname.yml** in the **proj/** directory, and add the following content:

```
---  
- name: demonstrate authentication to network devices by showing hostname  
  hosts: network  
  
  tasks:  
  
    - name: hostname on IOS machine  
      ios_command:  
        commands:  
          - show run | include hostname  
      register: result  
      when: ansible_network_os == 'ios'  
  
    - debug:  
      var: result.stdout
```

```
when: ansible_network_os == 'ios'

- name: hostname on VyOS machine
  vyos_command:
    commands:
      - show host name
  register: result
  when: ansible_network_os == 'vyos'

- debug:
    var: result.stdout
  when: ansible_network_os == 'vyos'
```

- ▶ 4. Run the play, using **--ask-vault-pass** to request to be prompted for the Vault password. You will be prompted for your SSH password. You can provide anything; your input will be ignored.

```
[student@workstation proj]$ ansible-playbook --ask-vault-pass multi-vendor-
hostname.yml
SSH password: anything
Vault password: redhat
```

- ▶ 5. Create a Vault password file, using file-system permissions to secure it.

- 5.1. Create a protected directory named **~/.rhv** outside of the project directory to hold the Vault password file. Set the permissions on this directory to owner-only read/write/execute (700).

```
[student@workstation proj]$ mkdir -p ~/.rhv
[student@workstation proj]$ chmod 700 ~/.rhv
```

- 5.2. Create a Vault password file named **~/.rhv/vault-secret**. Set the permissions on this file to owner-only read/write (600).

```
[student@workstation proj]$ echo redhat > ~/.rhv/vault-secret
[student@workstation proj]$ cat ~/.rhv/vault-secret
redhat
[student@workstation proj]$ chmod 600 ~/.rhv/vault-secret
[student@workstation proj]$ ls -l ~/.rhv/vault-secret
-rw----- 1 student student 7 Jul 30 11:18 /home/student/.rhv/vault-secret
```

- ▶ 6. Configure Ansible locally to automatically use the Vault password file.

- 6.1. Edit the project-specific (local) Ansible configuration. Ensure that the **ask_pass** line is removed, meaning you will no longer be prompted. Also add this line:

```
vault_password_file = /home/student/.rhv/vault-secret
```

Here is a an example of a local **ansible.cfg** file that contains the newly added line:

```
[student@workstation proj]$ cat ansible.cfg
[defaults]
inventory      = inventory
host_key_checking = False
# so you don't have to say gather_facts: no
gathering = explicit
vault_password_file = /home/student/.rhv/vault-secret

[persistent_connection]
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

- 7. Run the play again, this time without being prompted to supply the Vault password.

```
[student@workstation proj]$ ansible-playbook multi-vendor-hostname.yml
```

This concludes the guided exercise.

Creating Inventories Using YAML

Objectives

After completing this section, you should be able to:

- Manage hosts and host groups in inventories using a dynamic inventory.
- Use the **ansible-inventory** command to validate data provided by dynamic inventories.

Reviewing What Inventories Provide

Ansible looks up host and host group information in an inventory. The names in the inventory are used by plays to determine on which hosts and host groups they should run. They are also used to determine the values of host variables and group variables, some of which might change how Ansible connects or authenticates to those managed hosts.

You have seen how to create and edit inventories as files, and how the **ansible**, **ansible-inventory**, and **ansible-playbook** commands locate inventory files. You saw cases, for instance, where the **-i** option is used to have the **ansible** and **ansible-*** programs use a particular inventory file.

Describing Static Inventories

Supported formats for inventory files include INI and YAML.

```
# INI format - filename: inventory
[servers]
server1
server2

[leafs]
leaf1
leaf2

[spines]
spine1
spine2

[network:children]
leafs
spines
```

```
# YAML format - filename: inventory.yml
servers:
  hosts:
    server1:
    server2:

  spines:
```

```
hosts:  
  spine1:  
  spine2:  
  
leafs:  
  hosts:  
    leaf1:  
    leaf2
```

► Guided Exercise

Creating Inventories Using YAML

In this exercise, you will create a static inventory that uses YAML format.

Outcomes

You should be able to:

- Convert the INI format static inventory file that was created in the guided exercise into one that uses the YAML format.
- Validate the inventory you created.

Before You Begin

Open a terminal window on the **workstation** VM.

- 1. Convert the INI format static inventory file that was created in the guided exercise into one that uses the YAML format.
- 1.1. The following is the `~/inventory` file that was created in the guided exercise. If it does not already exist, create it now.

```
[leafs]
leaf[01:02]

[spines]
spine[01:02]

[cloud-services]
cs01

[ios:children]
cloud-services

[vyos:children]
spines
leafs

[network:children]
vyos
ios

[servers]
server[01:03]
```

- 1.2. Using that inventory file, the `ansible-inventory -i inventory --graph` command produces this output:

```
[student@workstation ~]$ ansible-inventory -i inventory --graph
@all:
  |--@network:
  |  |--@ios:
  |  |  |--cloud-services:
  |  |  |  |--cs01
  |  |  |--@vyos:
  |  |  |  |--leafs:
  |  |  |  |  |--leaf01
  |  |  |  |  |--leaf02
  |  |  |  |--@spines:
  |  |  |  |  |--spine01
  |  |  |  |  |--spine02
  |--@servers:
  |  |--server01
  |  |--server02
  |  |--server03
  |--@ungrouped:
```

1.3. Create a hosts inventory file named **inventory.yml** containing this text:

```
servers:
  hosts:
    server[01:03]

spines:
  hosts:
    spine[01:02]

leafs:
  hosts:
    leaf[01:02]

cloud-services:
  hosts:
    cs01

ios:
  children:
    cloud-services:

vyos:
  children:
    spines:
      leafs:

network:
  children:
    vyos:
      ios:
```

► 2. Validate the inventory you created.

- 2.1. Use the **ansible-inventory** command with the INI format inventory file to generate a graph. Redirect the output to a file named **ini-inventory-graph.out**.

```
[student@workstation proj]$ ansible-inventory -i inventory \
> --graph > ini-inventory-graph.out
```

- 2.2. Repeat the same command, this time with the YAML format inventory file. Redirect the output to a file named **yml-inventory-graph.out**.

```
[student@workstation proj]$ ansible-inventory -i inventory.yml \
> --graph > yml-inventory-graph.out
```

- 2.3. Use the **sdiff** tool to compare the two files side by side.

```
[student@workstation proj]$ sdiff -w 60 ini-inventory-graph.out yml-inventory-
graph.out
@all:                                @all:
|--@network:                          |--@network:
|  |--@ios:                            |  |--@ios:
|  |  |--@cloud-services:             |  |  |--@cloud-services:
|  |  |  |--cs01                      |  |  |  |--cs01
|  |  |--@vyos:                       |  |  |--@vyos:
|  |  |--@leafs:                      |  |  |--@leafs:
|  |  |  |--leaf01                    |  |  |  |--leaf01
|  |  |  |--leaf02                    |  |  |  |--leaf02
|  |  |--@spines:                     |  |  |--@spines:
|  |  |  |--spine01                  |  |  |  |--spine01
|  |  |  |--spine02                  |  |  |  |--spine02
|--@servers:                           |--@servers:
|  |--server01                         |  |--server01
|  |--server02                         |  |--server02
|  |--server03                         |  |--server03
|  -@ungrouped:                      |  -@ungrouped:
```

This concludes the guided exercise.

Generating and Using Dynamic Inventories

Objectives

After completing this section, you should be able to run an existing dynamic inventory script that provides automatically updated inventory information to Ansible in JSON format.

Moving Beyond Static Inventories

Files created and edited by hand are tedious to maintain over time as environments change. This is especially challenging when virtual machine instances are dynamically provisioned or deployed; in a cloud computing environment, for instance.

You need inventories stay current by being updated automatically on an ongoing basis. A *dynamic inventory* is a script that returns group and host information in JSON form.

Dynamic inventory scripts rely on information from one or more external sources. The sources can be anything: cloud provider APIs, Cobbler, LDAP directories, DNS, Consul, Infoblox, Device42, CMDB software; whatever the dynamic inventory script was designed to use.

Dynamic inventories are recommended for any large and rapidly changing IT environment, where systems are often deployed, tested, and then removed.

Managing Inventories Under Ansible Tower

Ansible Tower comes with built-in dynamic inventory support for a number of external inventory sources, including:

- Amazon EC2
- Google Compute Engine
- Microsoft Azure Resource Manager
- VMware vCenter
- Red Hat Satellite 6
- Red Hat CloudForms
- OpenStack
- Custom or DIY

In the next section, you will start to work with Red Hat Ansible Tower. You will see how Ansible Tower makes it easy to transition from static to dynamic inventories.

Unlocking Dynamic Inventories

Here we take a closer look at how dynamic inventories work.

A dynamic inventory is a script that must:

- Have a **--list** option that returns a JSON array of group hashes/dictionaries.
- Have a **--host** option that returns either a hash/dictionary of variables or an empty JSON hash/dictionary ({}).
- Non-Ansible Tower dynamic inventory files (scripts or programs) must be marked executable by the file system.

You do not need to create any dynamic inventory scripts for this course, but information is provided on how to understand them.

List Host Groups

Groups are used to find hosts.

The **--list** option must return a JSON encoded hash/dictionary of all groups. It must be formatted as shown here:

```
{  
  "foo": {  
    "hosts": [  
      "host1",  
      "host2"  
    ],  
    "vars": {  
      "var1": true  
    },  
    "children": [  
      "bar"  
    ]  
  },  
  "bar": {  
    "hosts": [  
      "host3",  
      "host4"  
    ],  
    "vars": {  
      "var2": 500  
    },  
    "children": []  
  }  
}
```

Obtaining Host Information

Hosts are used to find variables and their values.

When run with the **--host *HOSTNAME*** option, the script must return either an empty JSON hash/dictionary, or a hash/dictionary of inventory variables for that host:

```
{  
  "foovar": "fooval",  
  "barvar": "barval",  
}
```

Optimizing Inventory Performance

Calling **--host** for every host is not nearly as efficient as having the script return all of the host variables at once. Dynamic inventory scripts can do this by returning all host variables under a top-level element named **_meta**.

```
{
  "_meta": {
    "hostvars": {
      "host1": {
        "var1" : "value"
      },
      "host2": {
        "var2": "value"
      }
    }
  }
}
```

Describing the Outer Structure

What should be the outer structure of well-formed data from a dynamic inventory script?

Ideally, the top-level, skeletal structure of the JSON object returned by a dynamic inventory script should look like this:

```
{
  "_meta": {
    "hostvars": {}
  },
  "all": {
    "children": [
      "ungrouped"
    ]
  },
  "ungrouped": {}
}
```

Validating Dynamic Inventories

The **-i *inventory*** convention at the command line used by the **ansible** and **ansible-*** programs to load inventory data also works with dynamic inventories. That means you can use **ansible-inventory -i *script*** along with other options to verify that the script is working properly and returning results as expected. You can compare this to results obtained with a static inventory. The file does not need to be named **dyn_inv.py**, as long as it is executable.

```
[user@host ~]$ ansible-inventory -i inventory.yml --graph
@all:
|--@leafs:
|   |--leaf1
|   |--leaf2
|--@servers:
|   |--server1
|   |--server2
|--@spines:
|   |--spine1
|   |--spine2
|--@ungrouped:
```

```
[user@host ~]$ ansible-inventory -i dyn_inv.py --graph
@all:
|--@leafs:
|   |--leaf1
|   |--leaf2
|--@servers:
|   |--server1
|   |--server2
|--@spines:
|   |--spine1
|   |--spine2
|--@ungrouped:
```

► Guided Exercise

Generating and Using Dynamic Inventories

In this exercise, you will resolve names by way of a dynamic inventory script when executing ad hoc commands and running plays.

Outcomes

You should be able to:

- Use the **ansible-inventory** command to view the contents provided by a dynamic inventory script.
- Execute an ad hoc command using a dynamic inventory script.
- Use the **ansible-playbook** command with a dynamic inventory script to run a play or plays in a playbook.
- Rename the dynamic inventory script and use an ad hoc command to illustrate that the name is not significant.
- Remove the execute bit from the dynamic inventory script and attempt to run an ad hoc command to illustrate that the script must be executable.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **proj/** directory.

- 1. Use the **ansible-inventory** command to view the contents provided by a dynamic inventory script.
- 1.1. Download the dynamic inventory simulation script **dyninv.py** and the required **data.yml** data file.

```
[student@workstation proj]$ wget \
> http://materials.example.com/playbooks/data.yml
[student@workstation proj]$ wget \
> http://materials.example.com/playbooks/dyninv.py
```

- 1.2. Make the script executable.

```
[student@workstation proj]$ chmod +x dyninv.py
```

- 1.3. Use the **ansible-inventory** command to view the contents provided by a dynamic inventory script.

```
[student@workstation proj]$ ansible-inventory -i dyninv.py --list
```

This demonstrates that the dynamic inventory script is working; host inventory information is being provided by the script instead of a static inventory file.

- 2. Execute an ad hoc command using a dynamic inventory script. It should list interfaces on the VyOS device named **spine01**.

```
[student@workstation proj]$ ansible -i dyninv.py -m vyos_command \
> -a "commands='sh int'" spine01
```

- 3. Use the **ansible-playbook** command with a dynamic inventory script to run a play or plays in a playbook.

- 3.1. Write a playbook named **vyos-dyn-facts1.yml** that gathers facts from devices running VyOS. Ensure it contains the following:

```
---
- name: gather facts for vyos devices
  hosts: vyos

  tasks:
    - name: invoke vyos_facts with gather_subset=all
      vyos_facts:
        gather_subset: all
      when: ansible_network_os == 'vyos'

    - debug:
        var: ansible_facts
```

- 3.2. Use **ansible-playbook** to run the playbook. Specify the dynamic inventory script **dyninv.py** as the host inventory with the **-i INVENTORY** option. Limit it to just **spine01**.

```
[student@workstation proj]$ ansible-playbook -i dyninv.py -l spine01 \
> vyos-dyn-facts1.yml
```

- 4. Rename the dynamic inventory script and execute an ad hoc command to illustrate that the name is not significant.

- 4.1. Rename the script.

```
[student@workstation proj]$ mv dyninv.py myinv
```

- 4.2. Execute an ad hoc command referencing the dynamic inventory script that lists the interfaces on the VyOS device named **spine01**.

```
[student@workstation proj]$ ansible -i myinv -m vyos_command \
> -a "commands='sh int'" spine01
```

This illustrates that the name of the script is not significant, as long as it is a valid file name. We see the **_meta** block, groups and hosts, and so forth.

- ▶ 5. Remove the execute bit from the dynamic inventory script and attempt to execute an ad hoc command to illustrate that the script must be executable.
 - 5.1. Use the **chmod** command to remove the execute bit from the dynamic inventory script.

```
[student@workstation proj]$ chmod -x myinv
```

- 5.2. Execute the same ad hoc command that was successful earlier with the same dynamic inventory script when it was executable.

```
[student@workstation proj]$ ansible -i myinv -m vyos_command \
> -a "commands='sh int'" spine01
[WARNING]: * Failed to parse /home/student/proj/myinv with script plugin:
problem running /home/student/proj/myinv --list ([Errno 13] Permission denied)

...output omitted...

[WARNING]: Could not match supplied host pattern, ignoring: spine01
```

This illustrates that the dynamic inventory script must be executable.

This concludes the guided exercise.

Centrally Running Ansible with Red Hat Ansible Tower

Objectives

After completing this section, you should be able to:

- Store and manage Ansible projects in the Git version control system for use with Ansible Tower.
- Create an Ansible Tower project that uses a Git repository to get and update Ansible materials.
- Create a Job Template that runs a playbook, launch a job using that template, and determine whether the job succeeded or had failures.

Describing Red Hat Ansible Tower

Ansible Tower is a framework for managing IT automation securely and efficiently at scale.

Scaling the IT automation infrastructure across an enterprise can present significant challenges. Ansible core has no built-in ability to manage shared access to playbooks. And to run successfully, some plays require privileged, administrative access. But Ansible core also has no built-in ability to manage privilege levels and access to sensitive information appropriately.

With Ansible Tower:

- Role-based access control keeps environments secure and teams efficient. Complete auditability of compliance with security policies.
- Unprivileged users can safely deploy entire applications with an automated deployment access.
- Ansible automation activity is centrally logged, ensuring complete auditability of compliance with security policies.

Identifying Ansible Tower Features

The key features of Ansible Tower include:

- A centralized web interface for playbook management, role-based access control, and centralized logging and auditing.
- A REST API that ensures that Ansible Tower integrates easily with an enterprise's existing workflows and tool sets.
- API and notification features that make it particularly easy to associate Ansible Playbooks with other tools such as Jenkins, CloudForms, or Red Hat Satellite, to enable continuous integration and deployment.
- Mechanisms to enable centralized use and control of machine credentials and other secrets without exposing them to end users.

Additional Ansible Tower Features

Additional features of Ansible Tower include:

- Visual Dashboard
- Role-based Access Control (RBAC)
- Graphical Inventory Management
- Job Scheduling
- Real-time and Historical Job Status Reporting
- Push-Button Automation
- Remote Command Execution
- Credential Management
- Centralized Logging and Auditing
- Integrated Notifications
- Multiplaybook Workflows
- System Tracking
- RESTful API

Describing the Automation Ecosystem

Human	people	
Organization	organizations, teams roles: users, admins	
Use Case UI UAX API	What configuration management security & compliance continuous delivery app deployment orchestration provisioning	How job templates workflow templates project development playbook development inventory development credentials management permissions management
Automation	front end: Ansible Core back end: Ansible Core	front end: Ansible Tower back end: Ansible Tower
Infrastructure	networks, containers, cloud, services	

Managing the Elements of Ansible Tower

Automation solutions are composed, piece by piece, from the following elements:

- Organizations, teams, and roles (user and admin, for instance).
- Ansible inventories, which define hosts and groups of hosts.
- Role-based permissions to perform actions within Ansible Tower.
- Credentials, which provide remote access to machines.
- Ansible projects, which contain playbooks.
- Job templates, which specify that a playbook from a project should be run on hosts in a particular inventory using specific machine credentials.

A job happens when a user runs a playbook on an inventory by launching a job template.

Navigating the Ansible Tower Web Interface

The Dashboard is the main control center for Ansible Tower.

In the upper left are links to common Ansible Tower resources:

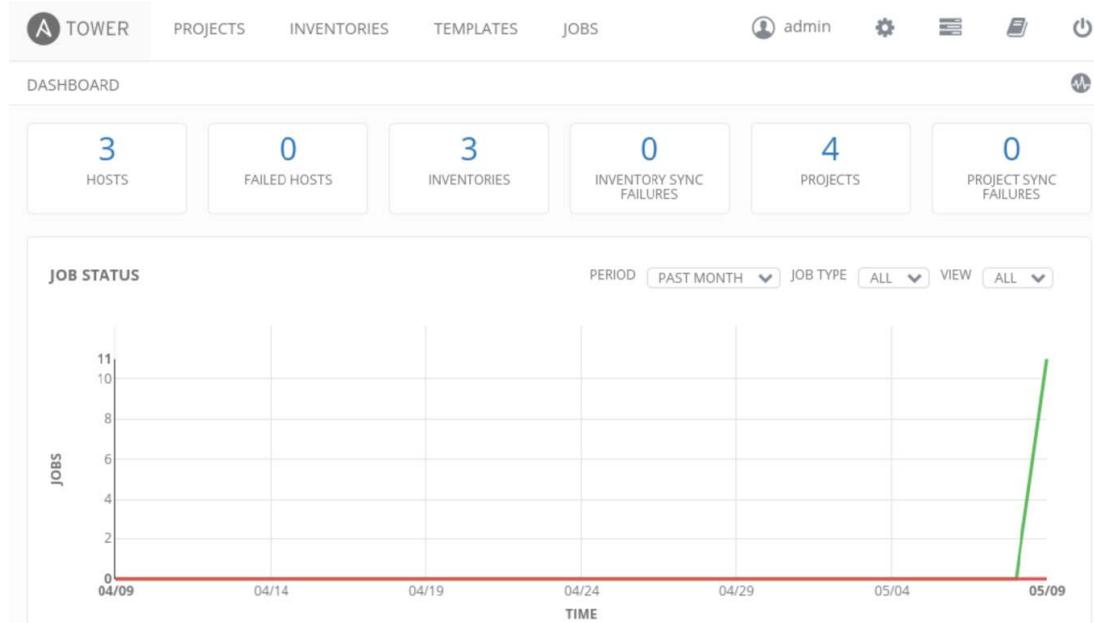
- Projects
- Inventories
- Templates
- Jobs

In the upper right are links to various Ansible Tower administration tools:

- Account configuration
- Settings
- My View
- View Documentation
- Log Out

Viewing the Dashboard

An overview of Ansible Tower activity in one place.



Modifying Settings

If you do not see what you are looking for elsewhere, review the settings.

The screenshot shows the Ansible Tower Settings page. At the top, there are tabs for TOWER, PROJECTS, INVENTORIES, TEMPLATES, and JOBS. On the right, there are user and settings icons. Below the tabs, the word "SETTINGS" is displayed. The main area contains six cards arranged in a grid:

- ORGANIZATIONS**: Group all of your content to manage permissions across departments in your company.
- USERS**: Allow others to sign into Tower and own the content they create.
- TEAMS**: Split up your organization to associate content and control permissions for groups.
- CREDENTIALS**: Add passwords, SSH keys, etc. for Tower to use when launching jobs against machines, or when syncing inventories or projects.
- MANAGEMENT JOBS**: Manage the cleanup of old job history, activity streams, data marked for deletion, and system tracking info.
- INVENTORY SCRIPTS**: Create and edit scripts to dynamically load hosts from any source.
- NOTIFICATIONS**: Create templates for sending notifications with Email, HipChat, Slack, and SMS.
- VIEW YOUR LICENSE**: View and edit your license information.
- CONFIGURE TOWER**: Edit Tower's configuration.

Managing Inventories Dynamically

When you choose a dynamic source, the group's hosts are managed dynamically.

The screenshot shows the "CREATE GROUP" dialog in Ansible Tower. At the top, there are tabs for DETAILS and NOTIFICATIONS. The DETAILS tab is selected. The form includes fields for NAME (Example group), DESCRIPTION, and SOURCE. The SOURCE dropdown is open, showing a list of options:

- Choose a source
- Choose a source
- Manual
- Rackspace Cloud Servers
- Amazon EC2
- Google Compute Engine
- Microsoft Azure Classic (deprecated)

At the bottom of the dialog are CANCEL and SAVE buttons.

Performing Project Operations

To store and manage Ansible projects in a VCS for use with Ansible Tower:

- Use the Ansible Tower [Settings > Credentials] page to create a set of credentials.
- Set the type of the credentials to Source Control.

To create an Ansible Tower project that uses a Git repository to get and update Ansible materials:

- Use the Ansible Tower **PROJECTS** page to create a project.

- Set the **SCM URL**, **SCM BRANCH** and **SCM CREDENTIALS** fields appropriately

Performing Job Related Operations

These are operations related to Jobs and Job Templates.

To create a Job Template that runs a playbook:

- Use the Ansible Tower **TEMPLATES** page to create a Job Template and setting its fields appropriately.

To launch a Job Template:

- On the Ansible Tower **TEMPLATES** page, click the rocket icon in the **ACTIONS** column for that template in the list of templates.

To view the status of a Job:

- Go to the Ansible Tower **JOB**s page and click the magnifying glass icon in the **ACTIONS** column for that job in the list of jobs.

Creating a Job Template

To create a Job Template that runs a playbook:

- Use the Ansible Tower **TEMPLATES** page to create a Job Template and setting its fields appropriately.

The screenshot shows the Ansible Tower interface with the 'TEMPLATES' tab selected. A single template named 'Demo Job Template' is listed. A context menu is open over this template, with 'Job Template' highlighted. Other options in the menu include 'Workflow Job Template', a trash icon, and a settings icon.

NAME	TYPE	ACTIVITY	LABELS
Demo Job Template	Job Template	● ●	

Launching a Job Template

To launch a Job Template:

- On the Ansible Tower **TEMPLATES** page, click the rocket icon in the **ACTIONS** column for that template in the list of templates.

The screenshot shows the Ansible Tower interface with the 'TEMPLATES' tab selected. A single job template, 'Demo Job Template', is listed. The 'ACTIONS' column for this template includes a magnifying glass icon, which is highlighted with a tooltip: 'Start a job using this template'.

NAME	TYPE	ACTIVITY	LABELS	ACTIONS
Demo Job Template	Job Template	● ●		

ITEMS 1 - 1 OF 1

Viewing the Status of a Job

To view the status of a Job:

- Go to the Ansible Tower **JOBS** page and click the magnifying glass icon in the **ACTIONS** column for that job in the list of jobs.

The screenshot shows the Ansible Tower interface with the 'JOBS' tab selected. A list of five jobs is displayed. The first job, 'Demo Job Template', has a magnifying glass icon in the 'ACTIONS' column, which is highlighted with a tooltip: 'View the job'.

ID	NAME	FINISHED	LABELS	ACTIONS
12	Demo Job Template	5/9/2018 9:20:44 AM		
13	Demo Project	5/9/2018 9:20:26 AM		
2	Demo Job Template	5/8/2018 10:58:09 PM		
3	Demo Project	5/8/2018 10:57:50 PM		
1	Demo Project	5/8/2018 10:57:24 PM		

ITEMS 1 - 5 OF 5

► Guided Exercise

Navigating the Red Hat Ansible Tower Web Interface

In this exercise, you will navigate through and interact with the Project, Inventory, Credential, and Template screens in the Ansible Tower web interface and launch a Job.

Outcomes

You should be able to:

- Identify the **Demo Project** that was created during the Ansible Tower installation.
- Browse the **Demo Inventory** that was created during the Ansible Tower installation and determine its managed hosts.
- View the details of the **Demo Credential** that was created during the Ansible Tower installation.
- Determine the Inventory, Project, and Credential used by the **Demo Job Template**.
- Successfully launch a Job.

Before You Begin

Open a web browser on the **workstation** VM. Log in to `http://tower.lab.example.com` as user **admin** and using **student** as the password.

- ▶ 1. Identify the **Demo Project** that was created during the Ansible Tower installation.
 - 1.1. Click the **PROJECTS** tab in the navigation menu to display the list of projects. You should see a project named **Demo Project**, which was created during the Ansible Tower installation.
 - 1.2. Click the **Demo Project** link to view the details of the Project.
 - 1.3. Look at the values of the **SCM TYPE** field and the **SOURCE DETAILS** section to determine the project's origin. The **SCM TYPE** is **Git** and the **SCM URL** is `http://git.lab.example.com:3000/ansible/ansible-tower-samples`.
- ▶ 2. Browse the **Demo Inventory** that was created during the Ansible Tower installation and determine its managed hosts.
 - 2.1. Click **INVENTORIES** in the navigation menu to display the list of known inventories. You should see an inventory named **Demo Inventory**, which was created during the Ansible Tower installation.
 - 2.2. Click the **Demo Inventory** link to view the details of the inventory. Click **HOSTS** and you will see that the inventory contains just one host, **localhost**.
- ▶ 3. View the details of the **Demo Credential** that was created during the Ansible Tower installation.

- 3.1. Click the **Settings** icon (the gear) in the administration toolbar to display the list of administrative interfaces.
 - 3.2. Click **CREDENTIALS** to view the list of credentials.
 - 3.3. Click the **Demo Credential** link to view the details of the credential. The Demo Credential is a machine credential that uses the user name **admin**.
- ▶ **4.** Determine the Inventory, Project, and Credential used by the **Demo Job Template**.
- 4.1. Identify the job template that was created during the Ansible Tower installation. Click **TEMPLATES** in the navigation menu to display the list of existing job templates. You should see a job template named **Demo Job Template**, which was created during the Ansible Tower installation.
 - 4.2. Click the **Demo Job Template** link to view the details of the Job Template.
 - 4.3. Look at the **INVENTORY** field. The Demo Job Template uses the **Demo Inventory** inventory.
 - 4.4. Look at the **PROJECT** field. The Demo Job Template uses the **Demo Project** project.
 - 4.5. Look at the **PLAYBOOK** field. The Job Template executes a **hello_world.yml** playbook.
 - 4.6. Look at the **CREDENTIAL** field. The Job Template uses the **MACHINE: Demo Credential** credential.
- ▶ **5.** Successfully launch a Job.
- 5.1. Launch a job using the **Demo Job Template**.
 - 5.1.1. Exit the template details view by clicking the **TEMPLATES** link in the breadcrumb navigation menu near the top of the screen.
 - 5.1.2. On the **TEMPLATES** screen, click the rocket icon under the **ACTIONS** column of the Demo Job Template row. This launches a job using the parameters configured in the **Demo Job Template** template and redirects you to the job details screen. As the job executes, the details of the job execution as well as its output are displayed.
 - 5.2. Determine the outcome of the job execution.
 - 5.2.1. When the job has completed successfully, the **STATUS** value changes to **Successful**.
 - 5.2.2. Review the output of the job execution to determine which tasks were executed. The **msg** module successfully displayed a "Hello World!" message.
 - 5.3. Review the changes to the dashboard reflecting the job execution.
 - 5.3.1. Click **TOWER** in the upper-left corner of the screen to return to the dashboard.
 - 5.3.2. Review the **JOB STATUS** graph. The green line on the graph indicates the number of recent successful job executions.
 - 5.3.3. Review the **RECENT JOB RUNS** section. This section provides a list of the jobs recently executed as well as their results. The **Demo Job Template** entry

indicates that the Job Template was used to execute a job. The green dot at the beginning of the entry indicates the successful completion of the executed job.

This concludes the guided exercise.

► Guided Exercise

Creating Inventories in Red Hat Ansible Tower

In this guided exercise, you will use Red Hat Ansible Tower to create an organization named FooBar and an inventory for the three-tier, hierarchical topology network used by the FooBar organization.

Outcomes

You should be able to:

- Create an organization in Red Hat Ansible Tower.
- Create an inventory in Red Hat Ansible Tower.
- Add a host group to the inventory you created.
- Add child groups to a parent group.
- Add hosts to groups.

Before You Begin

Open Firefox on the **workstation** VM. Log in to Ansible Tower at `http://tower.lab.example.com` as the user **admin** and using **student** as the password.

- ▶ 1. Create an organization named FooBar.
 - 1.1. Click the **Settings** icon (the gear) in the administration toolbar to display the list of administrative interfaces.
 - 1.2. Select **ORGANIZATIONS**.
 - 1.3. Click **ADD** to create a new organization.
 - 1.4. Set **NAME** to FooBar and click **SAVE**.
- ▶ 2. Create an inventory named **foobar**.
 - 2.1. Click the **INVENTORIES** navigation link at the top of the screen.
 - 2.2. Create the new inventory. On the **INVENTORIES** screen, click **ADD** and select **Inventory** (not **Smart Inventory**). A smart inventory is a collection of hosts defined by a stored search.
 - 2.3. On the **NEW INVENTORY** screen, the inventory **NAME** and **ORGANIZATION** fields are required. Set **NAME** to **foobar**. Set **ORGANIZATION** to **FooBar**. Click **SAVE**.
- ▶ 3. Add a host group to the inventory you created.
 - 3.1. Add a group named **network** to the **foobar** inventory.
On the **foobar** inventory detail screen, select **GROUPS**, and then click **ADD GROUP**. On the **CREATE GROUP** screen, enter **network** in the **NAME** field. Click **SAVE** to finalize the addition of the group to the inventory.

**Note**

When you create a group, the **DETAILS** panel of the **CREATE GROUP** screen has a **VARIABLES** field. The format of the **VARIABLES** field defaults to YAML. To associate variables with a group, define them in the **VARIABLES** field using the **key: value** format. You can add variables to existing groups by navigating to the **DETAILS** panel for the group and adding them to the **VARIABLES** field.

- ▶ **4.** Add child groups to a parent group.
 - 4.1. Add the **access**, **aggregation** and **core** groups as child groups of the **network** group just added to the inventory.
 - 4.1.1. On the **foobar** inventory detail screen, click the **network** group link.
 - 4.1.2. On the network group **DETAILS** screen, select **GROUPS** and then click **ADD**, selecting **New Group**. On the **CREATE GROUP** screen, enter **access** in the **NAME** field. Click **SAVE** to finalize the addition of the child group to the inventory.
 - 4.1.3. On the network group **GROUPS** screen, click **ADD**, selecting **New Group**. On the **CREATE GROUP** screen, enter **aggregation** in the **NAME** field. Click **SAVE** to finalize the addition of the child group to the inventory.
 - 4.1.4. On the network group **GROUPS** screen, click **ADD**, selecting **New Group**. On the **CREATE GROUP** screen, enter **core** in the **NAME** field. Click **SAVE** to finalize the addition of the child group to the inventory.
- ▶ **5.** Add hosts to groups.
 - 5.1. Add **acc01** and **acc02** to the **access** group just added to the inventory.
 - 5.1.1. On **GROUPS** panel of the **INVENTORIES / foobar / GROUPS / network** screen (**ASSOCIATED GROUPS**), click the **access** link in the list of **GROUPS**.
 - 5.1.2. Select the **HOSTS** panel. Click **ADD** and select **New Host**. On the **CREATE HOST** screen, enter **acc01** in the **HOST NAME** field. Click **SAVE** to finalize the addition of the host to the child group.
 - 5.1.3. On the **HOSTS** panel of the **INVENTORIES / foobar / GROUPS / access** screen (**ASSOCIATED HOSTS**), click **ADD** and select **New Host**. On the **CREATE HOST** screen, enter **acc02** in the **HOST NAME** field. Click **SAVE** to finalize the addition of the host to the child group.
 - 5.2. Add **agg01** and **agg02** to the **aggregation** group.
 - 5.2.1. Click **INVENTORIES / foobar / GROUPS** in the breadcrumbs link to go to the **foobar GROUPS** screen.
 - 5.2.2. Click the **network** link in the **GROUPS** list to go to the **GROUPS** panel on the **INVENTORIES / foobar / GROUPS / network** screen (**ASSOCIATED GROUPS**).
 - 5.2.3. Click the **aggregation** link in the list of **GROUPS** to go to the **GROUPS** panel on the **INVENTORIES / foobar / GROUPS / aggregation** screen (**ASSOCIATED GROUPS**).

- 5.2.4. Click **HOSTS** to display the **HOSTS** panel (**ASSOCIATED HOSTS**).
 - 5.2.5. Click **ADD** and select **New Host**. In **HOST NAME**, enter **agg01**. Click **SAVE** to finalize the addition of the host to the group.
 - 5.2.6. Click **ADD** and select **New Host**. In **HOST NAME**, enter **agg02**. Click **SAVE** to finalize the addition of the host to the group.
- 5.3. Add **cor01** and **cor02** to the **core** group.
 - 5.3.1. Click **INVENTORIES / foobar / GROUPS** in the breadcrumbs link to go to the **foobar GROUPS** screen.
 - 5.3.2. Click the **network** link in the **GROUPS** list to go to the **GROUPS** panel on the **INVENTORIES / foobar / GROUPS / network** screen (**ASSOCIATED GROUPS**).
 - 5.3.3. Click the **core** link in the list of **GROUPS** to go to the **GROUPS** panel on the **INVENTORIES / foobar / GROUPS / core** screen (**ASSOCIATED GROUPS**).
 - 5.3.4. Click **HOSTS** to display the **HOSTS** panel (**ASSOCIATED HOSTS**).
 - 5.3.5. Click **ADD** and select **New Host**. In **HOST NAME**, enter **cor01**. Click **SAVE** to finalize the addition of the host to the group.
 - 5.3.6. Click **ADD** and select **New Host**. In **HOST NAME**, enter **cor02**. Click **SAVE** to finalize the addition of the host to the group.

Finish

Because the Ansible Tower license used in the classroom limits the number of hosts that can be managed, remove the hosts you created for this exercise.

Click the **INVENTORIES** navigation link at the top of the screen. On the **INVENTORIES** screen, select the link for the **foobar** inventory.

On the **foobar** inventory detail screen, select **HOSTS** to display the hosts defined in the inventory.

For each host in the list, click **Delete host** (the trash can icon) in the **ACTIONS** column. Click **DELETE** to permanently delete the host from the inventory. Repeat this step for all of the hosts defined in this inventory.

This concludes the guided exercise.

► Lab

Administering Ansible

In this lab, you will create an inventory in Red Hat Ansible Tower for managing network resources.

Outcomes

You should be able to:

- Create an inventory in Red Hat Ansible Tower.
- Add a host group to the inventory you created.
- Add child groups to a parent group.
- Add hosts to groups.

Before You Begin

Open Firefox on the **workstation** VM. Log in to Ansible Tower at `http://tower.lab.example.com` as the user **admin** using **student** as the password.

1. Create an inventory in Red Hat Ansible Tower named **lab.example.com**.
2. Add a host group named **network** to the inventory you created. Associate the **ansible_connection: network_cli** variable and value pair with the group.
3. Add the child groups **vyos** and **ios** to the parent group **network**.
The **vyos** group should set the **ansible_network_os: vyos** and **ansible_user: vyos** variables. The **ios** group should set **ansible_network_os: ios** and **ansible_user: admin** variables.
4. Add the **spine01**, **spine02**, **leaf01**, and **leaf02** hosts to the **vyos** group. Add the **cs01** host to the **ios** group.

This concludes the lab.

► Solution

Administering Ansible

In this lab, you will create an inventory in Red Hat Ansible Tower for managing network resources.

Outcomes

You should be able to:

- Create an inventory in Red Hat Ansible Tower.
- Add a host group to the inventory you created.
- Add child groups to a parent group.
- Add hosts to groups.

Before You Begin

Open Firefox on the **workstation** VM. Log in to Ansible Tower at `http://tower.lab.example.com` as the user **admin** using **student** as the password.

1. Create an inventory in Red Hat Ansible Tower named **lab.example.com**.
 - 1.1. Click the **INVENTORIES** navigation link at the top of the screen.
 - 1.2. Create the new Inventory. On the **INVENTORIES** screen, click **ADD** and select **Inventory** (not **Smart Inventory**).
On the **DETAILS** panel of the **INVENTORIES / CREATE INVENTORY** screen, the **NAME** and **ORGANIZATION** fields are required. Use **lab.example.com** for **NAME**, and click the magnifying glass icon to select the **Default** organization. Click **SAVE**.
2. Add a host group named **network** to the inventory you created. Associate the **ansible_connection: network_cli** variable and value pair with the group.
 - 2.1. On the **DETAILS** panel of the **INVENTORIES / lab.example.com** screen, select **GROUPS**, and then click **ADD GROUP**. On the **CREATE GROUP** screen, enter **network** in the **NAME** field. In the **VARIABLES** field, type the following:

```
---  
ansible_connection: network_cli
```

Click **SAVE** to finalize the addition of the group to the inventory.

3. Add the child groups **vyos** and **ios** to the parent group **network**.
The **vyos** group should set the **ansible_network_os: vyos** and **ansible_user: vyos** variables. The **ios** group should set **ansible_network_os: ios** and **ansible_user: admin** variables.
 - 3.1. On the **DETAILS** panel of the **INVENTORIES / lab.example.com / GROUPS / network** screen, select **GROUPS**. Click **ADD** and select **New Group**.
 - 3.2. On the **DETAILS** panel of the **CREATE GROUP** screen, enter **vyos** in the **NAME** field. In the **VARIABLES** field, define the variables as shown below.

```
---
ansible_network_os: vyos
ansible_user: vyos
```

Click **SAVE** to finalize the addition of the child group to the inventory.

- 3.3. On the **GROUPS** panel of the **INVENTORIES / lab.example.com / GROUPS / network** screen (**ASSOCIATED GROUPS**), click **ADD** and select **New Group**.
- 3.4. On the **DETAILS** panel of the **CREATE GROUP** screen, enter **ios** in the **NAME** field. In the **VARIABLES** field, define the variables as shown below.

```
---
ansible_network_os: ios
ansible_user: admin
```

Click **SAVE** to finalize the addition of the child group to the inventory.

4. Add the **spine01**, **spine02**, **leaf01**, and **leaf02** hosts to the **vyos** group. Add the **cs01** host to the **ios** group.
 - 4.1. Add **spine01**, **spine02**, **leaf01**, and **leaf02** to the **vyos** group.
 - 4.1.1. On the **GROUPS** panel of the **INVENTORIES / lab.example.com / GROUPS / network** screen (**ASSOCIATED GROUPS**), click the **vyos** link.
 - 4.1.2. On the **GROUPS** panel of the **INVENTORIES / lab.example.com / GROUPS / vyos** screen (**ASSOCIATED GROUPS**), select **HOSTS**.
 - 4.1.3. On the **HOSTS** panel, click **ADD** and select **New Host**. On the **CREATE HOST** screen, enter **spine01** in the **NAME** field. Click **SAVE** to finalize the addition of the host to the child group.
 - 4.1.4. On the **HOSTS** panel, click **ADD** and select **New Host** to repeat the process and add **spine02**, **leaf01** and **leaf02** to the group.
 - 4.2. Add **cs01** to the **ios** group.
 - 4.2.1. On the **GROUPS** link in the breadcrumbs to go to the **GROUPS** panel of the **INVENTORIES / lab.example.com** screen.
 - 4.2.2. Click the **network** link to go to the **GROUPS** panel of the **INVENTORIES / lab.example.com / GROUPS / network** screen (**ASSOCIATED GROUPS**).
 - 4.2.3. Click the **ios** link to go to the **GROUPS** panel of the **INVENTORIES / lab.example.com / GROUPS / ios** screen (**ASSOCIATED GROUPS**).
 - 4.2.4. Select **HOSTS** to go to the **HOSTS** panel. Click **ADD** and select **New Host**. Enter **cs01** in the **HOST NAME** field and then click **SAVE** to finalize the addition of the host to the group.

This concludes the lab.

Chapter 5

Automating Simple Network Operations

Goal

Automate simple operations on network devices

Objectives

- Interrogate network devices with automation.
- Back up device configurations.
- Perform simple actions using playbooks.
- Apply simple changes using playbooks.

Sections

- Gathering Network Information with Ansible (and Guided Exercises)
- Configuring Network Devices (and Guided Exercise)
- Configuring the Host Name (and Guided Exercise)
- Configuring System Settings (and Guided Exercise)
- Generating Configuration Settings from Jinja2 Templates (and Guided Exercises)
- Enabling and Disabling Interfaces (and Guided Exercises)
- Reinitializing Layer 3 Interfaces (and Guided Exercise)
- Provisioning the Start-up Network (and Guided Exercise)
- Provisioning Spine and Leaf Devices (and Guided Exercise)
- Setting Parameters with Ansible Tower Surveys (and Guided Exercise)

Lab

Automating Simple Network Operations

Gathering Network Information with Ansible

Objectives

After completing this section, you should be able to:

- Gather data about network devices and networks using facts, ad hoc commands, and playbooks.
- Selectively target and extract information.

Naming of Common Network Device Modules

Ansible provides many platform-specific modules engineered to make it easier to automate the management of networking devices. The names of these modules generally follow a common pattern. These are four common families of modules:

- Gather facts from networking devices using the ***os_facts** family of Ansible modules.
- Issue commands to networking devices using the ***os_command** family of Ansible modules.
- Configure networking devices using the ***os_config** family of Ansible modules.
- Configure layer 3 interfaces using the ***os_l3_interface** family of Ansible modules.

Naming of Other Network Device Modules

The ***os_facts**, ***os_commands**, ***os_config**, and ***os_l3_interface** families of modules represent a small part of the very large collection of ***os_*** modules that make it easier to manage networking devices.

Use the `ansible-doc -l` command to explore what is available.

Modules that Gather Facts on Network Devices

The modules in the ***os_facts** family are used to gather facts from networking devices:

- `cnos_facts`
- `edgeos_facts`
- `enos_facts`
- `eos_facts`
- `ios_facts`
- `junos_facts`
- `nxos_facts`
- `vyos_facts`

Modules that Run Commands on Network Devices

The modules in the ***os_command** family are used to issue commands to networking devices:

- `aireos_command`
- `cnos_command`
- `edgeos_command`
- `enos_command`
- `eos_command`

- `ios_command`
- `junos_command`
- `nxos_command`
- `sros_command`
- `vyos_command`

Modules that Configure Network Devices

The modules in the `*os_config` family are used to configure networking devices:

- `aireos_config`
- `edgeos_config`
- `enos_config`
- `eos_config`
- `fortios_config`
- `ios_config`
- `junos_config`
- `nxos_config`
- `sros_config`
- `vyos_config`

Modules that Configure Layer 3 Interfaces

The modules in the `*os_l3_interface` family are used to configure layer 3 interfaces on networking devices:

- `eos_l3_interface`
- `ios_l3_interface`
- `junos_l3_interface`
- `nxos_l3_interface`
- `vyos_l3_interface`

Gathering Facts

- Each networking platform has an `*os_facts` module.
- The Lab Network has VyOS and IOS devices, so you will use the `vyos_facts` and `ios_facts` modules.

Exploring `vyos_facts` and `ios_facts`

You can use an ad hoc command to show which facts are returned by default.

Now that you are somewhat familiar with Ansible Vault, you can use encrypted files to store all sensitive data, including passwords.

The following command displays the default set of facts returned by the `vyos_facts` module:

```
[user@host ~]$ ansible -i inventory --ask-vault-pass -m vyos_facts vyos
```

The following command displays the facts provided by default by the `ios_facts` module:

```
[user@host ~]$ ansible -i inventory --ask-vault-pass -m ios_facts ios
```

Expanding the Result Set

How many facts can you gather using **vyos_facts** and **ios_facts**?

The following command displays all facts supported by the **vyos_facts** module:

```
$ ansible -i inventory --ask-vault-pass -m vyos_facts spine01 \
> -a 'gather_subset=all'
```

The following command displays all facts supported by the **ios_facts** module:

```
$ ansible -i inventory --ask-vault-pass -m ios_facts cs01 -a 'gather_subset=all'
```

Selecting Facts

The **gather_subset=all** option returns a lot of information. The following playbook displays only specific pieces of information when run. The **vyos_facts** module was used. Note that variables are automatically registered by the facts module.

```
[user@host ~]$ cat vyos-some-facts1.yml
---
- name: a play that gathers some facts and displays them
  hosts: vyos

  tasks:
    - name: invoke vyos_facts
      vyos_facts:
        gather_subset: default

    - debug:
        msg:
          - "Host name: {{ ansible_net_hostname }}"
          - "Model: {{ ansible_net_model }}"
          - "Version: {{ ansible_net_version }}"
          - "SerialNum: {{ ansible_net_serialnum }}"
```

Investigating Fact Selection

The following example runs the playbook to gather some facts.

```
[user@host proj]$ ansible-playbook -l leaf01 vyos-some-facts1.yml

PLAY [a play that gathers some facts and displays them] ****
TASK [invoke vyos_facts] ****
ok: [leaf01]

TASK [debug] ****
ok: [leaf01] => {
  "msg": [
    "Host name: vyos",
    "Model: VMware",
    "Version: VyOS"
```

```
        "SerialNum: VMware-56"
    ]
}

PLAY RECAP ****
leaf01 : ok=2      changed=0      unreachable=0      failed=0
```

► Guided Exercise

Gathering Facts

Facts about network devices contribute to the information that you need to keep the **example.com** network running smoothly. In later exercises, you will build plays that use facts to perform actions; to clear layer 3 addresses from interfaces, for instance.

In *Chapter 4, Administering Ansible*, you removed the need for each invocation to pass the SSH credentials to the managed nodes in order for **ansible-playbook** to run. If you did not complete *Guided Exercise: Safeguarding Sensitive Data with Ansible Vault*, do it now, because this and later lessons assume that it is done. Some exercises in this chapter presume a new structure for **group_vars**, for which you can refer to *Guided Exercise: Identifying Resources for Installed Plug-ins*.

In this exercise, you will gather and process facts.

Outcomes

You should be able to:

- Examine the default set of facts returned by the **vyos** and **ios** instances of **os_facts**.
- Examine the full set of facts returned by the **vyos** and **ios** instances of **os_facts**.
- Recognize and understand differences between fact subsets, and how to interpret and make use of the data returned by facts modules.
- Compose and run a playbook that displays specific facts.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **proj/** directory.

► 1. Examine the default set of facts returned by the **vyos** and **ios** instances of **os_facts**.

- 1.1. Execute this ad hoc command to display the default set of facts returned for a host by the **vyos_facts** module.

```
[student@workstation proj]$ ansible -m vyos_facts spine01
spine01 | SUCCESS => {
    "ansible_facts": {
        "ansible_net_gather_subset": [
            "neighbors",
            "default"
        ],
        "ansible_net_hostname": "vyos",
        "ansible_net_model": "OpenStack",
        "ansible_net_serialnum": "00000000-0000-0000-0000-ac1f6bc7ee96",
        "ansible_net_version": "VyOS"
    },
    "changed": false
}
```

- 1.2. Execute this ad hoc command to display the default set of facts returned for a host by the **ios_facts** module.

```
[student@workstation proj]$ ansible -m ios_facts cs01
```

- 2. Examine the full set of facts returned by the **vyos** and **ios** instances of **os_facts**.

- 2.1. Execute this ad hoc command to display the full set of facts available under **vyos_facts**.

```
[student@workstation proj]$ ansible -m vyos_facts -a 'gather_subset=all' spine01
```

- 2.2. Execute this ad hoc command to display the full set of facts available under **ios_facts**.

```
[student@workstation proj]$ ansible -m ios_facts -a 'gather_subset=all' cs01
```

- 3. Recognize and understand differences between fact subsets, and how to interpret and make use of the data returned by facts modules.

- 3.1. What differences did you notice in the results returned by the four commands?

- 3.1.1. All four commands return data in JSON format.

- 3.1.2. The **ios_facts** module returns interface data in both **default** mode and **all** facts mode, but interface data is not provided by **vyos_facts** even when the **all** subset is requested.

- 3.1.3. Both the **ios_facts** and **vyos_facts** modules return an **ansible_net_config** variable only when **gather_subset=all** is specified.

Running **ansible-doc ios_facts** and **ansible-doc vyos_facts** tells us this about **ansible_net_config** (**ansible-doc** returns the same information about **ansible_net_config** for both modules):

```
ansible_net_config:  
description: The current active config from the device  
returned: when config is configured  
type: string
```

- 3.2. Run **ansible-doc vyos_facts** and review the EXAMPLES section.

```
[student@workstation proj]$ ansible-doc vyos_facts  
...output omitted...  
EXAMPLES:  
- name: collect all facts from the device  
  vyos_facts:  
    gather_subset: all  
  
- name: collect only the config and default facts  
  vyos_facts:  
    gather_subset: config
```

```
- name: collect everything except the config
vyos_facts:
  gather_subset: "!config"
```

Now we know that **config** is defined as a subset, which makes it easy to either include or exclude the configuration data. The **ios_config** documentation mentions that the **ios_config** module has a subset named **hardware**, so we can gather or ignore hardware facts.

This illustrates the following:

- A great deal of data can be gathered by way of facts.
- The fact-gathering mechanism is flexible, with the ability to gather or not gather fact subsets based on your needs.
- The **ansible-doc** command provides information that helps to make sense of what facts are available, how to manage your access to facts, data types of facts, and details about the nature of specific ones.

► 4. Compose and run a playbook that displays specific facts.

- 4.1. Start by converting the **vyos** all facts ad hoc command into a playbook named **vyos-all-facts1.yml**. Set the value of **hosts** to **vyos**.

It is convenient to work with a single host while developing a playbook and becoming familiar with its behavior, and the **--limit** option makes it easy to do this. With this initial version of the playbook, the objective is simply to gather facts, register the data that is returned as a variable, and use the **debug** module to display what we have.

```
[student@workstation proj]$ cat vyos-all-facts1.yml
---
- name: gather all facts from a particular vyos machine
  hosts: vyos
  vars:
    subset: all

  tasks:

    - name: invoke vyos_facts with gather_subset=all
      vyos_facts:
        gather_subset: "{{ subset }}"

    - debug:
        var: ansible_facts
```

- 4.2. Run the **vyos-all-facts1.yml** playbook, using the **--limit** option (**-l**) to limit the play to **spine01**.

```
[student@workstation proj]$ ansible-playbook -l spine01 vyos-all-facts1.yml

PLAY [gather all facts from a particular vyos machine] ****
TASK [invoke vyos_facts with gather_subset=all] ****
ok: [spine01]
```

```
TASK [debug] ****
ok: [spine01] => {
  "ansible_facts": [
    "net_commits": [
      {
        "by": "root",
        "comment": null,
        "datetime": "2020-07-31 06:56:50",
      ...output omitted...
    PLAY RECAP ****
spine01 : ok=2    changed=0    unreachable=0    failed=0
```

- 4.3. Make a copy of **vyos-all-facts1.yml** named **vyos-some-facts1.yml** and use it for editing.

```
[student@workstation proj]$ cp vyos-all-facts1.yml vyos-some-facts1.yml
```

- 4.4. Edit the **vyos-some-facts1.yml** playbook. Change the subset that is gathered from **all** to **default**. When you ran the ad hoc command that displayed the default set of facts returned by **vyos_facts**, it listed in the output four facts: **ansible_net_hostname**, **ansible_net_model**, **ansible_net_serialnum**, and **ansible_net_version**. Include a debug task in **vyos-some-facts1.yml** that displays those four facts.

```
[student@workstation proj]$ cat vyos-some-facts1.yml
---
- name: gather some facts from a particular vyos machine
  hosts: vyos
  vars:
    subset: default

  tasks:

    - name: invoke vyos_facts with gather_subset=all
      vyos_facts:
        gather_subset: "{{ subset }}"

    - debug:
        msg:
          - "Host name: {{ ansible_net_hostname }}"
          - "Model: {{ ansible_net_model }}"
          - "Version: {{ ansible_net_version }}"
          - "SerialNum: {{ ansible_net_serialnum }}"
```

- 4.5. Execute the **ansible-playbook** command to perform the play in **vyos-some-facts1.yml** playbook.

```
[student@workstation proj]$ ansible-playbook -l spine01 vyos-some-facts1.yml

PLAY [gather some facts from a particular vyos machine] ****
TASK [invoke vyos_facts with gather_subset=all] ****
ok: [spine01]
```

```
TASK [debug] *****
ok: [spine01] => {
    "msg": [
        "Host name: vyos",
        "Model: OpenStack",
        "Version: VyOS",
        "SerialNum: 00000000-0000-0000-0000-ac1f6bc7ee96"
    ]
}

PLAY RECAP *****
spine01 : ok=2    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

► Guided Exercise

Viewing System Settings

In this exercise, you will view system settings for network devices.

Outcomes

You should be able to:

- View individual system settings using ad hoc commands.
- View multiple system settings with a simple playbook.
- View multiple system settings using a multivendor playbook.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Use individual ad hoc commands to view the host name, domain, and name servers for **spine01**, which is a VyOS device.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" spine01
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host domain'" spine01
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh dns forwarding nameservers'" spine01
```

- 2. Use individual ad hoc commands to view the host name, domain, and name servers for **cs01**, which is an IOS device.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include hostname'" cs01
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include ip domain name'" cs01
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
```

- 3. Compose a playbook named **host-dnsinfo1.yml** that displays the host name, domain name, and name servers for **cs01**.

```
[student@workstation proj]$ cat host-dnsinfo1.yml
---
- name: display host name, domain name, nameservers for ios devices
  hosts: ios
  tasks:
```

```

- name: run the commands
  ios_command:
    commands:
      - sh run | include hostname
      - sh run | include ip domain name
      - sh run | include ip name-server
    register: result

- debug:
  var: result.stdout

```

Alternatively, you could define a variable of type sequence that contains the distinctive portion of the **show run | include** commands, and loop over that.

```

[student@workstation proj]$ cat host-dnsinfo1.yml
---
- name: display host name, domain name, nameservers for ios devices
  hosts: ios
  vars:
    includes:
      - hostname
      - domain name
      - name-server

  tasks:
    - name: run the commands
      ios_command:
        commands:
          - sh run | include {{ item }}
      loop: "{{ includes }}"
      register: result

    - debug:
      var: item.stdout
      loop: "{{ result.results }}"

```

- ▶ 4. Perform the play in the playbook you just created.

```

[student@workstation proj]$ ansible-playbook host-dnsinfo1.yml

PLAY [display host name, domain name, nameservers for ios devices] ****
TASK [run the commands] ****
ok: [cs01]

TASK [debug] ****
ok: [cs01] => {
    "result.stdout": [
        "hostname cs01",
        "ip domain name lab.example.com",
        ""
    ]
}

```

```
}
```

```
PLAY RECAP ****
cs01 : ok=2    changed=0   unreachable=0   failed=0
```

- 5. Compose a playbook named **multi-vendor-host-dnsinfo1.yml** that displays the host name, domain name, and name servers for both VyOS and IOS devices, and which defaults to hosts group network, and can target an individual host when the **ansible-playbook** command is used to set the variable **target** with the **-e** option.

You can copy the playbook from step 3 to save typing.

```
[student@workstation proj]$ cat multi-vendor-host-dnsinfo1.yml
---
- name: multi-vendor play to display host name, domain name, nameservers
  hosts: network

  tasks:

    - name: ios host, domain, nameserver
      ios_command:
        commands:
          - sh run | include hostname
          - sh run | include ip domain name
          - sh run | include ip name-server
      register: result
      when: ansible_network_os == 'ios'

    - name: display ios host settings
      debug:
        var: result.stdout
      when: ansible_network_os == 'ios'

    - name: vyos host, domain, nameserver
      vyos_command:
        commands:
          - sh host name
          - sh host domain
          - sh config | grep name-server
      register: result
      when: ansible_network_os == 'vyos'

    - name: display vyos host settings
      debug:
        var: result.stdout
      when: ansible_network_os == 'vyos'
```

Note that the same alternative approach with loops can also be employed here.

- 6. Perform the play contained in your new playbook. First use the limit option (**-l SUBSET**) to specify **cs01**, then run it in default mode.

```
[student@workstation proj]$ ansible-playbook -l cs01 \
> multi-vendor-host-dnsinfo1.yml

PLAY [multi-vendor play to display host name, domain name, nameservers] *****

TASK [ios host, domain, nameserver] *****
ok: [cs01]

TASK [display ios host settings] *****
ok: [cs01] => {
    "result.stdout": [
        "hostname cs01",
        "ip domain name lab.example.com",
        ""
    ]
}

TASK [vyos host, domain, nameserver] *****
skipping: [cs01]

TASK [display vyos host settings] *****
skipping: [cs01]

PLAY RECAP *****
cs01 : ok=2    changed=0    unreachable=0    failed=0

[student@workstation proj]$ ansible-playbook multi-vendor-host-dnsinfo1.yml
```

This concludes the guided exercise.

Configuring Network Devices

Objectives

After completing this section, you should be able to:

- Back up network device configurations.
- Implement Move/Add/Change/Delete (MACD) changes using Ansible **os_config** modules.

Backing Up Device Configurations

Ansible ***os_config** modules make it easy to back up network device configurations:

- Setting the **backup** argument to **yes** causes the ***os_config** module to create a full backup of the current running configuration from the remote device.
- The backup file is written to the **backup/** directory in the playbook root directory or role root directory, if the playbook is part of an Ansible Role. If the directory does not exist, it is created.
- Backup files are named using this format:

<inventory_hostname> _config.yyyy-mm-dd@HH:MM:SS.

- The file format of the backed up running configuration is platform dependent: an **ios_config** backup generates a text file containing an IOS running configuration, a **vyos_config** backup generates a text file containing a VyOS running configuration, and so forth.

Backing up Configurations on Demand

Ad hoc commands can be used to back up running configurations on an on-demand basis.

An ad hoc command using the **ios_config** module:

```
$ ansible -i inventory --ask-vault-pass -m ios_config -a 'backup=yes' cs01
```

The same example using the **vyos_config** module:

```
$ ansible -i inventory --ask-vault-pass -m vyos_config -a 'backup=yes' spine01
```

Backing Up Configurations With Playbooks

Playbooks can drive the act of backing up network device configurations. It is often useful to have a play in which a task that modifies the configuration is preceded by a task that backs up the running configuration before the change.

A play with a task that uses the **vyos_config** module:

```
- hosts: vyos
  tasks:
    - name: Backup VyOS running config
      vyos_config:
        backup: yes
```

A play with a task that uses the **ios_config** module:

```
- hosts: ios
  tasks:
    - name: Backup IOS running config
      ios_config:
        backup: yes
```

► Guided Exercise

Backing Up Network Device Configurations

Backing up the current, known good running configuration on a regular basis is an important part of documenting your network. Backing up configurations before and after changes is often an integral part of the change control process for production networks.

In this exercise, you will back up network device configurations.

Outcomes

You should be able to back up the configuration of a network device.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Create a playbook named `multi-vendor-backup3.yml` that defaults to backing up configurations for all of the devices belonging to the **network** group. Remember that you can also use the `--limit=SUBSET (-l SUBSET)` option to limit the target hosts to a particular group or host, as long as that group or host is a subset of the group or hosts specified in the **hosts** field of the playbook.

```
[student@workstation proj]$ cat multi-vendor-backup3.yml
---
- name: back up network device configurations
  hosts: network

  tasks:
    - name: backup config of vyos device
      vyos_config:
        backup: yes
      when: ansible_network_os == 'vyos'

    - name: backup config of ios device
      ios_config:
        backup: yes
      when: ansible_network_os == 'ios'
```

- 2. Perform the play in your playbook.

```
[student@workstation proj]$ ansible-playbook multi-vendor-backup3.yml
```

- 3. Verify that device configurations were successfully backed up.

```
[student@workstation proj]$ ls -l backup/
total 20
-rw-rw-r-- 1 student student 1768 Aug  3 14:15 cs01_config.2020-08-03@14:15:52
-rw-rw-r-- 1 student student 2890 Aug  3 14:15 leaf01_config.2020-08-03@14:15:51
-rw-rw-r-- 1 student student 2903 Aug  3 14:15 leaf02_config.2020-08-03@14:15:51
-rw-rw-r-- 1 student student 2883 Aug  3 14:15 spine01_config.2020-08-03@14:15:51
-rw-rw-r-- 1 student student 2892 Aug  3 14:15 spine02_config.2020-08-03@14:15:51
[student@workstation proj]$ head -n 6 backup/*
==> backup/cs01_config.2020-08-03@14:15:52 <==
Building configuration...

Current configuration : 1708 bytes
!
! Last configuration change at 14:51:02 UTC Mon Aug 3 2020
!

==> backup/leaf01_config.2020-08-03@14:15:51 <=
set interfaces ethernet eth0 address '172.25.250.151/24'
set interfaces ethernet eth0 duplex 'auto'
set interfaces ethernet eth0 hw-id '2c:c2:60:3c:26:65'
set interfaces ethernet eth0 smp_affinity 'auto'
set interfaces ethernet eth0 speed 'auto'
set interfaces ethernet eth1 address '10.10.10.1/30'

==> backup/leaf02_config.2020-08-03@14:15:51 <=
set interfaces ethernet eth0 address '172.25.250.161/24'
set interfaces ethernet eth0 duplex 'auto'
set interfaces ethernet eth0 hw-id '2c:c2:60:6b:20:19'
set interfaces ethernet eth0 smp_affinity 'auto'
set interfaces ethernet eth0 speed 'auto'
set interfaces ethernet eth1 duplex 'auto'
...output omitted...
```

This concludes the guided exercise.

Configuring the Host Name

Objectives

After completing this section, you should be able to use Ansible to configure the host name on Cisco IOS and VyOS network devices.

Configuring Network Devices

The same family of Ansible modules is used to back up running configurations and to modify the configuration of network devices: ***os_config**.

This makes it easy to back up the running configuration immediately prior to modifying the configuration.

► Guided Exercise

Configuring the Host Name

In this exercise, you will configure the host name for network devices.

Outcomes

You should be able to:

- View the host name using an ad hoc command.
- Configure the host name using an ad hoc command.
- Configure the host name with a simple playbook.
- Configure the host name using a multivendor playbook.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Execute an ad hoc command that displays the host name of VyOS device **leaf02**.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" leaf02
leaf02 | SUCCESS => {
  "changed": false,
  "stdout": [
    "vyos"
  ],
  "stdout_lines": [
    [
      "vyos"
    ]
  ]
}
```

- 2. Execute an ad hoc command that uses the **vyos_system** module to set the host name for **leaf02**.

```
[student@workstation proj]$ ansible -m vyos_system \
> -a "host_name={{ inventory_hostname }}" leaf02
leaf02 | SUCCESS => {
  "changed": true,
  "commands": [
    "set system host-name 'leaf02'"
  ]
}
```

- 3. Execute an ad hoc command to verify that the host name was changed as expected.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" leaf02
leaf02 | SUCCESS => {
  "changed": false,
  "stdout": [
    "leaf02"
  ],
  "stdout_lines": [
    [
      "leaf02"
    ]
  ]
}
```

- ▶ 4. Repeat the ad hoc command that sets the host name, but this time explicitly set it back to its original value.

```
[student@workstation proj]$ ansible -m vyos_system \
> -a "host_name=vyos" leaf02
leaf02 | SUCCESS => {
  "changed": true,
  "commands": [
    "set system host-name 'vyos'"
  ]
}
```

- ▶ 5. Create a playbook named **multi-vendor-set-hostname1.yml** that sets the host name to **inventory_hostname**. The value of **hosts** should default to **network**, and the **when** conditional should be used in order to invoke either **vyos_system** or **ios_system**.

```
[student@workstation proj]$ cat multi-vendor-set-hostname1.yml
---
- name: sets hostname in a multi-vendor way
  hosts: network

  tasks:

  - name: set hostname on vyos device
    vyos_system:
      host_name: "{{ inventory_hostname }}"
    when: ansible_network_os == 'vyos'

  - name: set hostname on ios device
    ios_system:
      hostname: "{{ inventory_hostname }}"
    when: ansible_network_os == 'ios'
```

- 6. Perform the play in the playbook you created.

```
[student@workstation proj]$ ansible-playbook -l leaf02 \
> multi-vendor-set-hostname1.yml
```

- 7. Execute an ad hoc command to confirm that the play did in fact succeed in changing the host name.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" leaf02
```

- 8. Execute an ad hoc command to return **leaf02**'s host name to its original value.

```
[student@workstation proj]$ ansible -m vyos_system -a "host_name=vyos" leaf02
leaf02 | SUCCESS => {
    "changed": true,
    "commands": [
        "set system host-name 'vyos'"
    ]
}
```

This concludes the guided exercise.

Configuring System Settings

Objectives

After completing this section, you should be able to make basic configuration settings such as the DNS name servers or domain name on Cisco IOS and VyOS network devices.

Contextualizing Configuration Statements

By default, configuration statements apply to the top level or global command context. The **parents** keyword applies a command or set of command lines in the context of a parent command.

The following task sets commands at the global level:

```
- name: configure top level commands
  ios_config:
    lines:
      - hostname {{ inventory_hostname }}
      - domain-name {{ domain_name }}
      - ip name-server {{ nameserver1 }}
      - username {{ ansible_user }} privilege 15 secret {{ ansible_ssh_pass }}
```

The following task applies commands within the context of a given interface:

```
- name: configure an interface
  ios_config:
    lines:
      - switchport access vlan {{ interface.vlan }}
      - description: {{ interface.description }}
      - no shutdown
  parents: interface {{ interface.name }}
```

► Guided Exercise

Configuring System Settings

In this exercise, you will configure system settings for network devices.

Outcomes

You should be able to:

- Execute an ad hoc command that configures the name-server setting on an IOS device.
- Perform a play that sets name servers on IOS devices.
- Perform a play that sets name servers using a multivendor playbook.
- Perform a play that sets host names, domain names, and name servers using a multivendor playbook.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

► 1. Execute an ad hoc command to configure the name server on **cs01**.

- 1.1. Execute an ad hoc command to display the current name-server settings for **cs01**.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
cs01 | SUCCESS => {
  "changed": false,
  "stdout": [
    ""
  ],
  "stdout_lines": [
    [
      ""
    ]
  ]
}
```

- 1.2. Execute an ad hoc command that sets **8.8.8.8** as a name server on **cs01**.

```
[student@workstation proj]$ ansible -m ios_system -a "name_servers=8.8.8.8" cs01
cs01 | SUCCESS => {
  "changed": true,
  "commands": [
    "ip name-server 8.8.8.8"
  ]
}
```

- 1.3. Execute an ad hoc command to display the new name-server settings on **cs01**.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
cs01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "ip name-server 8.8.8.8"
  ],
  "stdout_lines": [
    [
      "ip name-server 8.8.8.8"
    ]
  ]
}
```

▶ 2. Perform a play that sets name servers on **IOS** devices.

- 2.1. Compose a playbook named **ios-nameservers1.yml** that sets the name servers on members of the **ios** host group to **8.8.8.8** and **8.8.4.4**.

```
[student@workstation proj]$ cat ios-nameservers1.yml
---
- name: sets nameservers on ios device
  hosts: ios
  vars:
    nameservers:
    - 8.8.8.8
    - 8.8.4.4

  tasks:
    - name: set nameservers
      ios_system:
        name_servers: "{{ nameservers }}"
```

2.2. Perform the play found in your new playbook.

```
[student@workstation proj]$ ansible-playbook ios-nameservers1.yml
```

- 2.3. Execute an ad hoc command to display the updated name-server settings on **cs01**.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
cs01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "ip name-server 8.8.8.8 8.8.4.4"
  ],
  "stdout_lines": [
    [
      "ip name-server 8.8.8.8 8.8.4.4"
    ]
  ]
}
```

```

        ]
    ]
}
```

- 3. Perform a play that configures name servers on VyOS and IOS devices according to a group variable.

- 3.1. Add a list variable named **nameservers** to the **group_vars/network/vars.yml** file with **8.8.8.8** and **8.8.4.4** as list elements:

```
[student@workstation proj]$ cat group_vars/network/vars.yml
ansible_connection: network_cli
nameservers:
- 8.8.8.8
- 8.8.4.4
```

- 3.2. Compose a multivendor playbook named **multi-vendor-nameservers1.yml** that sets the name servers on members of the **network** host group according to items in the name servers list variable contained in the **group_vars/network/vars.yml** file. Make it use the network host group by default, but make it possible to apply the play to a particular target if the **-e** option is used with the **target** variable in the **ansible-playbook** command.

```
[student@workstation proj]$ cat multi-vendor-nameservers1.yml
---
- name: sets nameservers on devices
  hosts: network

  tasks:
    - name: set nameservers on ios devices
      ios_system:
        name_servers: "{{ nameservers }}"
      when: ansible_network_os == 'ios'

    - name: set nameservers on vyos devices
      vyos_system:
        name_servers: "{{ nameservers }}"
      when: ansible_network_os == 'vyos'
```

- 3.3. Perform the play found in your new playbook, applying it to the **spine02** device.

- 3.3.1. Execute an ad hoc command to display the **spine02**'s name servers before performing the play.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh config | grep name-server'" spine02
spine02 | SUCCESS => {
  "changed": false,
  "stdout": [
    ""
  ],
  "stdout_lines": [
```

```
[  
    "  
]  
}  
}
```

3.3.2. Perform the play. Limit it to **spine02**.

```
[student@workstation proj]$ ansible-playbook -l spine02 \  
> multi-vendor-nameservers1.yml
```

3.3.3. Display the **spine02** name servers after performing the play.

```
[student@workstation proj]$ ansible -m vyos_command \  
> -a "commands='sh config | grep name-server'" spine02  
spine02 | SUCCESS => {  
    "changed": false,  
    "stdout": [  
        "name-server 8.8.8.8\n      name-server 8.8.4.4"  
    ],  
    "stdout_lines": [  
        [  
            "name-server 8.8.8.8",  
            "      name-server 8.8.4.4"  
        ]  
    ]  
}
```

- ▶ 4. Perform a play that sets host name, domain name, and name servers using a multivendor playbook.

- 4.1. Add a **domain_name** variable to the **group_vars/network/vars.yml** file.

```
[student@workstation proj]$ cat group_vars/network/vars.yml  
ansible_connection: network_cli  
domain_name: lab.example.com  
nameservers:  
- 8.8.8.8  
- 8.8.4.4
```

- 4.2. Compose a playbook named **multi-vendor-host-domain-ns1.yml** that sets the host name, domain name, and name servers.

```
[student@workstation proj]$ cat multi-vendor-host-domain-ns1.yml  
---  
- name: sets host name, domain name, nameservers  
  hosts: network  
  
  tasks:  
  
  # --- host name ---  
  - name: set host name on vyos device
```

```

vyos_system:
  host_name: "{{ inventory_hostname }}"
when: ansible_network_os == 'vyos'

- name: set host name on ios device
  ios_system:
    hostname: "{{ inventory_hostname }}"
when: ansible_network_os == 'ios'

# --- domain name ---
- name: set domain name on vyos device
  vyos_system:
    domain_name: "{{ domain_name }}"
when: ansible_network_os == 'vyos'

- name: set domain name on ios device
  ios_system:
    domain_name: "{{ domain_name }}"
when: ansible_network_os == 'ios'

# --- nameserver ---
- name: set nameservers on ios devices
  ios_system:
    name_servers: "{{ nameservers }}"
when: ansible_network_os == 'ios'

- name: set nameservers on vyos devices
  vyos_system:
    name_servers: "{{ nameservers }}"
when: ansible_network_os == 'vyos'

```

4.3. Verify that the playbook syntax is valid.

```
[student@workstation proj]$ ansible-playbook --syntax-check \
> multi-vendor-host-domain-ns1.yml
```

```
Playbook: multi-vendor-host-domain-ns1.yml
```

4.4. Inspect the host name, domain name, and name servers on **leaf02** by performing the play found in the **multi-vendor-host-dnsinfo1.yml** playbook you created in *Guided Exercise: Backing Up Network Device Configurations*. Limit the hosts to **leaf02**.

```
[student@workstation proj]$ ansible-playbook -l leaf02 \
> multi-vendor-host-dnsinfo1.yml

PLAY [multi-vendor play to display host name, domain name, nameservers] *****

TASK [ios host, domain, nameserver] *****
skipping: [leaf02]

TASK [display ios host settings] *****
skipping: [leaf02]
```

```
TASK [vyos host, domain, nameserver] *****
ok: [leaf02]

TASK [display vyos host settings] *****
ok: [leaf02] => {
  "result.stdout": [
    "vyos",
    "",
    ""
  ]
}

PLAY RECAP *****
leaf02 : ok=2    changed=0   unreachable=0   failed=0
```

- 4.5. Perform the play found in your newly created playbook, **multi-vendor-host-domain-ns1.yml**, applying it only to **leaf02**.

```
[student@workstation proj]$ ansible-playbook -l leaf02 \
> multi-vendor-host-domain-ns1.yml
```

- 4.6. Inspect the state of **leaf02** after applying the multivendor change, again using **multi-vendor-host-dnsinfo1.yml** for that purpose.

```
[student@workstation proj]$ ansible-playbook -l leaf02 \
> multi-vendor-host-dnsinfo1.yml

PLAY [multi-vendor play to display host name, domain name, nameservers] *****

TASK [ios host, domain, nameserver] *****
skipping: [leaf02]

TASK [display ios host settings] *****
skipping: [leaf02]

TASK [vyos host, domain, nameserver] *****
ok: [leaf02]

TASK [display vyos host settings] *****
ok: [leaf02] => {
  "result.stdout": [
    "leaf02",
    "lab.example.com",
    "name-server 8.8.8.8\n      name-server 8.8.4.4"
  ]
}

PLAY RECAP *****
leaf02 : ok=2    changed=0   unreachable=0   failed=0
```

This concludes the guided exercise.

Generating Configuration Settings from Jinja2 Templates

Objectives

After completing this section, you should be able to use Ansible to apply configuration directives that have been generated from a Jinja2 template to a network device.

Updating With Other Modules

As a general rule, an ***os_config** module is the best choice for applying changes to devices. But there are exceptions; sometimes a special purpose module exists.

The **ios_banner** module gracefully handles multiline banners.

```
---
- name: play for setting banner motd on ios devices
  hosts: leaf1
  gather_facts: False

  tasks:
    - name: set banner motd
      ios_banner:
        banner: motd
        test: |
          Unauthorized access to this device is prohibited
        state: present
```

Sourcing Statements From a File

Configuration statements can be embedded directly within playbooks:

```
---
- hosts: ios
  tasks:
    - ios_config:
        lines: hostname "{{ inventory_hostname }}"
```

Configuration statements can also be sourced from a local file:

```
---
- name: A playbook that uses a static file to configure Cisco network devices
  hosts: ios

  tasks:
    - set_fact:
        config_filepath: "{{ playbook_dir }}/configs"
```

```
- name: configure using static config file
  ios_config:
    src "{{ config_filepath }}/{{ inventory_hostname }}.cfg
```

Generating Configurations With Templates

Jinja2 templates can be used to generate the files sourced to configure devices.

```
---
- name: A play that uses a Jinja2 template to generate a set of config files
  hosts: ios
  connection: local

  tasks:
    - template:
        src: ios-leaf-common.j2
        dest: "{{ config_filepath }}/{{ inventory_hostname }}.cfg"

- name: A play that applies configs generated above to network devices
  hosts: ios

  tasks:
    - name: configure using static config file
      ios_config:
        src: "{{ config_filepath }}/{{ inventory_hostname }}.cfg

    - name: save running to startup when modified
      ios_config:
        save_when: modified
```

Configuring Directly From Templates

Ansible ***os_config** modules are designed to support Jinja2 templates directly, so you can generate configurations and apply them in with a single task.

```
---
- name: A play that generates configs and applies them in one step
  hosts: ios

  tasks:
    - name: configure using static config file
      ios_config:
        src: ios-leaf.j2

    - name: save running to startup when modified
      ios_config:
        save_when: modified
```

► Guided Exercise

Configuring From Templates

In this exercise, you will configure a network device using a Jinja2 template.

Outcomes

You should be able to:

- View the host name using an ad hoc command.
- Add a variable and its value to support template-based configuration.
- Create a Jinja2 template that builds configuration statements.
- Perform a play that configures a device from a Jinja2 template.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Execute an ad hoc command that displays the host name of VyOS device **spine02**.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" spine02
spine02 | SUCCESS => {
    "changed": false,
    "stdout": [
        "vyos"
    ],
    "stdout_lines": [
        [
            "vyos"
        ]
    ]
}
```

- 2. Add a variable and its value to support template-based configuration.

- 2.1. Create the **host_vars/spine02/** directory if it does not already exist.

```
[student@workstation proj]$ mkdir -p host_vars/spine02
```

- 2.2. Create a host variables file named **host_vars/spine02/vars.yml**. This file should set a variable named **new_hostname** to the value **spineX1**.

```
[student@workstation proj]$ cat host_vars/spine02/vars.yml
new_hostname: spineX1
```

- 3. Create a Jinja2 template that builds configuration statements.

- 3.1. Create a templates directory named **j2/** if one does not already exist.

```
[student@workstation proj]$ mkdir -p j2
```

- 3.2. Create a Jinja2 template file named **j2/vyos-configure.j2**. This file should map variables to configuration statements.

```
[student@workstation proj]$ cat j2/vyos-configure.j2
set system host-name {{ new_hostname }}
```

- 4. Perform a play that configures a device from a Jinja2 template.

- 4.1. Compose a playbook named **j2cfg-spine02.yml**. It should contain a single task that uses the **vyos_config** module. It should source the configuration lines used by **vyos_config** from your Jinja2 template.

```
[student@workstation proj]$ cat j2cfg-spine02.yml
---
- name: play that configures spine02 using j2 template
  hosts: spine02
  vars:
    j2_template: j2/vyos-configure.j2
  tasks:
    - name: configure {{ inventory_hostname }}
      vyos_config:
        src: "{{ j2_template }}"
```

- 4.2. Check the new playbook's syntax.

```
[student@workstation proj]$ ansible-playbook --syntax-check j2cfg-spine02.yml
```

- 4.3. Perform the play in **j2cfg-spine02.yml**.

```
[student@workstation proj]$ ansible-playbook j2cfg-spine02.yml
```

- 5. Confirm that the change occurred as desired, then change the host name to its original value.

- 5.1. Execute an ad hoc command to confirm that the change occurred as expected.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" spine02
spine02 | SUCCESS => {
  "changed": false,
  "stdout": [
    "spineX1"
  ],
  "stdout_lines": [
    "spineX1"
```

```
        ]  
    ]  
}
```

- 5.2. Execute an ad hoc command to change the host name to its original value.

```
[student@workstation proj]$ ansible -m vyos_system -a "host_name=vyos" spine02  
spine02 | SUCCESS => {  
    "changed": true,  
    "commands": [  
        "set system host-name 'vyos'"  
    ]  
}
```

This concludes the guided exercise.

► Guided Exercise

Configuring Settings From Templates

This guided exercise illustrates how you can start with a simple Jinja2 template and add onto that to compose more complicated sequences of configuration statements. In the previous exercise, you set the local host name of a device. In this exercise, you add configuration statements that set the domain name and name servers.

In this exercise, you will configure system settings for network devices from templates.

Outcomes

You should be able to:

- Set variables or confirm that variables are already set to support template-based configuration.
- Create or update Jinja2 templates that build configuration statements.
- Perform a multivendor play that configures devices from Jinja2 templates.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Set variables or confirm that variables are already set to support template-based configuration. Verify that the **domain_name** and **nameservers** variables are set as indicated below in the `group_vars/network/vars.yml` file.

```
[student@workstation proj]$ cat group_vars/network/vars.yml
ansible_connection: network_cli
domain_name: lab.example.com
nameservers:
- 8.8.8.8
- 8.8.4.4
```

- 2. Create or update Jinja2 templates that build configuration statements.

- 2.1. Create a Jinja2 template named **j2/vyos-config.j2** that contains the following:

```
set system host-name {{ inventory_hostname }}
set system domain-name {{ domain_name }}
{% for nameserver in nameservers %}
set system name-server {{ nameserver }}
{% endfor %}
```

- 2.2. Create a Jinja2 template named **j2/ios-config.j2** that contains the following:

```
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
{% for nameserver in nameservers %}
ip name-server {{ nameserver }}
{% endfor %}
```

► 3. Perform a multivendor play that configures devices from Jinja2 templates.

- 3.1. Compose a multivendor playbook named **j2cfg.yml** that sources configuration statements from the vendor-specific Jinja2 templates you created. The names of the template files were made variables here to put them at the top of the play to make it easy to change the values in case they are ever renamed or moved. This also makes it possible to reuse the text of the task blocks in other playbooks.

```
[student@workstation proj]$ cat j2cfg.yml
---
- name: configure devices using j2 templates
  hosts: network
  vars:
    vyos_template: j2/vyos-config.j2
    ios_template: j2/ios-config.j2

  tasks:
    - name: configure {{ inventory_hostname }}
      vyos_config:
        src: "{{ vyos_template }}"
      when: ansible_network_os == 'vyos'

    - name: configure {{ inventory_hostname }}
      ios_config:
        src: "{{ ios_template }}"
      when: ansible_network_os == 'ios'
```

- 3.2. Perform the play found in your new playbook, limiting the change to **spine01**.

```
[student@workstation proj]$ ansible-playbook -l spine01 j2cfg.yml
```

- 3.3. Execute ad hoc commands or a play from an existing playbook to verify that the result is as desired.

```
[student@workstation proj]$ ansible-playbook -l spine01 \
> multi-vendor-host-dnsinfo1.yml

PLAY [multi-vendor play to display host name, domain name, nameservers] *****

TASK [ios host, domain, nameserver] *****
skipping: [spine01]

TASK [display ios host settings] *****
skipping: [spine01]
```

```
TASK [vyos host, domain, nameserver] *****
ok: [spine01]

TASK [display vyos host settings] *****
ok: [spine01] => {
    "result.stdout": [
        "spine01",
        "lab.example.com",
        "name-server 8.8.8.8\\n      name-server 8.8.4.4"
    ]
}

PLAY RECAP *****
spine01 : ok=2    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

Enabling and Disabling Interfaces

Objectives

After completing this section, you should be able to use Ansible to enable and disable interfaces on a network device.

Selecting an Interface Module

What are the differences between the ***os_interface** and ***os_l3_interface** modules?

- The ***os_l3_interface** module:
 - Manages IPv4 and IPv6 layer 3 addresses; it maps layer 3 addresses to interfaces.
 - Can map IPv4 or IPv6 addresses to interfaces explicitly; it can also map both IPv4 and IPv6 explicitly by field name at the same time.
 - Supports the aggregate argument, which takes as an argument a dictionary with name and layer 3 address fields.
- The ***os_interface module** is used for managing all other aspects of network interfaces, including descriptions and enabling and disabling interfaces.

► Guided Exercise

Bouncing an Interface

Shutting down and bringing back up a port or interface is a procedure that is routinely used to reinitialize it on a network device.

Outcomes

You should be able to:

- Execute an ad hoc command that displays the current state of an interface.
- Perform a play that shuts down an interface.
- Perform a play that brings up an interface.
- Perform a play that bounces an interface (shut/no shut).

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Execute an ad hoc command that displays the current state of interface **eth6** on the **leaf01** machine.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh int eth eth6|grep ^eth6'" leaf01
leaf01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "eth6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000"
    ],
    "stdout_lines": [
        [
            "eth6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000"
        ]
    ]
}
```

- 2. Perform a play that shuts down **eth6** on **leaf01**.

- 2.1. Compose a playbook named **ifdown.yml** with required extra variables named **target** and **intf** that shuts down the interface on the host.

```
[student@workstation proj]$ cat ifdown.yml
---
- name: play to shutdown an interface
  # Required arguments: target, intf
  hosts: "{{ target }}"
```

```

tasks:

- name: disable interface {{ intf }} on a VyOS box
  vyos_interface:
    name: "{{ intf }}"
    enabled: False
  when: ansible_network_os == 'vyos'

- name: disable interface {{ intf }} on an IOS box
  ios_interface:
    name: "{{ intf }}"
    enabled: False
  when: ansible_network_os == 'ios'

```

2.2. Perform the play in the playbook.

```
[student@workstation proj]$ ansible-playbook -e "target=leaf01 intf=eth6" \
> ifdown.yml
```

2.3. Display the state of **eth6** on **leaf01** after performing the play.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh int eth eth6|grep ^eth6'" leaf01
leaf01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "eth6: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast
state DOWN group default qlen 1000"
  ],
  "stdout_lines": [
    [
      "eth6: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast
state DOWN group default qlen 1000"
    ]
  ]
}
```

► 3. Perform a play that brings up interface **eth6** on **leaf01**.

3.1. Compose a playbook named **ifup.yml** with required extra variables named **target** and **intf** that brings up the specified interface on the specified host.

```
[student@workstation proj]$ cat ifup.yml
---
- name: bring up an interface
  # Required arguments: target, intf
  hosts: "{{ target }}"

  tasks:
    - name: enable interface {{ intf }} on a VyOS box
      vyos_interface:
```

```
name: "{{ intf }}"
enabled: True
when: ansible_network_os == 'vyos'

- name: enable interface {{ intf }} on an IOS box
  ios_interface:
    name: "{{ intf }}"
    enabled: True
  when: ansible_network_os == 'ios'
```

- 3.2. Perform the play in the playbook.

```
[student@workstation proj]$ ansible-playbook -e "target=leaf01 intf=eth6" \
> ifup.yml
```

- 3.3. Display the state of **eth6** on **leaf01** after performing the play.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh int eth eth6|grep ^eth6'" leaf01
leaf01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "eth6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000"
  ],
  "stdout_lines": [
    [
      "eth6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000"
    ]
  ]
}
```

- 4. Perform a play that bounces an interface.

- 4.1. Compose a playbook named **bounce-if.yml** containing a play that first shuts down an interface, then brings it up.

```
[student@workstation proj]$ cat bounce-if.yml
---
- name: bounce an interface (shut/no shut)
  # Required arguments: target, intf
  hosts: "{{ target }}"

  tasks:

  - name: shut interface {{ intf }} on a VyOS box
    vyos_interface:
      name: "{{ intf }}"
      enabled: False
    when: target in groups['vyos']

  - name: shut interface {{ intf }} on an IOS box
```

```

ios_interface:
  name: "{{ intf }}"
  enabled: False
when: target in groups['ios']

- name: pause briefly before enabling interfaces
  pause:
    seconds: 1

- name: enable interface {{ intf }} on a VyOS box
  vyos_interface:
    name: "{{ intf }}"
    enabled: True
when: target in groups['vyos']

- name: enable interface {{ intf }} on an IOS box
  ios_interface:
    name: "{{ intf }}"
    enabled: True
when: target in groups['ios']

```

- 4.2. Check the playbook's syntax, then perform its play, applying it to the **eth6** interface on **leaf01**.

```

[student@workstation proj]$ ansible-playbook -e "target=leaf01 intf=eth6" \
> --syntax-check bounce-if.yml

Playbook: bounce-if.yml
[student@workstation proj]$ ansible-playbook -e "target=leaf01 intf=eth6" \
> bounce-if.yml

PLAY [bounce an interface (shut/no shut)] ****
TASK [shut interface eth6 on a VyOS box] ****
changed: [leaf01]

TASK [shut interface eth6 on an IOS box] ****
skipping: [leaf01]

TASK [pause briefly before enabling interfaces] ****
Pausing for 1 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [leaf01]

TASK [enable interface eth6 on a VyOS box] ****
changed: [leaf01]

TASK [enable interface eth6 on an IOS box] ****
skipping: [leaf01]

PLAY RECAP ****
leaf01 : ok=3    changed=2    unreachable=0    failed=0

```

- 4.3. Display the state of **eth6** on **leaf01** after performing the play.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh int eth eth6|grep ^eth6'" leaf01
leaf01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "eth6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000"
    ],
    "stdout_lines": [
        [
            "eth6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000"
        ]
    ]
}
```

This concludes the guided exercise.

► Guided Exercise

Bouncing Specified IOS Interfaces

When you know in advance which interfaces you would like to bounce, you can bounce multiple interfaces based on a list provided by way of a variable that is set in a **vars** file.

In this exercise, you will automate the process of bouncing specified non-management interfaces on an IOS device.

Outcomes

You should be able to:

- Compose NOS-specific device configuration templates to provide parameterized configuration statements.
- Perform a play that bounces all interfaces on an IOS device.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Compose NOS-specific device configuration templates to provide parameterized configuration statements.
- 1.1. Create a **vars/** directory if it does not already exist.

```
[student@workstation proj]$ mkdir -p vars
```

- 1.2. Download the **layer3-branch-corp-cloud.yml** file into the **vars/** directory.
This file maps IPv4 and IPv6 protocol data to interfaces.

```
[student@workstation proj]$ cd vars
[student@workstation vars]$ wget \
> http://materials.example.com/full/vars/layer3-branch-corp-cloud.yml
[student@workstation vars]$ cd ..
[student@workstation proj]$
```

- 1.3. Create a Jinja2 template file named **j2/ios-if-shut-all.j2** that contains the following:

```
{% for intf in interface_data[inventory_hostname] %}
{% if not intf.name.startswith('Loopback') %}
interface {{ intf.name }}
    shutdown
{% endif %}
{% endfor %}
```

- 1.4. Create a Jinja2 template file named **j2/ios-if-noshut-all.j2** that contains the following:

```
{% for intf in interface_data[inventory_hostname] %}
{% if not intf.name.startswith('Loopback') %}
interface {{ intf.name }}
  no shutdown
{% endif %}
{% endfor %}
```

- 2. Perform a play that bounces all interfaces on an IOS device.

- 2.1. Compose a playbook named **ios-bounce-if-all.yml** that uses the templates.

```
[student@workstation proj]$ cat ios-bounce-if-all.yml
---
- name: bounce all interfaces on ios devices
  hosts: ios
  vars:
    shut_template: j2/ios-if-shut-all.j2
    no_shut_template: j2/ios-if-noshut-all.j2
  vars_files:
    - vars/layer3-branch-corp-cloud.yml

  tasks:

    - name: shut/no shut on all ios interfaces
      ios_config:
        src: "{{ shut_template }}"
      when: ansible_network_os == 'ios'

    - name: pause before enabling interfaces
      pause:
        seconds: 1
      when: ansible_network_os == 'ios'

    - name: shut/no shut on all ios interfaces
      ios_config:
        src: "{{ no_shut_template }}"
      when: ansible_network_os == 'ios'
```

- 2.2. Check the playbook syntax and then perform the play.

```
[student@workstation proj]$ ansible-playbook --syntax-check ios-bounce-if-all.yml
Playbook: ios-bounce-if-all.yml
[student@workstation proj]$ ansible-playbook ios-bounce-if-all.yml
```

This concludes the guided exercise.

Reinitializing Layer 3 Interfaces

Objectives

After completing this section, you should be able to:

- Run a task that loops over a list of network interfaces.
- Obtain a list of network interfaces from a network device.
- Reinitialize the layer 3 configuration of a network device.

Enumerating Interfaces

You saw a previous example in a Guided Exercise of a Jinja2 loop over a defined list of interfaces:

```
{% for intf in interface_data[inventory_hostname] %}  
{% if not intf.name.startswith('Loopback') %}  
interface {{ intf.name }}  
    no shutdown  
{% endif %}  
{% endfor %}
```

You can do a similar operation directly within a task using **loop**:

```
- name: enable all interfaces defined by variable  
  ios_interface:  
    enable: True  
  loop: "{{ interface_data[inventory_hostname] }}"
```

Introspecting Interfaces

Instead of providing a list of interfaces, you can obtain the list of interfaces from the device and apply an action to each.

- Some ***os_facts** modules set an **ansible_net_interfaces** variable with a list of interfaces that are configured to exist on devices. The **ios_facts** module does this.
- If you are working with a platform for which its ***os_facts** module does not support a documented way of automatically obtaining a list of interfaces, the ***os_code** module can be used.

Converting Output to List

The result returned from a command is associated with a variable name by using the **register** keyword. If you want a list of interface names, additional processing is necessary.

```
- name: send command to show interfaces  
  vyos_command:  
    commands:
```

```
- show interfaces
register: result

- set_fact:
  ethernet_interface_rows: "{{ result.stdout_lines[0] | select('search',
'^eth[0-9]+.*') | list }}"

- name: append to list
  set_fact:
    ethernet_interfaces: "{{ ethernet_interfaces }} + [ '{{ item.split(' ')
[0] }}' ]"
  loop: "{{ ethernet_interface_rows }}"
```

Reinitializing Layer 3

The ability to reinitialize layer 3 on a network device can be extremely useful. It is a powerful, and potentially dangerous, ability.

- Removes IPv4 and IPv6 layer 3 addresses.
- Removes routing configuration.
- Use in testing and training environments to establish a clean baseline.
- Might be useful along the side of production environments to process devices returned from the field, or in preparation for provisioning devices.
- Care should be taken to prevent accidental use in production.

► Guided Exercise

Reinitializing Layer 3 on VyOS Devices

In this exercise, you will reinitialize layer 3 on network devices running VyOS.

Outcomes

You should be able to:

- Compose a playbook with a play that reinitializes layer 3 on network devices running VyOS.
- Perform the play in the playbook to enact the desired changes.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Compose a playbook with a play that reinitializes layer 3 on network devices running VyOS. Create a playbook named **vyos-reinitialize-l3.yml** with the following content:

```
---
- name: >
  reinint layer 3 on
  non-mgmt ifaces of
  VyOS devices
hosts: vyos
vars:
  eth_intfs: []
  non_mgmt_eth_intfs: []
  mgmt_intf: eth0

tasks:

- name: send command to show interfaces
  vyos_command:
    commands:
      - show interfaces
    register: r

- name: get interface rows
  set_fact:
    eth_intf_rows: "{{ r.stdout_lines[0]|select('search', '^eth[0-9]+.*')|list }}"
    list }}"

- name: append to list
  set_fact:
    eth_intfs: "{{ eth_intfs }} + [ '{{ item.split(' ')[0] }}' ]"
    loop: "{{ eth_intf_rows }}"

- name: show ethernet interfaces
  debug:
```

```

var: eth_intfs

- name: get non-management interfaces
  set_fact:
    non_mgmt_eth_intfs: "{{ non_mgmt_eth_intfs }} + [ '{{ item }}' ]"
  loop: "{{ eth_intfs | reject('match', mgmt_intf) | list }}"

- name: show non-management ethernet interfaces
  debug:
    var: non_mgmt_eth_intfs

- name: remove IPv4 addresses
  vyos_l3_interface:
    name: "{{ item }}"
    state: absent
  loop: "{{ non_mgmt_eth_intfs }}"

- name: remove addresses from loopback
  vyos_l3_interface:
    name: lo
    state: absent

- name: disable IPv6
  vyos_config:
    lines:
      - set system ipv6 disable
    save: True

- name: remove interface descriptions
  vyos_interface:
    name: "{{ item }}"
    state: absent
  loop: "{{ non_mgmt_eth_intfs }}"

- name: disable OSPF
  vyos_config:
    lines:
      - delete protocols ospf area 0
    save: True

- name: reboot
  vyos_command:
    commands:
      - command: reboot now
    ignore_errors: yes

- name: wait for restart
  wait_for_connection:
    delay: 20
    timeout: 120

```

- 2. Perform the play in the playbook to enact the desired changes. If you are having difficulties with typing a playbook of this length, you can find the completed playbook in the **playbooks** directory on **materials.example.com**.

```
[student@workstation proj]$ ansible-playbook vyos-reinitialize-l3.yml
```

This concludes the guided exercise.

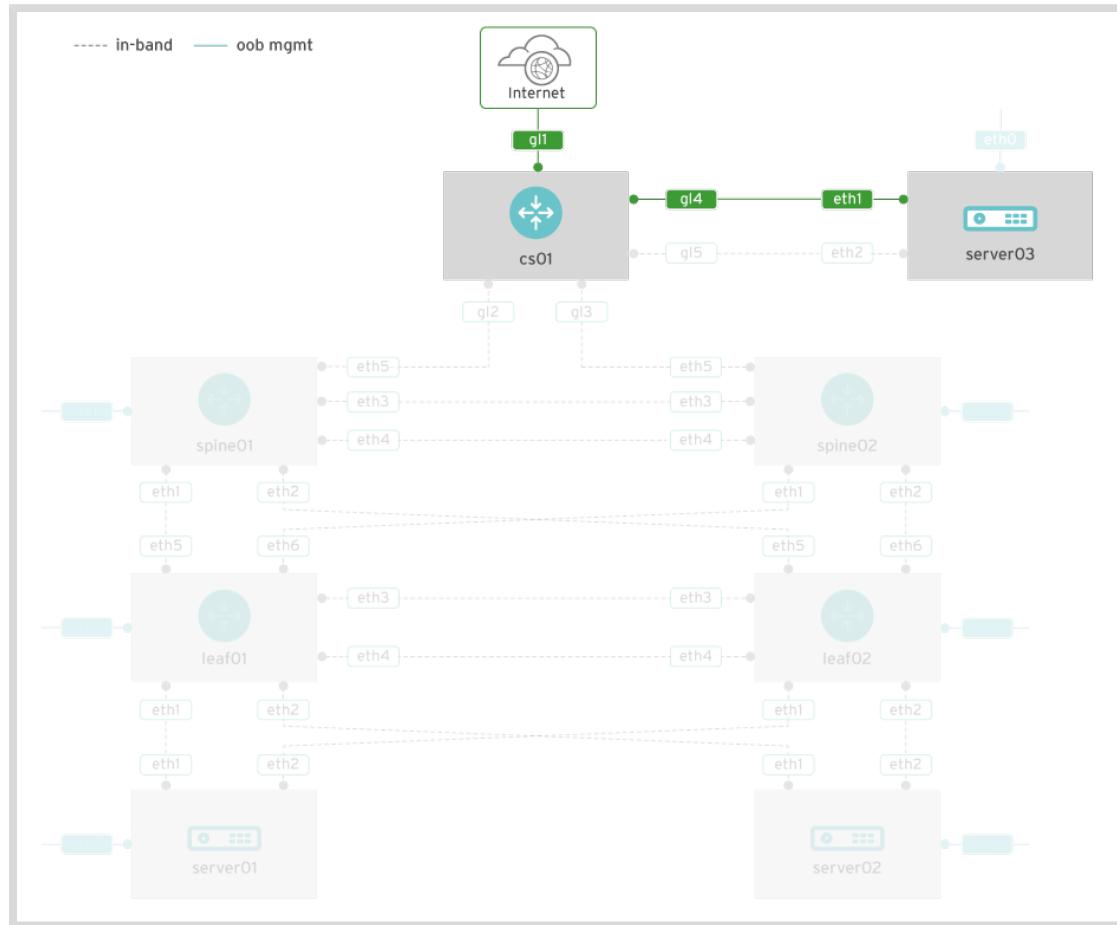
Provisioning the Start-up Network

Objectives

After completing this section, you should be able to use Ansible to provision a basic network containing a simple router.

Provisioning the Start-up Network

A simple network in the cloud with one router and a server.



► Guided Exercise

Provisioning the Start-up Network

In the start-up phase of **example.com**, their first network device has just been provisioned: a CSR1000V router. This is the **cs01** device. The new router is hosted with a cloud services provider (CSP). The act of provisioning it brings it online with a connection to the internet on interface **GigabitEthernet1**.

The first production application server has been provisioned. The server is named **server03**. The **eth1** interface of this server is connected to **vlan1** on a switch belonging to the CSP. The **GigabitEthernet4** interface on **cs01** is connected to the same VLAN and switch.

During this phase, the Production Services Network of **example.com** consists of one network device, **cs01**, which has a server connected to it.

Jasper asks you to compose a playbook that will:

1. Apply the interface description “outside” to the outside interface, **GigabitEthernet1**.
2. Configure layer 3 on the inside interface, **GigabitEthernet4**
3. Apply the interface description “inside” to the inside interface, **GigabitEthernet4**.
4. Bounce the inside interface.
5. Verify that the new server, **server03**, is reachable from **cs01** after bringing up the inside interface.

You will access the device by way of the **GigabitEthernet1** interface. This interface is already configured with respect to layer 3 and up. It is considered the outside interface.

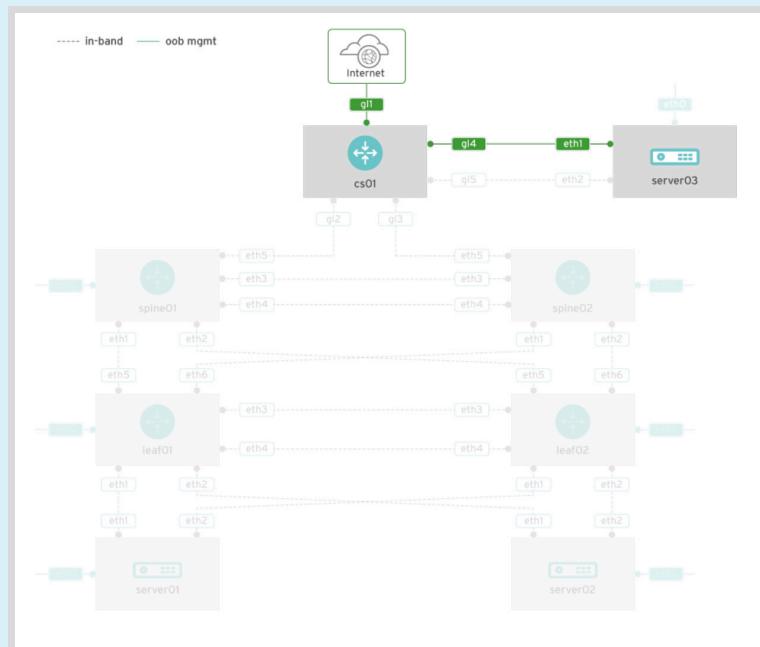


Figure 5.2: The start-up phase Production Services Network of **example.com**.

Start-up Phase Interface Descriptions

Device	Interface	Description
cs01	GigabitEthernet1	outside
cs01	GigabitEthernet4	inside

Start-up Phase Layer 3 Addressing (management network not shown)

Device	Interface	Description
cs01	Loopback1	172.16.0.1/32
cs01	GigabitEthernet4	172.16.10.1/30

In this exercise, you will provision the **example.com** Production Services Network that corresponds to the start-up phase.

Outcomes

You should be able to:

- Create a **vars** file defining the variables that will be used.
- Compose a playbook with a play that satisfies the business requirements as stated by Jasper.
- Perform the play in the playbook to enact the desired changes.
- Verify that the outcome is as intended.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

► 1. Create a **vars** file defining the variables that will be used.

- 1.1. Create a **vars** file named **vars/startup-data.yml** that defines a **layer3_data** variable and an **interface_data** variable as shown here:

```
layer3_data:
  cs01:
    - { name: Loopback1, ipv4: 172.16.0.1/32 }
    - { name: GigabitEthernet4, ipv4: 172.16.10.1/30 }

interface_data:
  cs01:
    GigabitEthernet1:
      description: outside
    GigabitEthernet4:
      description: inside
```

► 2. Compose a playbook with a play that satisfies the business requirements as stated by Jasper.

2.1. Create a playbook named **startup.yml** as shown here:

```

---
- name: define the startup example.com layer3 network
  hosts: cs01
  vars:
    mgmt_intf: GigabitEthernet1
    server_ipv4: 172.16.10.2/30
  vars_files:
    - vars/startup-data.yml

  tasks:

    - name: remove old layer3 interface data
      ios_l3_interface:
        aggregate: "{{ layer3_data[inventory_hostname] }}"
        state: absent

    - name: configure layer3 interfaces
      ios_l3_interface:
        aggregate: "{{ layer3_data[inventory_hostname] }}"

    - name: configure description of management interface
      # do management interface separately, do not shut down
      ios_interface:
        name: "{{ mgmt_intf }}"
        description: >
          {{ interface_data[inventory_hostname][mgmt_intf].description }}

    - name: configure interface description
      ios_interface:
        name: "{{ item.key }}"
        description: "{{ item.value.description }}"
        enabled: no
        when: not item.key == mgmt_intf
        with_dict: "{{ interface_data[inventory_hostname] }}"

    - name: bring interfaces up
      ios_interface:
        name: "{{ item.key }}"
        enabled: yes
        when: not item.key == mgmt_intf
        with_dict: "{{ interface_data[inventory_hostname] }}"

    - name: pause
      pause:
        seconds: 1

    - name: test connectivity to server
      ios_ping:
        dest: "{{ server_ipv4 | ipaddr('address') }}"
        register: result

```

```
- name: show result
  debug:
    var: result
```

- 3. Perform the play in the playbook to enact the desired changes.

```
[student@workstation proj]$ ansible-playbook startup.yml
```

- 4. Execute ad hoc commands and verify that layer 3 addresses are mapped to interfaces as described in the tables found at the start of this exercise. Only one networking device appears in the Start Up scenario: **cs01**. Here the syntax of the command is shown for use with **cs01**:

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh ip int br'" cs01
```

This concludes the guided exercise.

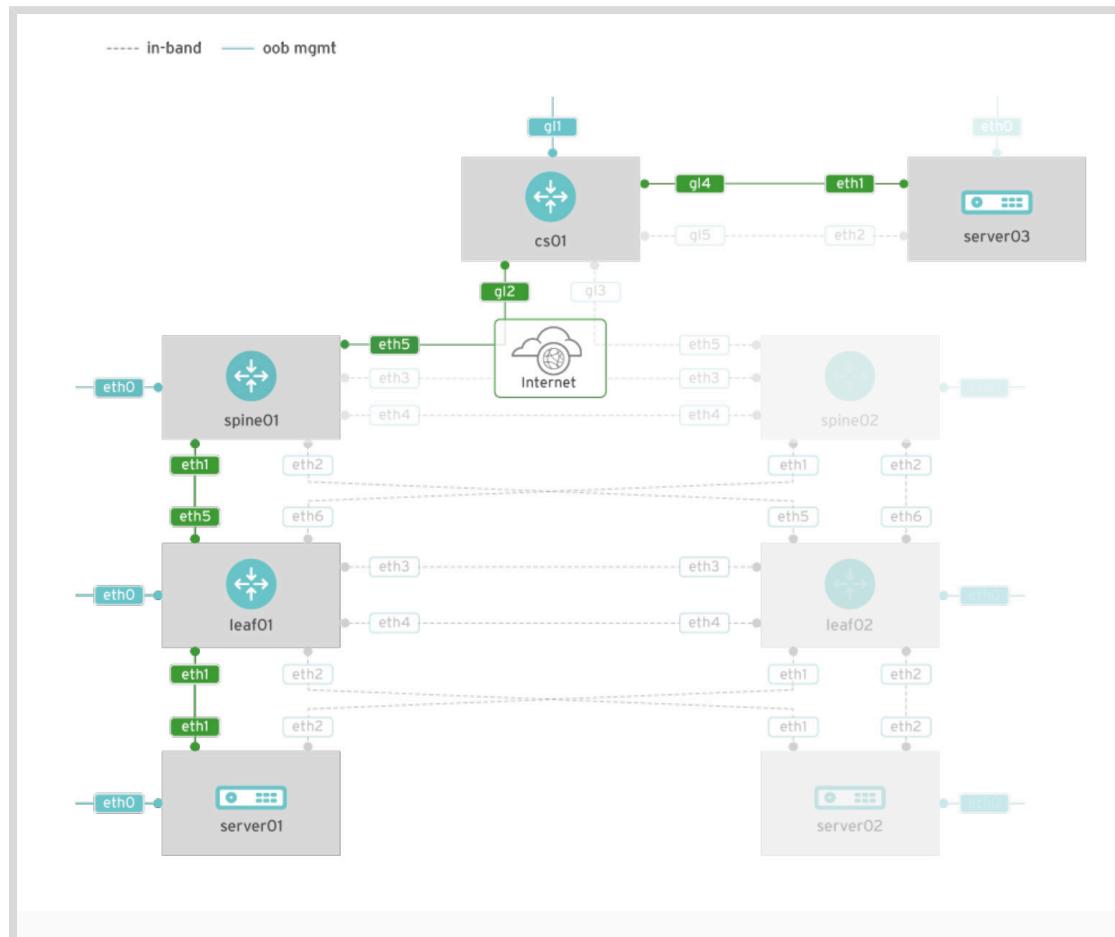
Provisioning Spine and Leaf Devices

Objectives

After completing this section, you should be able to use Ansible to provision network devices configured in a spine and leaf topology.

Provisioning the Expansion Network

Cloud plus an office location. Spine and leaf plus server at the office.



► Guided Exercise

Provisioning Spine and Leaf Devices

Example.com is moving into a new office building. This is their first brick-and-mortar location. They have a new server. A VyOS router named **spine01** is online and connected to the management network, as well as an intermediate device named **leaf01**, which is also running VyOS. They decided to use static routes for now: static routes are simple and reliable, and in this stage of their development, **example.com** is young enough and small enough that Jasper has not decided to make scalability a priority.

During this phase, the Production Services Network of **example.com** consists of **cs01** and **server03** in the cloud, plus now two network devices at the new office: **spine01** and **leaf01**. There is also a server at the office: **server01**. Counting the cloud and the office, there are now three network devices that will be active at layer 3 and two servers.

Jasper is asking for the following:

Make these changes on **spine01**:

1. Label interface **eth0** with the description **management**.
2. Label interface **eth5** with the description **outside**.
3. Label interface **eth1** with the description **internal**.
4. Configure layer 3 on interfaces **eth1** and **eth5** according to the table.
5. Do not change anything with respect to layer 3 on the **management** interface.
6. Make sure OSPF is not enabled.
7. The default gateway should point to the next hop address associated with the **outside** interface.
8. Add a static route for the server subnet that points to the next hop associated with the **internal** interface.

Make these changes on **leaf01**:

1. Label interface **eth0** with the description **management**.
2. Label interface **eth5** with the description **uplink**.
3. Label interface **eth1** with the description **server01**.
4. Configure layer 3 on the **uplink** and **server01** interfaces.
5. Do not change anything with respect to layer 3 on the **management** interface.
6. Make sure OSPF is not enabled.
7. The default gateway should point to the next hop address associated with the **uplink** interface.

Make these changes on **cs01**:

1. Label interface **GigabitEthernet1** with the description **management**.
2. Label interface **GigabitEthernet2** with the description **outside**.
3. Configure layer 3 on interface **GigabitEthernet2** according to the table.
4. The default route should point to the next hop address associated with the **outside** interface.

You will access **spine01** and **leaf01** by way of their **eth0** interfaces, and access **cs01** by way of its **GigabitEthernet1** interface. These interfaces are already configured with respect to layer 3 and up.

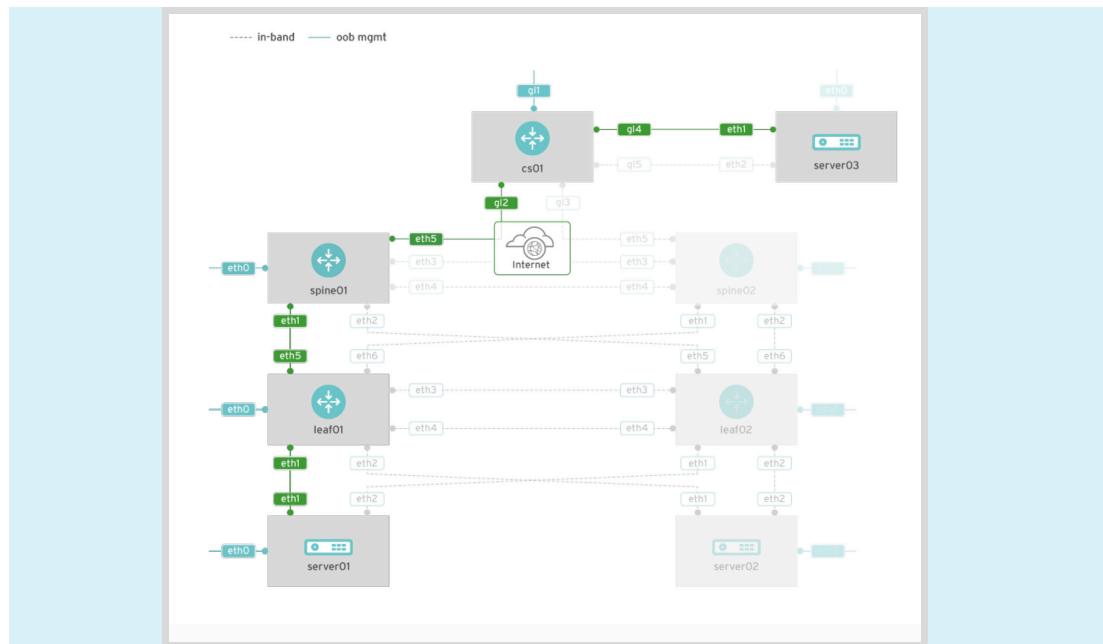


Figure 5.4: The expansion phase Production Services Network of example.com.

Expansion Phase Interface Descriptions

Devices	Interfaces	Descriptions
cs01	GigabitEthernet1	management
cs01	GigabitEthernet2	outside
cs01	GigabitEthernet4	inside
spine01	eth0	management
spine01	eth1	inside
spine01	eth5	outside
leaf01	eth0	management
leaf01	eth1	server01
leaf01	eth5	uplink

Expansion Phase Layer 3 Addressing (management interface not shown)

Devices	Interfaces	Descriptions
cs01	Loopback1	172.16.0.1/32
cs01	GigabitEthernet2	172.16.2.2/30
cs01	GigabitEthernet4	172.16.10.1/30
spine01	lo	10.0.0.1/32

Devices	Interfaces	Descriptions
spine01	eth1	10.10.5.1/30
spine01	eth5	172.16.2.1/30
leaf01	lo	10.0.0.11/32
leaf01	eth1	10.10.10.1/30
leaf01	eth5	10.10.5.2/30

In this exercise, you will provision the **example.com** Production Services network that corresponds to the expansion phase.

Outcomes

You should be able to:

- Create a **vars** file defining the variables that will be used.
- Compose a playbook with a play that satisfies the business requirements as stated by Jasper.
- Perform the play in the playbook to enact the desired changes.
- Verify that the outcome is as intended.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

► 1. Create a **vars** file defining the variables that will be used.

- 1.1. Download the **expansion-data.yml** file into the **vars/** directory. This file maps layer 3 address data to interfaces and also provides interface descriptions.

```
[student@workstation proj]$ cd vars
[student@workstation vars]$ wget \
> http://materials.example.com/full/vars/expansion-data.yml
[student@workstation vars]$ cd ..
[student@workstation proj]$
```

► 2. Compose a playbook with a play that satisfies the business requirements as stated by Jasper.

- 2.1. Create a playbook named **expansion.yml** as shown below. This could be developed as three separate playbooks and then combined, possibly by using **include** or **import**. Alternatively, roles could be used.

This provides an example of a simple, straightforward method of explicitly encoding actions as tasks. It works, and it is relatively easy to look at this playbook and understand what is being done. There are, however, more efficient ways to accomplish what this playbook does. The playbook used to provision the **example.com** network for the Consolidation phase accomplishes a similar set of objectives with one play.

2.2. Write a play containing tasks that configure **cs01**.

```

- name: configure cs01 for expansion
  hosts: cs01
  vars_files:
    - vars/expansion-data.yml

  tasks:

    - name: label the management interface
      ios_interface:
        name: GigabitEthernet1
        description: management

    - name: label the outside interface
      ios_interface:
        name: GigabitEthernet2
        description: outside

    - name: configure layer3 interfaces
      ios_l3_interface:
        aggregate: "{{ layer3_data[inventory_hostname] }}"

    - name: set default gateway
      ios_static_route:
        prefix: 0.0.0.0
        mask: 0.0.0.0
        next_hop: 172.16.2.1

    - name: bring the outside interface up
      ios_interface:
        name: GigabitEthernet2
        enabled: yes
  
```

2.3. Write a play containing tasks that configure **spine01**.

```

- name: configure spine01 for expansion
  hosts: spine01
  vars_files:
    - vars/expansion-data.yml

  tasks:

    - name: make sure ospf is not enabled
      vyos_config:
        lines:
          - delete protocols ospf

    - name: label the management interface (eth0)
      vyos_interface:
        name: eth0
        description: management

    - name: label the outside interface (eth5)
  
```

```

vyos_interface:
  name: eth5
  description: outside

  - name: label the inside interface (eth1)
    vyos_interface:
      name: eth1
      description: inside

  - name: configure layer 3
    vyos_l3_interface:
      aggregate: "{{ layer3_data[inventory_hostname] }}"

  - name: set default gateway
    vyos_static_route:
      prefix: 0.0.0.0
      mask: 0
      next_hop: 172.16.2.2

  - name: add static route for server subnet
    vyos_static_route:
      prefix: 10.10.10.0
      mask: 30
      next_hop: 10.10.5.2

```

2.4. Write a play containing tasks that configure **leaf01**.

```

- name: >
  a play that configures leaf01
  this is the expansion model
  hosts: leaf01
  vars_files:
    - vars/expansion-data.yml

  tasks:

  - name: label the management interface (eth0)
    vyos_interface:
      name: eth0
      description: management

  - name: label the uplink interface (eth5)
    vyos_interface:
      name: eth5
      description: uplink

  - name: label the server interface (eth1)
    vyos_interface:
      name: eth1
      description: server01

  - name: configure layer3 interfaces
    vyos_l3_interface:
      aggregate: "{{ layer3_data[inventory_hostname] }}"

```

```
- name: set default gateway
  vyos_static_route:
    prefix: 0.0.0.0
    mask: 0
    next_hop: 10.10.5.1
```

- 3. Perform the plays in the playbook to enact the desired changes.

```
[student@workstation proj]$ ansible-playbook expansion.yml
```

- 4. Execute ad hoc commands and verify that layer 3 addresses and interface descriptions are mapped to interfaces as described in the tables found at the start of this exercise. The expansion scenario networking devices are **spine01**, **leaf01**, and **cs01**. The following ad hoc command verifies the addresses and interface descriptions on **spine01** and **leaf01**:

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh int'" spine01,leaf01
```

The following ad hoc command checks layer 3 addresses on **cs01**.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh ip int br'" cs01
```

This concludes the guided exercise.

Setting Parameters with Ansible Tower Surveys

Objectives

After completing this section, you should be able to use a Survey in Red Hat Ansible Tower to interactively set extra variables that are used by your playbook.

Extending Parameterizing Automation

Actions with required data that is not available in advance are challenging to automate. This is true even when automating simple network operations.

Many network related operations exist that involve data that might not be available in advance. For example:

- Adding a static host entry
- Adding a group object, network object, or host object
- Adding an address to an ACL or FW rule
- Adding a destination address to lists of blocked malware sites
- Tracing the path traversed by packets between two points
- Using ICMP to test reachability between two points
- Changing the destination to which logged events are forwarded

Providing Extra Data for Playbooks

We would like a way to provide data at the start of an automated process.

You have already seen how Red Hat Ansible Engine lets you pass additional data to the **ansible** and **ansible-playbook** commands by way of the **--extra-vars** option (**-e**).

Red Hat Ansible Tower has two more ways to provide additional data at the start of an automated process: Prompt for Extra Variables and Surveys.

- Prompt for Extra Variables does exactly what it says: it prompts for extra variables.
- Surveys set extra variables in a user-friendly question and answer way. Surveys also allow for validation of user input.
- Surveys are defined in conjunction with Job Templates.
- Two types of Job Templates support Surveys: Run and Check.

► Guided Exercise

Setting Parameters with Ansible Tower Surveys

It is not at all uncommon for an organization to have a policy of not adding human-friendly names of network devices to DNS, for security reasons. Many humans, of course, prefer to work with names rather than IP addresses. Static local host names map a human friendly host name to a layer 3 address, without using DNS. Managing static local host names on network devices allows network administrators to work with names, without the potential security considerations associated with listing devices in DNS.

Maintaining static local host names is relatively simple with automation, but can be disastrous without it.

This exercise demonstrates how the Survey feature in Red Hat Ansible Tower makes it easy to add a static local host name to all managed network devices.

In this exercise, you will push a host name mapping to all managed network devices.

Outcomes

You should be able to:

- Create a Vault credential in Red Hat Ansible Tower.
- Create a Project in Red Hat Ansible Tower.
- Create a Job Template in Red Hat Ansible Tower.
- Create a Survey with the Job Template.
- Create Jobs by launching the Job Template.
- Review the status of Jobs.

Before You Begin

Open Firefox on the **workstation** VM. Log in to Ansible Tower at `https://tower.lab.example.com` as **admin** using **student** as the password. It is assumed that Lab 4 has been completed successfully so that the inventory named **lab.example.com** exists.

- ▶ 1. Create a Vault credential named **example.com-credential** in Red Hat Ansible Tower.
 - 1.1. Click the **Settings** icon (the gear) and select **CREDENTIALS**.
 - 1.2. Click **ADD** to display the **NEW CREDENTIAL** page. In the **NAME** field, type **example.com-credential**. Specify **Vault** as the **CREDENTIAL TYPE**. In the **ORGANIZATION** field, click the magnifying glass and select **Default**. In the **VAULT PASSWORD** field, type **redhat**. Click **SAVE** to create the Vault credential.
- ▶ 2. Create a project named **add-static-hostname** in Red Hat Ansible Tower.
 - 2.1. Select **PROJECTS** in the quick menu at the upper left.
 - 2.2. Click **ADD** to create a new project. In the **NAME** field, type **add-static-hostname**. Specify **Default** as the **ORGANIZATION**. For **SCM TYPE**, select

Git. In the **SCM URL** field of the **SOURCE DETAILS** panel, type `http://git.lab.example.com:3000/student/network-static-hostname.git`. In the **SCM BRANCH/TAG/COMMIT** field, type **master**. Within the lab environment, this is considered a public repository, so leave the **SCM CREDENTIAL** field empty. Click **SAVE** to create the project.

- ▶ **3.** Create a job template named **add-static-hostname** in Red Hat Ansible Tower.
 - 3.1. Select **TEMPLATES** in the quick menu at the top of the window.
 - 3.2. Click **ADD** and select **Job Template** to display the **NEW JOB TEMPLATE** page. In the **NAME** field, type **add-static-hostname**. Set the **JOB TYPE** to **Run**. In the **INVENTORY** field, type **lab.example.com**. That is the inventory you created in Lab 4. In the **PROJECT** field, click the magnifying glass and select the **add-static-hostname** project. In the **PLAYBOOK** field, select **multi-vendor-static-hostname.yml**. In the **CREDENTIAL** field, click the magnifying glass, select **Vault** for the **CREDENTIAL TYPE**, select **example.com-credential**, and click **SELECT** to complete your selection. Click **SAVE** to save your work.
- ▶ **4.** Create a survey with the job template.
 - 4.1. On the **NEW JOB TEMPLATE** page for the **add-static-hostname** job template, click **ADD SURVEY** to display the **ADD SURVEY PROMPT** page.
 - 4.2. In the **PROMPT** field, type “Hostname?”. In the **DESCRIPTION** field, type “A human-friendly name”. In the **ANSWER VARIABLE NAME** field, type **static_local_hostname**. In the **ANSWER TYPE** field, select **Text** and click **ADD** to add the survey prompt.
 - 4.3. On the **ADD SURVEY PROMPT** page, type “Address?” in the **PROMPT** field. In the **DESCRIPTION** field, type “An IPv4 host address”. In the **ANSWER VARIABLE NAME** field, type **static_local_ipv4addr**. In the **ANSWER TYPE** field, select **Text**. For **MINIMUM LENGTH**, type the minimum number of characters in an IPv4 address, which is 7. For **MAXIMUM LENGTH**, type the maximum number of characters in an IPv4 address, which is 15. Click **ADD** to add this survey prompt.
 - 4.4. Check your work in the **PREVIEW** pane at the right side of the window. Click the edit icon (the pencil) to make corrections. When you are satisfied with your work, click **SAVE**.
- ▶ **5.** Create jobs by launching the job template.
 - 5.1. At the **TEMPLATES / add-static-hostname** page, scroll down to the **TEMPLATES** pane at the bottom of the page. In the **add-static-hostname** job template row, click the launch icon (the rocket) to start a job using that template.
 - 5.2. On the **LAUNCH JOB | add-static-hostname SURVEY** page, type “tower” in the **HOSTNAME** field. Type **172.25.250.9** in the **ADDRESS** field, and then click **LAUNCH**.
- ▶ **6.** Review the status of jobs.
 - 6.1. Click **JOBS** in the quick menu at the top of the window. Locate the most recent **add-statichostname** job. Click the magnifying glass in its row to view the job.

This concludes the guided exercise.

▶ Lab

Automating Simple Network Operations

The process of configuring network devices is illustrated with simple changes in this lab.

In this lab, you will automate simple operations on network devices.

Outcomes

You should be able to:

- Consult local (**ansible-doc**) or web-based documentation to determine correct configuration statements to use with a given **os_config** module.
- Configure devices using statements embedded in a playbook.
- Configure devices by sourcing statements from a file.
- Generate configuration files using Jinja2 and apply the changes to devices.

Before You Begin

Open a terminal window on the **workstation** VM.

1. Create a playbook named **rename-spine01.yml**. Use **os_config** statements in the playbook to change the local host name of **spine01** on the network device to **spineA1**. Do not forget to use the **os_config** module that matches the device. Include a task that backs up the original configuration before the change is applied.
2. Perform the play in the **rename-spine01.yml** playbook.
3. Execute an ad hoc command to confirm that the host name of **spine01** is now **spineA1**. Also verify that the original device configuration was backed up.
4. Execute an ad hoc command to change the host name back to **spine01**.
5. Create a directory named **config/**.
6. Create a static configuration file named **config/spine02.cfg**. Put in this file this VyOS configuration statement that sets the host name to **spineB2**: **set system host-name spineB2**.
7. Create a playbook named **rename-spine02.yml**. Include in this playbook a task that sources configuration statements from the **config/spine02.cfg** file.
8. Perform the play in the **rename-spine02.yml** playbook. If you receive an error regarding the file path, make sure you have placed **spine02.cfg** into the **coding** directory.
9. Execute an ad hoc command to confirm that the host name of **spine02** is now **spineB2**.
10. Execute an ad hoc command to change the host name back to **spine02**.
11. Create the file **host_vars/cs01/vars.yml** if it does not already exist. Add a YAML mapping to the **host_vars/cs01/vars.yml** file that maps the variable name **new_hostname** to the value **csA1**.
12. Create a Jinja2 template named **ios-cs-hostname.j2** that consists of the IOS command to set the host name. When setting the host name with Ansible, it makes sense to use the

Ansible magic variable **inventory_hostname**, but for the purpose of this exercise, use the variable **new_hostname** instead. The IOS command to set the host name is **hostname hostname**.

13. Create a playbook named **rename-cs01.yml**. The play in this playbook should contain a task that uses the **ios_config** module, in which the lines used by **ios_config** are sourced from the **ios-cs-hostname.j2** template.
14. Perform the play in the **rename-cs01.yml** playbook.
15. Execute an ad hoc command and confirm that the host name of **cs01** is now **csA1**.
16. Execute an ad hoc command to change the host name back to **cs01**.

This concludes the lab.

► Solution

Automating Simple Network Operations

The process of configuring network devices is illustrated with simple changes in this lab.

In this lab, you will automate simple operations on network devices.

Outcomes

You should be able to:

- Consult local (**ansible-doc**) or web-based documentation to determine correct configuration statements to use with a given **os_config** module.
- Configure devices using statements embedded in a playbook.
- Configure devices by sourcing statements from a file.
- Generate configuration files using Jinja2 and apply the changes to devices.

Before You Begin

Open a terminal window on the **workstation** VM.

1. Create a playbook named **rename-spine01.yml**. Use **os_config** statements in the playbook to change the local host name of **spine01** on the network device to **spineA1**. Do not forget to use the **os_config** module that matches the device. Include a task that backs up the original configuration before the change is applied.

```
[student@workstation proj]$ cat rename-spine01.yml
---
- name: Lab 5, spine01 => spineA1
  hosts: spine01
  vars:
    new_hostname: spineA1

  tasks:
    - name: change the hostname
      vyos_config:
        backup: yes
        lines: set system host-name {{ new_hostname }}
```

2. Perform the play in the **rename-spine01.yml** playbook.

```
[student@workstation proj]$ ansible-playbook rename-spine01.yml
```

3. Execute an ad hoc command to confirm that the host name of **spine01** is now **spineA1**. Also verify that the original device configuration was backed up.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" spine01
spine01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "spineA1"
    ],
    "stdout_lines": [
        [
            "spineA1"
        ]
    ]
}
```

4. Execute an ad hoc command to change the host name back to **spine01**.

```
[student@workstation proj]$ ansible -m vyos_config \
> -a "lines='set system host-name spine01'" spine01
```

5. Create a directory named **config/**.

```
[student@workstation proj]$ mkdir config
```

6. Create a static configuration file named **config/spine02.cfg**. Put in this file this VyOS configuration statement that sets the host name to **spineB2**: **set system host-name spineB2**.

```
[student@workstation proj]$ cat config/spine02.cfg
set system host-name spineB2
```

7. Create a playbook named **rename-spine02.yml**. Include in this playbook a task that sources configuration statements from the **config/spine02.cfg** file.

```
[student@workstation proj]$ cat rename-spine02.yml
---
- name: Lab 5, spine02 => spineB2
  hosts: spine02
  vars:
    config_filepath: ./config
  tasks:
    - name: change the hostname
      vyos_config:
        src: "{{ config_filepath }}/{{ inventory_hostname }}.cfg"
        backup: yes
```

8. Perform the play in the **rename-spine02.yml** playbook. If you receive an error regarding the file path, make sure you have placed **spine02.cfg** into the **config** directory.

```
[student@workstation proj]$ ansible-playbook rename-spine02.yml
```

9. Execute an ad hoc command to confirm that the host name of **spine02** is now **spineB2**.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh host name'" spine02
```

10. Execute an ad hoc command to change the host name back to **spine02**.

```
[student@workstation proj]$ ansible -m vyos_config \
> -a "lines='set system host-name spine02'" spine02
```

11. Create the file **host_vars/cs01/vars.yml** if it does not already exist. Add a YAML mapping to the **host_vars/cs01/vars.yml** file that maps the variable name **new_hostname** to the value **csA1**.

```
[student@workstation proj]$ cat host_vars/cs01/vars.yml
new_hostname: csA1
```

12. Create a Jinja2 template named **ios-cs-hostname.j2** that consists of the IOS command to set the host name. When setting the host name with Ansible, it makes sense to use the Ansible magic variable **inventory_hostname**, but for the purpose of this exercise, use the variable **new_hostname** instead. The IOS command to set the host name is **hostname hostname**.

```
[student@workstation proj]$ cat j2/ios-cs-hostname.j2
hostname {{ new_hostname }}
```

13. Create a playbook named **rename-cs01.yml**. The play in this playbook should contain a task that uses the **ios_config** module, in which the lines used by **ios_config** are sourced from the **ios-cs-hostname.j2** template.

```
[student@workstation proj]$ cat rename-cs01.yml
---
- name: a play that uses a Jinja2 template and host vars to rename cs01
  hosts: cs01

  tasks:

  - name: configure host
    ios_config:
      src: j2/ios-cs-hostname.j2
```

14. Perform the play in the **rename-cs01.yml** playbook.

```
[student@workstation proj]$ ansible-playbook rename-cs01.yml
```

15. Execute an ad hoc command and confirm that the host name of **cs01** is now **csA1**.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh run | include hostname'" cs01
cs01 | SUCCESS => {
    "changed": false,
    "stdout": [
```

```
        "hostname csA1"
    ],
    "stdout_lines": [
        [
            "hostname csA1"
        ]
    ]
}
```

16. Execute an ad hoc command to change the host name back to **cs01**.

```
[student@workstation proj]$ ansible -m ios_config -a "lines='hostname cs01'" cs01
```

This concludes the lab.

Chapter 6

Automating Complex Network Operations

Goal	Automate complex operations on network devices
Objectives	<ul style="list-style-type: none">• Apply changes to a network that involve many tasks using playbooks.• Resolve network issues using playbooks.
Sections	<ul style="list-style-type: none">• Aggregating Logged Events to Syslog (and Guided Exercise)• Managing Access Control Lists on IOS (and Guided Exercise)• Enabling SNMP (and Guided Exercise)• Overcoming Real World Challenges (and Guided Exercise)• Implementing Dynamic Routing with OSPF (and Guided Exercises)• Implementing OSPF with Multiple Autonomous Systems (and Guided Exercise)• Implementing Dynamic Routing with EBGP (and Guided Exercise)• Upgrading the Network (and Guided Exercise)
Lab	Automating Complex Operations

Aggregating Logged Events to Syslog

Objectives

After completing this section, you should be able to forward logging events to a syslog server for log aggregation.

Aggregating Logged Events

Ansible can automate the process of configuring devices to send logged event information to a syslog server.

The following task configures log aggregation for a VyOS device:

```
- name: set the logging target
vyos_config:
  lines:
    - set system syslog host {{ syslog_ipv4addr }} facility local7 level debug
```

The following task configures log aggregation for an IOS device:

```
- name: set the logging target
ios_config:
  lines:
    - service timestamps log datetime
    - service timestamps debug datetime
    - logging {{ syslog_ipv4addr }}
    - logging trap 7
```

► Guided Exercise

Aggregating Logged Events to Syslog

The robustness and reliability of managed network resources depends on good situational awareness: having the right information, at the right time, in the right place. One way to gather useful information is to forward event information from your network devices to a central **syslog** server. This makes it possible to construct rule-based alerting on suspicious event patterns.

In this exercise, you will send logged events to a **syslog** server.

Outcomes

You should be able to:

- Add group variables that support template-driven configuration for sending logged event information to the **workstation** machine acting as a **syslog** server.
- Update NOS-specific device configuration templates to provide parameterized configuration statements for sending logged event information to a **syslog** server.
- Perform a multivendor play that configures network devices to send logged event information to a **syslog** server.
- Monitor traffic on the **syslog** port on the **workstation** VM to confirm that log messages are arriving.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj` directory.

- 1. Add variables that support template-driven configuration for sending logged event information to a **syslog** server.
- 1.1. Add a variable named **syslog_ipv4** to the **network** group's variables file. Its value should be the IP address of the **workstation** machine, which has already been configured to accept **syslog** traffic. Modify **group_vars/network/vars.yml** to include the following content:

```
ansible_connection: network_cli
domain_name: lab.example.com
syslog_ipv4: 172.25.250.254
nameservers:
- 8.8.8.8
- 8.8.4.4
```

- 1.2. Add a variable named **vyos_loglevel** to the **vyos** group's variables file. Modify **group_vars/vyos/vars.yml** to include the following content:

```
ansible_network_os: vyos
ansible_user: vyos
vyos_loglevel: info
```

- 1.3. Add a variable named **ios_loglevel** to the **ios** group's variables file. Modify **group_vars/ios/vars.yml** to include the following content:

```
ansible_network_os: ios
ansible_user: admin
# level 6 = informational, level 7 = debug
ios_loglevel: 7
```

- 2. Update NOS-specific device configuration templates to provide parameterized configuration statements for sending logged event information to a **syslog** server.
- 2.1. Add a line to the device configuration Jinja2 template for VyOS devices and **j2/vyos-config.j2**. It should map appropriate variables to the VyOS statements that enable logging to the **syslog** server.

```
[student@workstation proj]$ cat j2/vyos-config.j2
set system host-name {{ inventory_hostname }}
set system domain-name {{ domain_name }}
{% for nameserver in nameservers %}
set system name-server {{ nameserver }}
{% endfor %}
set system syslog host {{ syslog_ipv4 }} facility local7 level {{ vyos_loglevel }}
```

- 2.2. Add lines to the device configuration Jinja2 template for IOS devices, **j2/ios-config.j2**. It should map appropriate variables to the IOS statements that enable logging to the **syslog** server.

```
[student@workstation proj]$ cat j2/ios-config.j2
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
{% for nameserver in nameservers %}
ip name-server {{ nameserver }}
{% endfor %}
service timestamps log datetime
service timestamps debug datetime
logging {{ syslog_ipv4 }}
logging trap {{ ios_loglevel }}
```

- 3. Perform a multivendor play that configures network devices to send logged event information to a **syslog** server.

Perform the play found in the **j2cfg.yml** file. You already created playbook **j2cfg.yml**, which sources configuration statements from the VyOS and IOS Jinja2 templates. Limit the change to **cs01**.

```
[student@workstation proj]$ ansible-playbook -l cs01 j2cfg.yml
```

- 4. Monitor traffic on the **syslog** port on the **workstation** VM to confirm that log messages are arriving.

4.1. Run **tcpdump**, listening on **eth0** port 514.

```
[student@workstation ~]$ sudo tcpdump -Xni eth0 port 514
[sudo] password for student: student
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

- 4.2. Open another terminal window on the **workstation** machine. Use SSH to connect to **cs01** as **admin** using **student** as the password. Generate a log message on **cs01**.

```
[student@workstation ~]$ ssh admin@cs01
Password: student

cs01#send log this is a test
```

- 4.3. In your **tcpdump** terminal window, you should see a hex dump of the log message you generated on **cs01**.

```
[student@workstation ~]$ sudo tcpdump -Xni eth0 port 514
[sudo] password for student: student
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
10:56:23.721408 IP 172.25.250.195.53969 > 172.25.250.254.syslog: SYSLOG
local7.debug, length: 98
 0x0000: 4500 0076 0002 0000 ff11 6d7f ac19 fac3 E..v.....m.....
 0x0010: ac19 fafe d2d1 0202 0062 96ce 3c31 3931 .....b..<191
 0x0020: 3e35 383a 202a 4a75 6c20 2033 2031 343a >58:.*Jul..3.14:
 0x0030: 3531 3a30 323a 2025 5359 532d 372d 5553 51:02:.%SYS-7-US
 0x0040: 4552 4c4f 475f 4445 4255 473a 204d 6573 ERLOG_DEBUG:.Mes
 0x0050: 7361 6765 2066 726f 6d20 7474 7931 2875 sage.from.tty1(u
 0x0060: 7365 7220 6964 3a20 6164 6d69 6e29 3a20 ser.id:.admin):.
 0x0070: 7468 6973 2069 7320 6120 7465 7374      this.is.a.test
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
```

Use **Control+C** to break out of the **tcpdump** session.

- 4.4. If the message is not displayed in the terminal where **tcpdump** is running, make sure the **syslog** port is open on the firewall.

```
[student@workstation ~]$ sudo firewall-cmd --zone=trusted \
> --permanent --add-port=514/udp
[student@workstation ~]$ sudo firewall-cmd --reload
[student@workstation ~]$ sudo firewall-cmd --zone=trusted --list-ports
```

This concludes the guided exercise.

Managing Access Control Lists on IOS

Objectives

After completing this section, you should be able to create an ACL on a Cisco IOS device.

Managing Access Control Lists (ACLs)

On IOS devices, ACLs are used to control access to services.

This playbook creates a MGMT-ACCESS ACL, which can then be associated with SNMP or SSH, for instance.

```
---
- name: A play that creates a management access ACL
  hosts: ios
  gather_facts: no

  tasks:
    - name: create a standard ACL
      ios_config:
        lines:
          # each of the following two items consist of a single line
          # with no line breaks
          - 10 permit {{ workstation_ipv4 | ipaddr('address') }}
            {{ workstation_ipv4 | ipaddr('wildcard') }} log
          - 20 permit {{ tower_ipv4 | ipaddr('address') }} {{ tower_ipv4 |
            ipaddr('wildcard') }} log
        parents: ["access-list standard 1"]
        before: ["no access-list standard 1"]
        match: exact
```

► Guided Exercise

Managing Access Control Lists on IOS

Access control lists are used in many different situations. You can use them to deny access from sources of known malicious traffic, and to allow access to various services in use, such as SNMP.

In this exercise, you will automate the process of creating and managing an Access Control List (ACL) on IOS devices.

Outcomes

You should be able to:

- Add group variables that support template-driven configuration to support a management access ACL.
- Update NOS-specific device configuration templates to provide parameterized configuration statements that define a management access ACL.
- Perform a play that configures network devices from the updated Jinja2 templates.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj** directory.

- 1. Add group variables that support template-driven configuration to support a management access ACL.

- 1.1. Create a **group_vars/all/** directory if it does not already exist.

```
[student@workstation proj]$ mkdir -p group_vars/all
```

- 1.2. Add variables **workstation_ipv4** and **tower_ipv4** to the **group_vars/all/vars.yml** file. Create this file if it does not already exist. Edit it to make sure it contains the following contents:

```
workstation_ipv4: 172.25.250.254/24
tower_ipv4: 172.25.250.9/24
```

- 2. Update NOS-specific device configuration templates to provide parameterized configuration statements that define a management access ACL.

- 2.1. Add lines to the device configuration Jinja2 template for IOS device configuration. Under IOS, SNMP requires a standard access list, so that is used here. Modify **j2/ios-config.j2** to include the following content:

```
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
{% for nameserver in nameservers %}
```

```

ip name-server {{ nameserver }}
{% endfor %}
service timestamps log datetime
service timestamps debug datetime
logging {{ syslog_ipv4 }}
logging trap {{ ios_loglevel }}
access-list 1 permit {{ workstation_ipv4 | ipaddr('address') }} log
access-list 1 permit {{ tower_ipv4 | ipaddr('address') }} log

```

- 3. Perform a play that configures network devices from the updated Jinja2 templates.

- 3.1. Perform the play found in the **j2cfg.yml** file. Limit it to **cs01**. You already created playbook **j2cfg.yml**, which sources configuration statements from the VyOS- and IOS- Jinja2 templates.

```
[student@workstation proj]$ ansible-playbook -l cs01 j2cfg.yml
```

- 3.2. Execute an ad hoc command that verifies that the ACL has been created.

```

[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh access-list 1'" cs01
cs01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "Standard IP access list 1\n      20 permit 172.25.250.9 log\n
      10 permit 172.25.250.254 log"
    ],
    "stdout_lines": [
        [
            "Standard IP access list 1",
            "      20 permit 172.25.250.9 log",
            "      10 permit 172.25.250.254 log"
        ]
    ]
}

```

This concludes the guided exercise.

Enabling SNMP

Objectives

After completing this section, you be able to enable SNMP management and configure a read-only community with Ansible.

Enabling SNMP

Ansible makes it easy to manage configuration of SNMP settings, including the community string and access restrictions.

The following task configures the community string for read-only access on an IOS device:

```
- name: set the RO community string
  ios_config:
    lines:
      - snmp-server community {{ ro_community }} RO
```

A similar configuration statement for VyOS is shown below:

```
- name: set the RO community string
  vyos_config:
    lines:
      - set service snmp community {{ ro_community }} authorization ro
```

► Guided Exercise

Enabling SNMP

Enabling SNMP (the Simple Network Monitoring Protocol) makes it possible to monitor the health of your network with a wide selection of tools.

In this exercise, you will enable SNMP on your network devices.

Outcomes

You should be able to:

- Add a variable to the **network** group variables file to support template-driven configuration that enables SNMP.
- Update NOS-specific device configuration templates to provide parameterized configuration statements that enable SNMP.
- Perform a multivendor play that configures network devices from the updated Jinja2 templates.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Add a variable to the **network** group variables file to support template-driven configuration that enables SNMP. Add another variable, named **ro_community**, and set the value of this variable to **redhat**. The updated **group_vars/network/vars.yml** variable file should have the following content:

```
ansible_connection: network_cli
domain_name: lab.example.com
nameservers:
- 8.8.8.8
- 8.8.4.4
syslog_ipv4: 172.25.250.254
ro_community: redhat
snmp_clients:
- 172.25.250.254
- 172.25.250.9
```

- 2. Update NOS-specific device configuration templates to provide parameterized configuration statements that enable SNMP.
- 2.1. Add a line to the device configuration Jinja2 template for VyOS devices. It should map appropriate variables to the VyOS statements that enable SNMP. The updated **j2/vyos-config.j2** template should have the following content:

```
set system host-name {{ inventory_hostname }}
set system domain-name {{ domain_name }}
{% for nameserver in nameservers %}
```

```
set system name-server {{ nameserver }}
{% endfor %}
set system syslog host {{ syslog_ipv4 }} facility local7 level {{ vyos_loglevel }}
set service snmp community {{ ro_community }} authorization ro
{% for snmp_client in snmp_clients %}
set service snmp community {{ ro_community }} client {{ snmp_client }}
{% endfor %}
```

- 2.2. Add lines to the device configuration Jinja2 template for IOS devices. It should map appropriate variables to the IOS statements that enable SNMP. The updated **j2/ios-config.j2** template should have the following content:

```
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
{% for nameserver in nameservers %}
ip name-server {{ nameserver }}
{% endfor %}
service timestamps log datetime
service timestamps debug datetime
logging {{ syslog_ipv4 }}
logging trap {{ ios_loglevel }}
access-list 1 permit {{ workstation_ipv4 | ipaddr('address') }} log
access-list 1 permit {{ tower_ipv4 | ipaddr('address') }} log
snmp-server community {{ ro_community }} RO 1
```



Note

The **snmp-server** command only supports standard access lists.

- 3. Perform a multivendor play that configures network devices from the updated Jinja2 templates.
- 3.1. Perform the play found in the **j2cfg.yml** file. You already created the **j2cfg.yml** playbook, which sources configuration statements from the VyOS and IOS Jinja2 templates.

```
[student@workstation proj]$ ansible-playbook j2cfg.yml
```

- 3.2. Install the SNMP network management utilities on **workstation**.

```
[student@workstation proj]$ sudo yum install net-snmp-utils
```

- 3.3. Verify that SNMP is now enabled and working properly. On **workstation**, run **snmpwalk** using the read-only community string you defined (**redhat**). Filter the results based on the pattern **sysName**.

```
[student@workstation proj]$ snmpwalk -v1 -c redhat spine01 sysName
SNMPv2-MIB::sysName.0 = STRING: spine01
[student@workstation proj]$ snmpwalk -v1 -c redhat spine02 sysName
SNMPv2-MIB::sysName.0 = STRING: spine02
[student@workstation proj]$ snmpwalk -v1 -c redhat leaf01 sysName
```

```
SNMPv2-MIB::sysName.0 = STRING: leaf01
[student@workstation proj]$ snmpwalk -v1 -c redhat leaf02 sysName
SNMPv2-MIB::sysName.0 = STRING: leaf02
[student@workstation proj]$ snmpwalk -v1 -c redhat cs01 sysName
SNMPv2-MIB::sysName.0 = STRING: cs01.lab.example.com
```

This concludes the guided exercise.

Overcoming Real-world Challenges

Objectives

After completing this section, you should be able to employ Ansible to verify reachability between two network devices.

Verifying Reachability With ICMP

The Internet Control Message Protocol (ICMP), defined in RFC 792, is part of the Internet Protocol Suite. It was designed to provide a mechanism by which connecting devices communicate with end stations, such as a source hosts. It is best known today for testing reachability.

It is often useful to control which interface traffic is being sourced from when testing reachability. Both IOS and VyOS support this on the CLI, using slightly different syntaxes.

IOS extended ping is most familiar in a menu-driven form, but supports a single-line form. Many options are available; the following is a simple example:

```
# ping destination source source-ip-or-interface repeat repeatcount
```

The following is an example of the extended form of ping available under VyOS:

```
$ ping destination interface source-ip-or-interface count repeatcount
```

Generating ping Packets

The use of ping to verify reachability is illustrated here on VyOS.

The method is nearly identical with IOS, but uses the **vyos_command** module instead of **ios_command**, and uses the slightly different VyOS extended ping command syntax.

```
---
- name: test reachability using ICMP
  hosts: vyos
  gather_facts: no
  vars:
    count: 3
    # (to) and (fr) vars (destination and source interface IPs) are --extra-vars
    cmds: [ "ping {{ to }} interface {{ fr }} count {{ count }}" ]

  tasks:

    - name: on {{ inventory_hostname }} to {{ to }} from {{ fr }}
      vyos_command:
        commands: "{{ cmds }}"
      register: ping_result
      when: ansible_network_os == 'vyos'
```

```
- name: display the result
  debug:
    msg: "{{ ping_result }}"
```

Viewing Results

The play from the example is performed and the results are shown here.

```
$ ansible-playbook -l spine01 -e"fr=10.10.5.1 to=10.10.5.2" \
> vyos-sourced-icmp-ping.yml

PLAY [test reachability using ICMP]
*****
TASK [on spine01 to 10.10.5.2 from 10.10.5.1]
*****
ok: [spine01]

TASK [display the result]
*****
ok: [spine01] => {
  "msg": {
    "changed": false,
    "failed": false,
    "stdout": [
      "PING 10.10.5.2 (10.10.5.2) from 10.10.5.1 : 56(84) bytes of
data.\n64 bytes from 10.10.5.2: icmp_req=1 ttl=64 time=0.236 ms\n64 bytes
from 10.10.5.2: icmp_req=2 ttl=64 time=0.287 ms\n64 bytes from 10.10.5.2:
icmp_req=3 ttl=64 time=0.278 ms\n\n--- 10.10.5.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms\nrtt min/avg/max/mdev =
0.236/0.267/0.287/0.022 ms"
    ],
    "stdout_lines": [
      [
        "PING 10.10.5.2 (10.10.5.2) from 10.10.5.1 : 56(84) bytes of
data.",
        "64 bytes from 10.10.5.2: icmp_req=1 ttl=64 time=0.236 ms",
        "64 bytes from 10.10.5.2: icmp_req=2 ttl=64 time=0.287 ms",
        "64 bytes from 10.10.5.2: icmp_req=3 ttl=64 time=0.278 ms",
        "",
        "--- 10.10.5.2 ping statistics ---",
        "3 packets transmitted, 3 received, 0% packet loss, time 1998ms",
        "rtt min/avg/max/mdev = 0.236/0.267/0.287/0.022 ms"
      ]
    ]
  }
}

PLAY RECAP
*****
spine01 : ok=2    changed=0    unreachable=0    failed=0
```

Redesigning For Consolidation

The consolidation phase: cloud plus a data center.

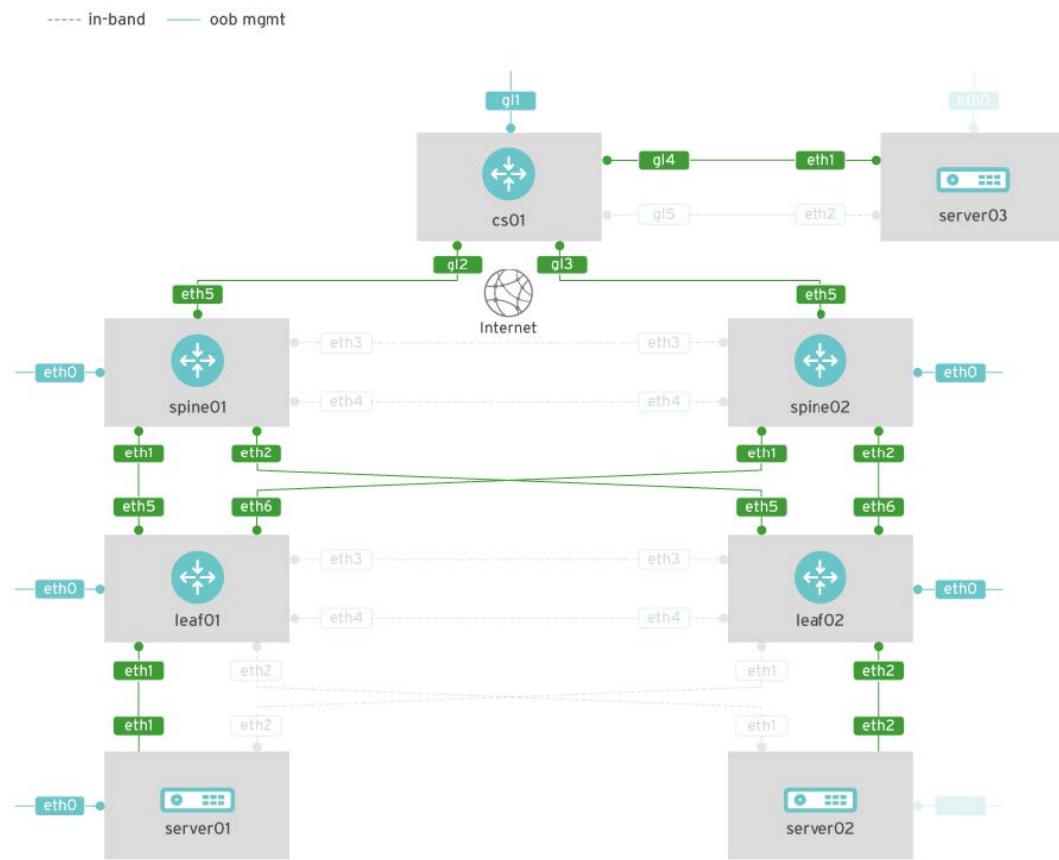


Figure 6.1: The consolidation phase

► Guided Exercise

Provisioning the Consolidation Network

In the consolidation phase of **example.com**, the non-cloud Production Services network devices and server have been temporarily relocated to a data center. A third server is being added: **server02**. Redundant network devices are to be provisioned in the data center named **spine02** and **leaf02**. For now, there is only one connection to the internet (interface **eth1** on **spine01**).

We know that the **eth1** interface on **spine01** is actually a link to the **GigabitEthernet2** interface of **cs01**, but for the sake of the story, we imagine this Ethernet interface on **spine01** (**eth1**, that is) is connected to the Network Termination Equipment (NTE) that marks the demarcation point (Demarc) between **example.com** as the customer and the Internet Service Provider (ISP) providing internet access to the data center.

The Production Services Network of **example.com**, during this phase, consists of five network devices at two different locations, plus three servers:

1. In the cloud
 - a. Network devices
 - i. **cs01**
 - b. Servers
 - i. **server03**
2. At the office
 - a. Network devices
 - i. **spine01**
 - ii. **spine02**
 - iii. **leaf01**
 - iv. **leaf02**
 - b. Servers
 - i. **server01**
 - ii. **server02**

Jasper asks you to compose a playbook that will:

1. Apply interface descriptions to interfaces as documented in the table.
2. Configure layer 3 on interfaces as documented in the table.
3. Enable OSPF and configure it to advertise network routes appropriately.
4. Verify that the servers are reachable from the network.

You will be accessing the network devices by way of management interfaces. Those interfaces are already configured with respect to layer 3 and up, and should not be changed.

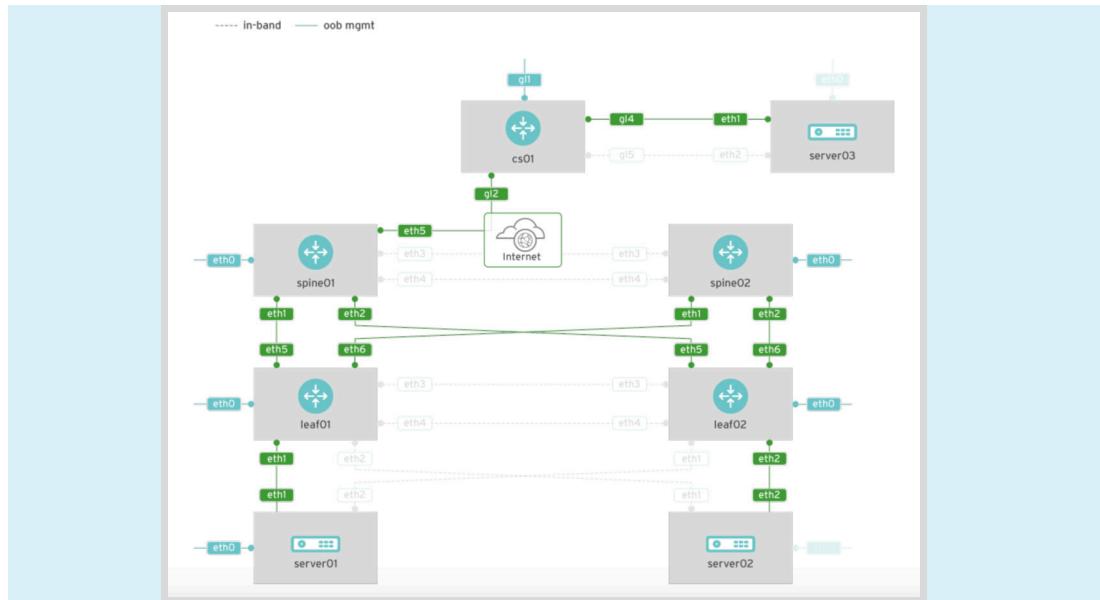


Figure 6.2: The Consolidation Phase Production Services Network of example.com

Consolidation Phase Interface Descriptions

Devices	Interfaces	Descriptions
cs01	GigabitEthernet1	management
	GigabitEthernet2	outside
	GigabitEthernet4	inside
spine01	eth0	management
	eth1	leaf01
	eth2	leaf02
	eth5	outside
spine02	eth0	management
	eth1	leaf01
	eth2	leaf02
leaf01	eth0	management
	eth1	server01
	eth5	spine01
	eth6	spine02
leaf02	eth0	management
	eth2	server02

Devices	Interfaces	Descriptions
	eth5	spine01
	eth6	spine02

Consolidation Phase Layer 3 Addressing (management interface not shown)

Devices	Interfaces	Descriptions
cs01	Loopback1	172.16.0.1/32
	GigabitEthernet2	172.16.2.2/30
	GigabitEthernet4	172.16.10.1/30
spine01	lo	10.0.0.1/32
	eth1	10.10.5.1/30
	eth2	10.10.6.1/30
	eth5	172.16.2.1/30
leaf01	lo	10.0.0.11/32
	eth1	10.10.10.1/30
	eth5	10.10.5.2/30
	eth6	10.10.7.2/30
spine02	lo	10.0.0.2/32
	eth1	10.10.7.1/30
	eth2	10.10.8.1/30
leaf02	lo	10.0.0.12/32
	eth2	192.168.10.1/30
	eth5	10.10.6.2/30
	eth6	10.10.8.2/30

In this exercise, you will provision the **example.com** Production Services network that corresponds to the consolidation phase.

Outcomes

You should be able to:

- Create a **vars** file defining the variables that will be used.
- Compose a playbook with a play that satisfies the business requirements as stated by Jasper.

- Perform the play in the playbook to enact the desired changes.
- Verify that the outcome is as intended.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Create a **vars** file defining the variables that will be used. Download the **consolidation-data.yml** file into the **vars/** directory. This file maps layer 3 address data to interfaces and also provides interface descriptions.

```
[student@workstation proj]$ cd vars  
[student@workstation vars]$ wget \  
> http://materials.example.com/full/vars/consolidation-data.yml  
[student@workstation vars]$ cd ..  
[student@workstation proj]$
```

- 2. Compose a playbook with a play that satisfies the business requirements as stated by Jasper.

Create a playbook named **consolidation.yml** as shown here.

Notice how much simpler this playbook is than the one that was used for the expansion phase. That is intentional. The difference between the two playbooks illustrates how effective **with_dict** can be.

```
---  
- name: consolidate the layer 3 network  
  hosts: network  
  vars_files:  
    - vars/consolidation-data.yml  
  
  tasks:  
  
    - name: configure interface descriptions  
      vyos_interface:  
        name: "{{ item.key }}"  
        description: "{{ item.value.description }}"  
        with_dict: "{{ interface_data[inventory_hostname] }}"  
        when: ansible_network_os == 'vyos'  
  
    - name: configure layer 3  
      vyos_l3_interface:  
        aggregate: "{{ layer3_data[inventory_hostname] }}"  
        when: ansible_network_os == 'vyos'  
  
    - name: configure interface descriptions  
      ios_interface:  
        name: "{{ item.key }}"  
        description: "{{ item.value.description }}"  
        with_dict: "{{ interface_data[inventory_hostname] }}"  
        when: ansible_network_os == 'ios'
```

```
- name: configure layer 3
  ios_l3_interface:
    aggregate: "{{ layer3_data[inventory_hostname] }}"
  when: ansible_network_os == 'ios'
```

- 3. Perform the play in the playbook to enact the desired changes.

```
[student@workstation proj]$ ansible-playbook consolidation.yml
```

- 4. Execute ad hoc commands and verify that layer 3 addresses and interface descriptions are mapped to interfaces as described in the tables found at the start of this exercise. Here the syntax of the command is shown for use with the VyOS devices:

```
[student@workstation proj]$ ansible -m vyos_command -a "commands='sh int'" vyos
```

The following ad hoc command can be used to verify that layer 3 addresses are listed correctly for the IOS networking device **cs01**. The following command should be entered as a single line.

```
[student@workstation proj]$ ansible -m ios_command -a "commands='sh ip int br'" cs01
```

This concludes the guided exercise.

Implementing Dynamic Routing with OSPF

Objectives

After completing this section, you should be able to use Ansible to configure the OSPF routing protocol on network devices.

Implementing OSPF

Network administration may involve implementing, managing, or troubleshooting dynamic routing protocols.

The following are minimal OSPF configurations for a VyOS and an IOS device:

VyOS:

```
set interfaces loopback lo address 10.0.0.1/32
set protocols ospf parameters router-id 10.0.0.1
set protocols ospf area 0 network 10.10.1.0/24
set protocols ospf log-adjacency-changes
```

IOS:

```
no router ospf 1
router ospf 1
router-id 172.16.0.1
network 172.16.2.0 0.0.0.3 area 0
network 172.16.5.0 0.0.0.3 area 0
network 172.16.10.0 0.0.0.3 area 0
```

In the IOS example, 1 is the process id of the OSPF process. In both, the router id is set to a loopback IPv4 address. This is a single area example, all routers in area 0.

Developing Templates for OSPF

Statements such as the ones listed on the previous slide are easy to convert into playbook tasks using suitable ***os_config** modules (using **vyos_config** and **ios_config**, in this case). A more flexible way to manage the process, though, uses **Jinja2** templates in conjunction with ***os_config** modules to generate and apply blocks of configuration statements.

```
---
- name: Play that sets up OSPF on the VyOS devices
  hosts: vyos
  gather_facts: no
  vars_files:
    - "{{ playbook_dir }}/vars/ospf-vars.yml"

  tasks:
    - name: configure devices
```

```
vyos_config:  
  src: j2/vyos-ospf.j2  
  save: yes  
  
- name: Play that sets up OSPF on the IOS devices  
  hosts: ios  
  gather_facts: no  
  vars_files:  
    - "{{ playbook_dir }}/vars/ospf-vars.yml"  
  
tasks:  
  - name: configure devices  
    ios_config:  
      src: j2/ios-ospf.j2  
      save: yes
```

► Guided Exercise

Implementing Dynamic Routing with OSPF

In this exercise, you will automate the process of configuring dynamic routing with OSPF.

Outcomes

You should be able to:

- Compose NOS-specific device configuration templates to provide parameterized configuration statements.
- Perform a play that configures OSPF on network devices.
- Verify that the outcome is as intended.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Compose NOS-specific device configuration templates to provide parameterized configuration statements.
- 1.1. Create a Jinja2 template that constructs statements to configure OSPF on VyOS devices. Create a file named **j2/vyos-ospf.j2** with the following content:

```
{% for intf in layer3_data[inventory_hostname] %}  
{% if intf.name == 'lo' %}  
set protocols ospf parameters router-id {{ intf.ipv4 | ipaddr('address') }}  
set protocols ospf log-adjacency-changes  
set protocols ospf redistribute connected metric-type 2  
{% if inventory_hostname == 'spine01' or inventory_hostname == 'spine02' %}  
set protocols ospf default-information originate always  
set protocols ospf default-information originate metric 10  
set protocols ospf default-information originate metric-type 2  
{% endif %}  
{% else %}  
set protocols ospf area 0 network {{ intf.ipv4 | ipaddr('network/prefix') }}  
{% endif %}  
{% endfor %}
```

- 1.2. Create a Jinja2 template that constructs statements to configure OSPF on IOS devices. Create a file named **j2/ios-ospf.j2** with the following content: Note the line starting with **network** may have wrapped; it should end with **area 0**.

```
router ospf 1
{% for intf in layer3_data[inventory_hostname] %}
{% if intf.name.startswith('Loopback') %}
  router-id {{ intf.ipv4 | ipaddr('address') }}
{% else %}
  network {{ intf.ipv4 | ipaddr('network') }} {{ intf.ipv4 | ipaddr('hostmask') }}
  area 0
{% endif %}
{% endfor %}
```

**Important**

There should be a single, very long line between `{% else %}` and `{% endif %}` that starts with a space and the word **network** and ends with **area 0**. The line might be wrapped due to length in this student guide.

▶ 2. Perform a play that configures OSPF on network devices.

- 2.1. Compose a playbook that configures OSPF based on data and templates. Create a file named **ospf-consolidation.yml** with the following content:

```
---
- name: play that configures OSPF according to data and templates
  hosts: network
  vars:
    vyos_ospf_template: j2/vyos-ospf.j2
    ios_ospf_template: j2/ios-ospf.j2

  vars_files:
    - vars/consolidation-data.yml

  tasks:
    - name: configure ospf on vyos
      vyos_config:
        src: "{{ vyos_ospf_template }}"
        save: True
      when: ansible_network_os == 'vyos'

    - name: configure ospf on ios
      ios_config:
        src: "{{ ios_ospf_template }}"
        save_when: changed
      when: ansible_network_os == 'ios'
```

- 2.2. Check playbook syntax, then perform the play. If you are having trouble, you may refer to <http://materials.example.com/full> from your workstation.

```
[student@workstation proj]$ ansible-playbook --syntax-check \
> ospf-consolidation.yml

Playbook: ospf-consolidation.yml
[student@workstation proj]$ ansible-playbook ospf-consolidation.yml
```

- 3. Execute an ad hoc command that displays IP routes on VyOS device **spine01**.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh ip ro'" spine01
spine01 | SUCCESS => {
...output omitted...
  "stdout_lines": [
    [
      "Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF",
      "I - ISIS, B - BGP, > - selected route, * - FIB route",
      "",
      "0  0.0.0.0/0 [110/10] via 10.10.5.2, 00:07:07",
      "S>* 0.0.0.0/0 [1/0] via 172.16.2.2, eth5",
      "C>* 10.0.0.1/32 is directly connected, lo",
      "O>* 10.0.0.2/32 [110/20] via 10.10.5.2, eth1, 00:11:03",
      " *           via 10.10.6.2, eth2, 00:11:03",
      "O>* 10.0.0.11/32 [110/20] via 10.10.5.2, eth1, 00:11:03",
      "O>* 10.0.0.12/32 [110/20] via 10.10.6.2, eth2, 00:11:13",
      "O  10.10.5.0/30 [110/10] is directly connected, eth1, 00:12:08",
      "C>* 10.10.5.0/30 is directly connected, eth1",
      "O  10.10.6.0/30 [110/10] is directly connected, eth2, 00:12:08",
      "C>* 10.10.6.0/30 is directly connected, eth2",
      "O>* 10.10.7.0/30 [110/20] via 10.10.5.2, eth1, 00:11:04",
      "O>* 10.10.8.0/30 [110/20] via 10.10.6.2, eth2, 00:11:14",
      "O  10.10.10.0/30 [110/20] via 10.10.5.2, 00:11:04",
      "S>* 10.10.10.0/30 [1/0] via 10.10.5.2, eth1",
      "C>* 127.0.0.0/8 is directly connected, lo",
      "O  172.16.2.0/30 [110/10] is directly connected, eth5, 00:12:07",
      "C>* 172.16.2.0/30 is directly connected, eth5",
      "O>* 172.16.10.0/30 [110/11] via 172.16.2.2, eth5, 00:12:03",
      "O  172.25.250.0/24 [110/20] via 10.10.5.2, 00:11:03",
      " *           via 10.10.6.2, 00:11:03",
      "C>* 172.25.250.0/24 is directly connected, eth0",
      "O>* 192.168.10.0/30 [110/20] via 10.10.6.2, eth2, 00:11:14"
    ]
  ]
}
```

- 4. Execute an ad hoc command that displays IP routes on IOS device **cs01**.

```
[student@workstation proj]$ ansible -m ios_command -a "commands='sh ip ro'" cs01
cs01 | SUCCESS => {
...output omitted...
  "stdout_lines": [
    [

```

```

        "Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B -
BGP",
        "      D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
",
        "      N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type
2",
        "      E1 - OSPF external type 1, E2 - OSPF external type 2",
        "      i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS
level-2",
        "      ia - IS-IS inter area, * - candidate default, U - per-user
static route",
        "      o - ODR, P - periodic downloaded static route, H - NHRP, l -
LISP",
        "      a - application route",
        "      + - replicated route, % - next hop override, p - overrides
from PfR",
        """",
        "0*E2 0.0.0.0/0 [110/10] via 172.16.2.1, 00:11:46, GigabitEthernet2",
        "  10.0.0.0/8 is variably subnetted, 8 subnets, 2 masks",
        "0 E2 10.0.0.2/32 [110/20] via 172.16.2.1, 00:11:46,
GigabitEthernet2",
        "0 E2 10.0.0.11/32 [110/20] via 172.16.2.1, 00:11:46,
GigabitEthernet2",
        "0 E2 10.0.0.12/32 [110/20] via 172.16.2.1, 00:11:46,
GigabitEthernet2",
        "0 10.10.5.0/30 [110/11] via 172.16.2.1, 00:11:36,
GigabitEthernet2",
        "0 10.10.6.0/30 [110/11] via 172.16.2.1, 00:11:56,
GigabitEthernet2",
        "0 10.10.7.0/30 [110/21] via 172.16.2.1, 00:11:36,
GigabitEthernet2",
        "0 10.10.8.0/30 [110/21] via 172.16.2.1, 00:11:46,
GigabitEthernet2",
        "0 10.10.10.0/30 [110/21] via 172.16.2.1, 00:11:36,
GigabitEthernet2",
        "  172.16.0.0/16 is variably subnetted, 5 subnets, 2 masks",
"C 172.16.0.1/32 is directly connected, Loopback1",
"C 172.16.2.0/30 is directly connected, GigabitEthernet2",
"L 172.16.2.2/32 is directly connected, GigabitEthernet2",
"C 172.16.10.0/30 is directly connected, GigabitEthernet4",
"L 172.16.10.1/32 is directly connected, GigabitEthernet4",
"  172.25.0.0/16 is variably subnetted, 2 subnets, 2 masks",
"C 172.25.250.0/24 is directly connected, GigabitEthernet1",
"L 172.25.250.195/32 is directly connected, GigabitEthernet1",
"  192.168.10.0/30 is subnetted, 1 subnets",
"0 192.168.10.0 [110/21] via 172.16.2.1, 00:11:46,
GigabitEthernet2"
        "0 192.168.5.0 [110/11] via 172.16.5.1, 00:03:58,
GigabitEthernet3",
        "  192.168.10.0/30 is subnetted, 1 subnets",

```

```
"0      192.168.10.0 [110/21] via 172.16.5.1, 00:03:58,
GigabitEthernet3"
]
}
}
```

This concludes the guided exercise.

► Guided Exercise

Verifying End-to-End Reachability

In this scenario, it is important to the networking team that the devices they manage can be reached by way of the management network. The customer served by the networking team, though, cares about connectivity from end station to end station. The closest point to an end station that falls within the responsibility of the networking team is the access interface through which traffic enters or exits the network.

A common troubleshooting pattern involves segmenting the problem domain into the network, the two local spans, and the two end stations. Local spans can be relatively easy to verify.

Notice that the assertions built into this exercise test for 0% packet loss or 100% success. When MAC addresses are not already available in the ARP cache, though, Cisco's **ping** lists the first few packets as discarded.



Note

Assertions of 0% packet loss or 100% success may fail the first time, until MAC addresses are cached. In this exercise, if the play fails unexpectedly on the assertions, try rerunning the playbook first.

In this exercise, you will automate the process of verifying end-to-end reachability across the network.

Outcomes

You should be able to:

- Update the host inventory to support end-to-end reachability testing.
- Add variables that identify ingress and egress points as source and destination for ping test.
- Perform a play that verifies reachability across the network on an end-to-end basis.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Update the host inventory to support end-to-end reachability testing. Ensure that the hosts inventory contains a group that identifies some devices as access-layer routers. Modify the **inventory** file so it contains content similar to the following:

```
[leafs]
leaf[01:02]
```

```
[spines]
spine[01:02]
```

```
[border-routers]
spine01
spine02
cs01

[access-layer]
leaf01
leaf02
cs01

[cloud-services]
cs01

[ios]
cs01

[vyos:children]
spines
leafs

[network:children]
vyos
ios
```

- ▶ 2. Add variables that identify ingress and egress points as source and destination for ping test. Create a variables file named **vars/ping-srcdst.yml** that contains this data:

```
pingcount: 2
ping_data:
  leaf01:
    - { src: "10.10.10.1", dst: "192.168.10.1" }
    - { src: "10.10.10.1", dst: "172.16.10.1" }
  leaf02:
    - { src: "192.168.10.1", dst: "10.10.10.1" }
    - { src: "192.168.10.1", dst: "172.16.10.1" }
  cs01:
    - { src: "172.16.10.1", dst: "10.10.10.1" }
    - { src: "172.16.10.1", dst: "192.168.10.1" }
```

It is possible to use Ansible and Jinja2 to automatically generate the data in this variables file, but for now we are going to keep matters as simple as possible and start with this data.

- ▶ 3. Perform a play that verifies reachability across the network on an end-to-end basis.
- 3.1. Compose a multivendor playbook that loops over the ping data tuples for each host and pings from source to destination. Include a task that loops over the result set and asserts that output from the ping test matches patterns that reliably indicate a successful test.

Create a file named **e2e.yml** with content similar to the following:

```
---
- name: verify connectivity end-to-end
  hosts: access-layer
  vars_files:
```

```

- vars/ping-srcdst.yml

tasks:

- name: run ping commands on VyOS access layer device
  # this runs a ping command across the link
  vyos_command:
    commands:
      - ping {{ item.dst }} interface {{ item.src }} count {{ pingcount }}
  register: ping_result
  loop: "{{ ping_data[inventory_hostname] }}"
  when: ansible_network_os == 'vyos'
  # registering within loop associates values with varname.results

- name: looped assertion of ping results from VyOS access layer device
  assert:
    that: "'0% packet loss' in item.stdout[0]"
  loop: "{{ ping_result.results }}"
  when: ansible_network_os == 'vyos'

- name: prime IOS arp cache
  ios_command:
    commands:
      - ping {{ item.dst }} source {{ item.src }} repeat 1
  loop: "{{ ping_data[inventory_hostname] }}"
  when: ansible_network_os == 'ios'

- name: "run ping commands on IOS access layer device {{ inventory_hostname }}"
  ios_command:
    commands:
      - ping {{ item.dst }} source {{ item.src }} repeat {{ pingcount }}
  register: ping_result
  loop: "{{ ping_data[inventory_hostname] }}"
  when: ansible_network_os == 'ios'

- name: looped assertion of ping results from IOS access layer device
  assert:
    that: "'Success rate is 100 percent' in item.stdout[0]"
  loop: "{{ ping_result.results }}"
  when: ansible_network_os == 'ios'

```

3.2. Perform the play found in your new playbook.

```
[student@workstation proj]$ ansible-playbook e2e.yml
```

If all goes well, the play recap should show **ok=2** and **failed=0** for each device.

```

PLAY RECAP
PLAY RECAP ****
cs01 : ok=2    changed=0    unreachable=0    failed=0
leaf01 : ok=2    changed=0    unreachable=0    failed=0
leaf02 : ok=2    changed=0    unreachable=0    failed=0

```

This concludes the guided exercise.

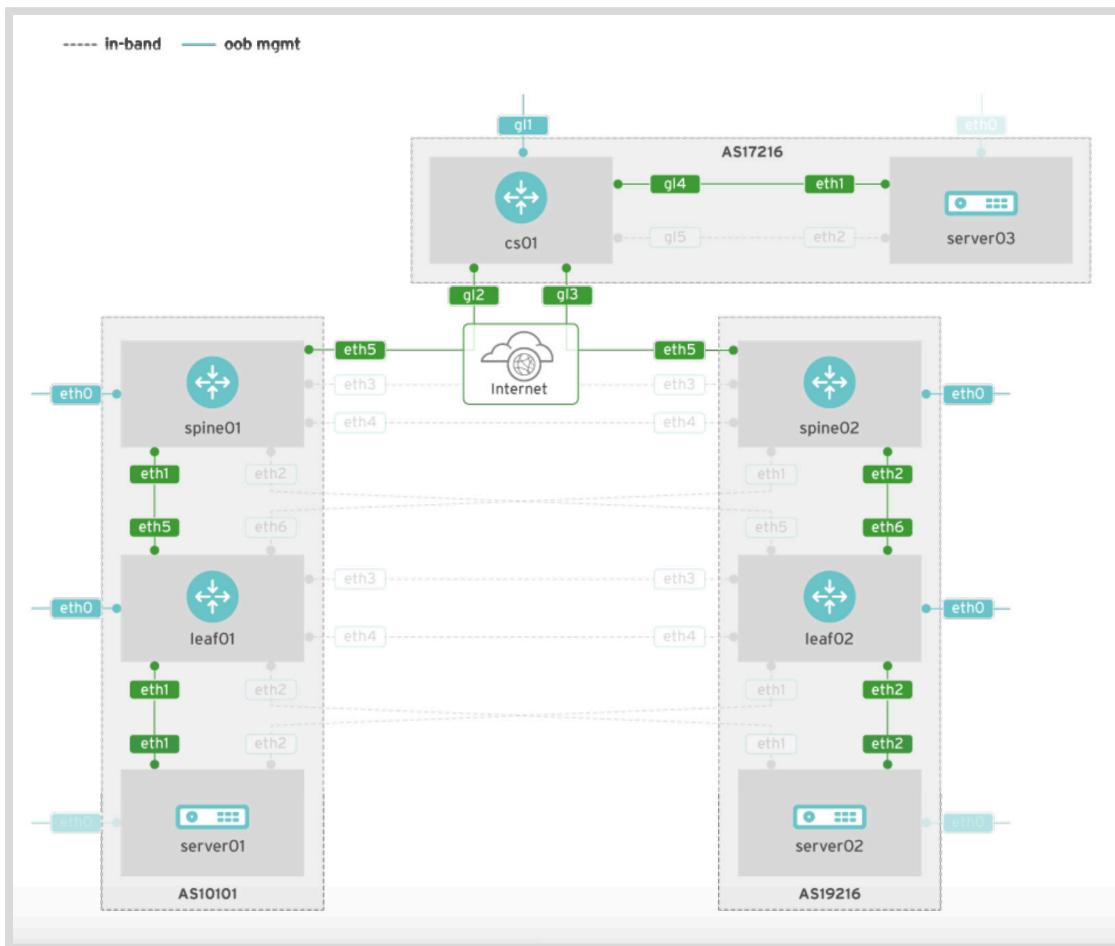
Implementing OSPF with Multiple Autonomous Systems

Objectives

After completing this section, you should be able to automate the configuration of your network routers to use OSPF between multiple autonomous systems.

Provisioning the Break Up Network

The next automation scenario will use Ansible to configure three separate networks for three businesses. Each network will be an independent autonomous system (AS).



► Guided Exercise

Provisioning the Break Up Network

In this scenario, **example.com** is in a break-up phase. You are now part of a team headed by Jasper that has been spun off into a different company, named Ideal Networking Incorporated, that manages networks for many different businesses. What used to be the Production Services Network of **example.com** is now three different networks, owned by three different companies. The three companies that used to be **example.com** are Ideal Services Incorporated (ISI), Ideal Products Incorporated (IPI), and Ideal Technologies Incorporated (ITI). They still work closely together with each other. Ideal Networking Incorporated manages all three networks.

The Production Services Network of **example.com**, during this phase, will consist of three different networks associated with three different Autonomous Systems (AS):

1. ITI - **AS17216**
 - a. Network devices
 - i. **cs01** (cloud)
 - b. Servers
 - i. **server03**
2. ISI - **AS10101**
 - a. Network devices
 - i. **spine01**
 - ii. **leaf01**
 - b. Servers
 - i. **server01**
3. IPI - **AS19216**
 - a. Network devices
 - i. **spine02**
 - ii. **leaf02**
 - b. Servers
 - i. **server02**

In this guided exercise, you will change the layer 3 addresses and interface descriptions according to values found in the *Break Up Phase Layer 3 Addressing* and *Break Up Phase Interface Descriptions* tables.

During a brief interim period, routing information among the networks of the three new companies will be shared by way of the OSPF routing protocol as a single zone: a single routing domain.

In the next guided exercise, though, you will separate the three networks into distinct autonomous systems that share routing information explicitly by way of EBGP; three different routing domains that explicitly share routes in accordance with peering agreements.

These are the immediate business requirements:

1. Apply interface descriptions to interfaces as documented in the table.
2. Configure layer 3 on interfaces as documented in the table.
3. Configure dynamic routing with OSPF using a single zone.

The decision to use a single zone instead of multiple OSPF zones was made because management knows that the network is to be divided, in the next guided exercise, into three different autonomous systems that do not use OSPF to advertise routes between each other. After that change is complete, there will be three small networks, each consisting of a single zone.

The process of separating the old, consolidated **example.com** network into three independent networks involves a significant amount of change, including many deletions. Nothing is being removed on the IOS device, but many changes need to happen on the VyOS devices.

Given the extraordinary nature of this change, management has given approval for the VyOS devices to be reinitialized with respect to the non-management interfaces. This makes your job easier, because (1) you already have a playbook that can do this, and (2) you can build a new playbook that provisions the three networks without the additional complexity associated with surgical removal of legacy resources.

Management also has given approval to perform the change in two phases. The outcome of the first phase is expected to be end-to-end reachability with routes by way of dynamic routing using OSPF. During the first phase, it is acceptable for routes to be advertised by way of OSPF across the boundaries of the three networks.

The second part of the Break Up scenario is described in the next guided exercise.

You will be accessing the network devices by way of management interfaces. Those interfaces are already configured with respect to layer 3 and up, and should not be changed.

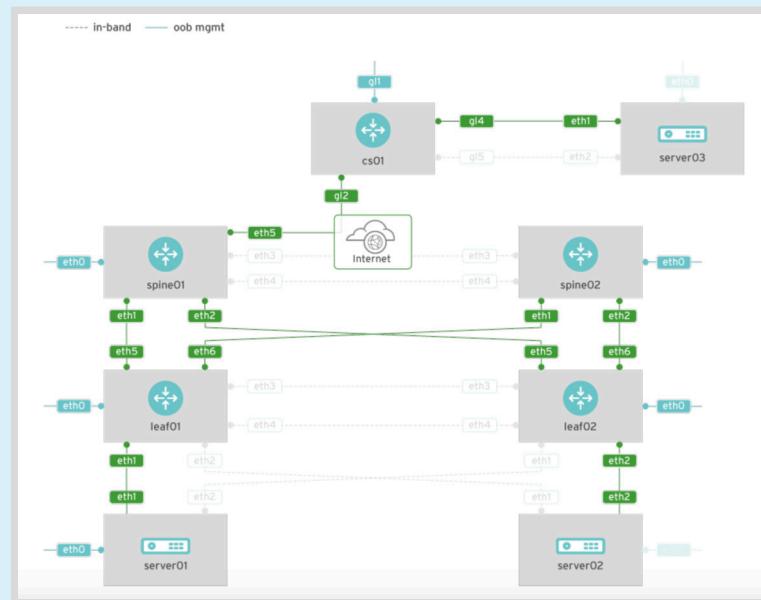


Figure 6.4: The Consolidation Phase Production Services Network of example.com

Break Up Phase Interface Descriptions

Devices	Interfaces	Descriptions
cs01	GigabitEthernet1	management
	GigabitEthernet2	outside-as10101

Devices	Interfaces	Descriptions
	GigabitEthernet3	outside-as19216
	GigabitEthernet4	inside
spine01	eth0	management
	eth1	leaf01
	eth5	outside-as17216
spine02	eth0	management
	eth2	leaf02
	eth5	outside-as17216
leaf01	eth0	management
	eth1	server01
	eth5	uplink
leaf02	eth0	management
	eth2	server02
	eth6	uplink

Break Up Phase Layer 3 Addressing (management interfaces not shown)

Devices	Interfaces	Descriptions
cs01	Loopback1	172.16.0.1/32
	GigabitEthernet2	172.16.2.2/30
	GigabitEthernet3	172.16.5.2/30
	GigabitEthernet4	172.16.10.1/30
spine01	lo	10.0.0.1/32
	eth1	10.10.5.1/30
	eth5	172.16.2.1/30
spine02	lo	192.168.0.1/32
	eth2	192.168.5.1/30
	eth5	172.16.5.1/30
leaf01	lo	10.0.0.11/32

Devices	Interfaces	Descriptions
	eth1	10.10.10.1/30
	eth5	172.16.5.1/30
leaf02	lo	192.168.0.2/32
	eth2	192.168.10.1/30
	eth6	192.168.5.2/30

In this exercise, you will provision the **example.com** Production Services network that corresponds to the break up phase.

Outcomes

You should be able to:

- Reinitialize non-management resources on the VyOS devices.
- Create a variables file that defines the variables that are needed.
- Perform a play that implements layer 3 address and interface description changes.
- Perform a play that implements OSPF configuration changes.
- Verify that the outcome is as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to the **~/proj/** directory.

- 1. Reinitialize non-management resources on the VyOS devices. If you did not create a playbook named **vyos-reinitialize-l3.yml** in a previous guided exercise, download it from <http://materials.example.com/playbooks/>.

```
[student@workstation proj]$ ansible-playbook vyos-reinitialize-l3.yml
```

- 2. Create a variables file that defines the variables that are needed. Download the **breakup-data.yml** file into the **vars/** directory. This file maps layer 3 address data to interfaces and also provides interface descriptions.

```
[student@workstation proj]$ cd vars
[student@workstation vars]$ wget \
> http://materials.example.com/full/vars/breakup-data.yml
[student@workstation vars]$ cd ..
[student@workstation proj]$
```

- 3. Perform a play that implements layer 3 address and interface description changes.

- 3.1. Write a play that implements layer 3 address and interface description changes. Create a file named **breakup.yml** with the following content:

```
---
- name: break up the layer 3 network
  hosts: network
```

```

vars_files:
  - vars/backup-data.yml

tasks:

- name: remove the legacy layer3 data
  vyos_l3_interface:
    aggregate: "{{ remove_layer3_data[inventory_hostname] }}"
    state: absent
  when: ansible_network_os == 'vyos'

- name: configure interface descriptions
  vyos_interface:
    name: "{{ item.key }}"
    description: "{{ item.value.description }}"
  with_dict: "{{ interface_data[inventory_hostname] }}"
  when: ansible_network_os == 'vyos'

- name: configure layer 3
  vyos_l3_interface:
    aggregate: "{{ layer3_data[inventory_hostname] }}"
  when: ansible_network_os == 'vyos'

- name: configure interface descriptions
  ios_interface:
    name: "{{ item.key }}"
    description: "{{ item.value.description }}"
  with_dict: "{{ interface_data[inventory_hostname] }}"
  when: ansible_network_os == 'ios'

- name: configure layer 3
  ios_l3_interface:
    aggregate: "{{ layer3_data[inventory_hostname] }}"
  when: ansible_network_os == 'ios'

```

- 3.2. Perform the play found in the **breakup.yml** playbook. This provisions the networks at layer 3.

```
[student@workstation proj]$ ansible-playbook breakup.yml
```

- 3.3. Execute ad hoc commands and verify that layer 3 addresses and interface descriptions are mapped to interfaces as described in the tables found at the start of this exercise. Here the syntax of the command is shown for use with the VyOS devices:

```
[student@workstation proj]$ ansible -m vyos_command -a "commands='sh int'" vyos
```

The following ad hoc command can be used to verify that layer 3 addresses are listed correctly for the IOS networking device **cs01**.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh ip int br'" cs01
```

▶ 4. Perform a play that implements OSPF configuration changes.

- 4.1. Copy the existing **ospf-consolidation.yml** file and modify it to create a new **ospf-breakup.yml** playbook.

```
[student@workstation proj]$ cp ospf-consolidation.yml ospf-breakup.yml
```

The only change is to use the **breakupdata.yml** variables file. After editing, the **ospf-breakup.yml** file should contain the following:

```
---
```

```
- name: play that configures OSPF according to data and templates
  hosts: network
  vars:
    vyos_ospf_template: j2/vyos-ospf.j2
    ios_ospf_template: j2/ios-ospf.j2

  vars_files:
    - vars/breakup-data.yml

  tasks:
    - name: configure ospf on vyos
      vyos_config:
        src: "{{ vyos_ospf_template }}"
        save: True
      when: ansible_network_os == 'vyos'

    - name: configure ospf on ios
      ios_config:
        src: "{{ ios_ospf_template }}"
        save_when: changed
      when: ansible_network_os == 'ios'
```

4.2. Perform the play in the **ospf-breakup.yml** playbook.

```
[student@workstation proj]$ ansible-playbook ospf-breakup.yml
```

4.3. Execute ad hoc commands to confirm that the desired routes are present.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh ip ro'" vyos
[student@workstation proj]$ ansible -m ios_command -a "commands='sh ip ro'" ios
```

▶ 5. Verify that the outcome is as desired. Perform the end-to-end reachability test using the **e2e.yml** playbook.

```
[student@workstation proj]$ ansible-playbook e2e.yml
```

This concludes the guided exercise.

Implementing Dynamic Routing with EBGP

Objectives

After completing this section, you should be able to use Ansible to configure the External Border Gateway Protocol (EBGP) to perform dynamic routing between multiple autonomous systems.

Implementing EBGP

Other routing protocols, such as EBGP, for instance, can be implemented in a similar way.

The following is a minimal configuration for EBGP on a VyOS device:

```
set interfaces loopback lo address 10.0.0.1/32
set protocols bgp 100 parameters router-id '10.0.0.1'
set protocols bgp 100 network 10.10.1.0/24
set protocols bgp 100 network 10.10.10.0/24
set protocols bgp 100 neighbor 172.25.250.61 remote-as '200'
```

This configuration:

- Binds IPv4 address 10.0.0.1/32 to the loopback interfaces.
- Configures the BGP router-id as the loopback address.
- Defines the networks known to this router.
- Defines the BGP neighbor.

Describing a VyOS Template for EBGP

A Jinja2 template takes BGP data and uses it to produce VyOS EBGP configuration statements.

```
$ cat j2/vyos-bgp.j2
set protocols bgp {{ bgp_data[inventory_hostname]['as'] }} parameters router-id
{{ bgp_data[inventory_hostname]
['router-id'] | ipaddr("address") }}
{% for nei in bgp_data[inventory_hostname]['neighbors'] -%}
set protocols bgp {{ bgp_data[inventory_hostname]['as'] }} neighbor {{ nei['ipv4']
| ipaddr('address') }} remote-as
{{ nei['as'] }}
{% endfor -%}
{% for net in bgp_data[inventory_hostname]['networks'] -%}
set protocols bgp {{ bgp_data[inventory_hostname]['as'] }} network {{ net }}
{% endfor -%}
```

The following playbook uses the template to implement EBGP on a VyOS device:

```
- name: Play that sets up eBGP on the spine devices
hosts: spines
gather_facts: no
vars_files:
```

```
- "{{ playbook_dir }}/vars/bgp-vars.yml

tasks:
  - name: configure devices
    vyos_config:
      src: j2/vyos-bgp.j2
      save: yes
```

► Guided Exercise

Implementing Dynamic Routing with EBGP

This is a continuation of the process of implementing the Break Up network scenario for **example.com**. In this Guided Exercise, dynamic advertisement of routes by way of OSPF will not be permitted across network boundaries. Instead, routes will be explicitly advertised by way of EBGP.

For the purpose of the story, it is to be imagined that the links between networks happen by way of ISP connections to the public internet.

External Border Gateway Protocol is a BGP extension used to advertise routes from neighbor to neighbor between distinct autonomous systems (AS).

These are the business requirements for the final part of the Break Up phase of **example.com**:

- OSPF should be enabled within each of the three autonomous systems, configured to advertise network routes inside the system.
- OSPF should not advertise routes across the boundaries of autonomous systems.
- Leaf nodes may have a default route that points to the directly connected layer 3 address of their upstream neighbor.
- Configure EBGP on edge devices. Advertise routes for networks within each AS by way of BGP. Do not advertise infrastructure-only subnets, unless included by way of route summarization. All three companies agree that their servers should be able to reach servers on any of the three networks, but network devices do not need to be reachable from sources outside of the autonomous system where they reside.
- Verify that the servers are reachable from the network, and that all three servers can reach each other.

In this exercise, you will implement EBGP to provide routing and connectivity among otherwise isolated autonomous systems.

Outcomes

You should be able to:

- Convert the fully layer 3 interconnected network into three isolated networks.
- Add variables to support template-driven configuration.
- Create NOS-specific device configuration templates with parameterized configuration statements.
- Perform a multivendor play that configures network devices from the updated Jinja2 templates.
- Verify that the end result is what was intended.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Convert the single, fully layer 3 interconnected OSPF network into three isolated networks.

- 1.1. Prevent **spine01** from sending OSPF route announcements out **eth5** to **cs01**.

```
[student@workstation proj]$ ansible -m vyos_config \
> -a "lines='set prot ospf passive-interface eth5'" spine01
spine01 | SUCCESS => {
    "changed": true,
    "commands": [
        "set prot ospf passive-interface eth5"
    ],
    "filtered": []
}
```

- 1.2. Prevent **spine02** from sending OSPF route announcements out **eth5** to **cs01**.

```
[student@workstation proj]$ ansible -m vyos_config \
> -a "lines='set prot ospf passive-interface eth5'" spine02
spine02 | SUCCESS => {
    "changed": true,
    "commands": [
        "set prot ospf passive-interface eth5"
    ],
    "filtered": []
}
```

- 1.3. Prevent **cs01** from sending OSPF route announcements out **Gi2** and **Gi3**

Create a Jinja2 template file named **j2/ios-ospf-passint-def.j2** with the following content:

```
router ospf 1
  passive-interface default
```

Create an Ansible Playbook named **ios-ospf-passint-def.yml** with the following content:

```
---
- name: set IOS device to OSPF passive interface default
  hosts: cs01
  vars:
    passint_template: j2/ios-ospf-passint-def.j2

  tasks:

    - name: configure OSPF 1 (process-id 1)
      ios_config:
        src: "{{ passint_template }}"
```

Perform the play found in your new playbook.

```
[student@workstation proj]$ ansible-playbook ios-ospf-passint-def.yml
```

```
PLAY [set IOS device to OSPF passive interface default] ****
```

```
TASK [configure OSPF 1 (process-id 1)] *****
changed: [cs01]

PLAY RECAP *****
cs01 : ok=1    changed=1    unreachable=0    failed=0
```

- 1.4. Verify that you now have three, isolated networks.

The **spine01** device should now only know about the management network (172.25.250.*) and 10.* networks.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh ip ro'" spine01
```

The **spine02** device should only known about the management network and 192.168.* networks.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh ip ro'" spine02
```

The **cs01** device should only know about the management network and 172.16.* networks.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh ip ro'" cs01
```

- ▶ 2. Add variables to support template-driven configuration.

If you do not already have a **vars** file named **vars/ebgp-breakup-data.yml**, download it now.

```
[student@workstation proj]$ cd vars
[student@workstation vars]$ wget \
> http://materials.example.com/full/vars/ebgp-breakup-data.yml
[student@workstation vars]$ cd ..
[student@workstation proj]$
```

- ▶ 3. Create NOS-specific device configuration templates with parameterized configuration statements.

- 3.1. Create a Jinja2 template with EBGP configuration statements for VyOS devices. Create a file named **j2/vyos-bgp.j2** with the following content. Note the first and third lines of this file are very long; do not include any newlines.

```

set protocols bgp {{ bgp_data[inventory_hostname]['as'] }} parameters router-id
{{ bgp_data[inventory_hostname]['router-id'] | ipaddr("address") }}
{% for nei in bgp_data[inventory_hostname]['neighbors'] %}
set protocols bgp {{ bgp_data[inventory_hostname]['as'] }} neighbor {{ nei['ipv4'] }
| ipaddr('address') }} remote-as {{ nei['as'] }}
{% endfor %}
{% for net in bgp_data[inventory_hostname]['networks'] %}
set protocols bgp {{ bgp_data[inventory_hostname]['as'] }} network {{ net }}
{% endfor %}

```

- 3.2. Create a Jinja2 template with EBGP configuration statements for IOS devices. Create a file named **j2/ios-bgp.j2** with the following content.

```

router bgp {{ bgp_data[inventory_hostname]['as'] }}
{% for net in bgp_data[inventory_hostname]['networks'] %}
  network {{ net | ipaddr('network') }} mask {{ net | ipaddr('netmask') }}
{% endfor %}
{% for nei in bgp_data[inventory_hostname]['neighbors'] %}
  neighbor {{ nei['ipv4'] | ipaddr('address') }} remote-as {{ nei['as'] }}
{% endfor %}

```

- 4. Perform a multivendor play that configures network devices from the updated Jinja2 templates.

- 4.1. Make sure that the **border-routers** host group exists in the Ansible hosts inventory. Your **inventory** file should include the following:

```

[leafs]
leaf[01:02]

[spines]
spine[01:02]

[border-routers]
spine01
spine02
cs01

[access-layer]
leaf01
leaf02
cs01
...output omitted...

```

- 4.2. Compose an Ansible Playbook named **ebgp-breakup.yml** consisting of the following:

```

---
- name: configure eBGP on VyOS and IOS border routers
  hosts: border-routers
  vars:
    vyos_bgp_tpl: j2/vyos-bgp.j2

```

```

ios_bgp_tpl: j2/ios-bgp.j2
vars_files:
- vars/ebgp-breakup-data.yml

tasks:

- name: static routes on cs01 to support non-local BGP routes
  ios_config:
    lines:
      - ip route 10.10.0.0 255.255.0.0 GigabitEthernet2 172.16.2.1
      - ip route 192.168.0.0 255.255.0.0 GigabitEthernet3 172.16.5.1
  when: inventory_hostname == 'cs01'

- name: "map bgp data to ios device using {{ ios_bgp_tpl }}"
  ios_config:
    src: "{{ ios_bgp_tpl }}"
  when: ansible_network_os == 'ios'

- name: "map bgp data to vyos device using {{ vyos_bgp_tpl }}"
  vyos_config:
    src: "{{ vyos_bgp_tpl }}"
  when: ansible_network_os == 'vyos'

```

- 4.3. Perform the play found in your new playbook. Refer to <http://materials.example.com/full/j2> if you are having trouble with your templates.

```

[student@workstation proj]$ ansible-playbook ebgp-breakup.yml
[student@workstation proj]$ ansible-playbook ebgp-breakup.yml

PLAY [configure eBGP on VyOS and IOS border routers] ****
TASK [static routes on cs01 to support non-local BGP routes] ****
skipping: [spine01]
skipping: [spine02]
changed: [cs01]

TASK [map bgp data to ios device using j2/ios-bgp.j2] ****
skipping: [spine01]
skipping: [spine02]
changed: [cs01]

TASK [map bgp data to vyos device using j2/vyos-bgp.j2] ****
skipping: [cs01]
changed: [spine02]
changed: [spine01]

PLAY RECAP ****
cs01 : ok=2    changed=2    unreachable=0    failed=0
spine01 : ok=1    changed=1    unreachable=0    failed=0
spine02 : ok=1    changed=1    unreachable=0    failed=0

```

- 5. Verify that the end result is what was intended.

- 5.1. Perform ad hoc commands to verify neighbor status. The **cs01** device should see two neighbors.

```
[student@workstation proj]$ ansible -m ios_command \
> -a "commands='sh ip bgp sum'" cs01
```

- 5.2. The **spine01** and **spine02** devices should see one neighbor each.

```
[student@workstation proj]$ ansible -m vyos_command \
> -a "commands='sh ip bgp sum'" spines
```

- 5.3. Perform the play found in the **e2e.yml** playbook you created earlier.

```
[student@workstation proj]$ ansible-playbook e2e.yml
```

If all goes well, the play recap should show **failed=0** for each device.

```
PLAY RECAP ****
cs01 : ok=3    changed=0    unreachable=0    failed=0
leaf01 : ok=2    changed=0    unreachable=0    failed=0
leaf02 : ok=2    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

Upgrading the Network

Objectives

After completing this section, you should be able to use Ansible to upgrade the operating system of network devices in your environment.

Upgrading the Network OS on Devices

An important DevNetOps role involves vigilance and prompt action concerning security updates. This often entails upgrading the network OS on devices. How that happens depends on the platform. The process of upgrading a network OS is illustrated here using VyOS, because VyOS has a single, simple, upgrade methodology. An overview is given of the process of upgrading IOS devices.

Upgrading IOS on Cisco Devices

Be sure to thoroughly test and verify automation processes before involving production resources.

Upgrading IOS on Cisco devices typically involves these steps, each of which can be automated with Ansible:

- Gather information (facts).
- Obtain an updated image and copy it to device flash.
- Set the device to boot from the new image.
- Ensure that the running configuration is saved and backed up.
- Reload the device.
- Verify connectivity and functionality when the boot sequence finishes.

Upgrading VyOS on VyOS Devices

VyOS has a simpler upgrade procedure.

An upgrade method for VyOS:

- Store the URL of the new image in a variable.
- Get the old system image.
- Download a fresh image.
- Get a new system image.
- Reload only if the new image is different from the old image.
- Wait for restart.

A more sophisticated method avoids downloading the new image unless needed.

```
# sysimg_url var defined in group_vars
- name: a play that upgrades a VyOS device
  hosts: spine02
  gather_facts: no

  tasks:
    - name: get old system image information
```

```
vyos_command:  
  commands:  
    - show system image  
register: old_system_image  
  
- name: download fresh system image  
vyos_command:  
  commands:  
    - add system image {{ sysimg_url }}  
  
- name: get new system image information  
vyos_command:  
  commands:  
    - show system image  
register: new_system_image  
  
- name: reload only if changed  
vyos_command:  
  commands:  
    - command: reboot now  
ignore_errors: yes  
when: old_system_image.stdout != new_system_image.stdout  
  
- name: wait for restart  
wait_for_connection:  
  delay: 20  
  timeout: 120  
when: old_system_image.stdout != new_system_image.stdout
```

► Guided Exercise

Upgrading VyOS

Upgrading the Network Operating System on devices can be an important part of ongoing maintenance for the sake of avoiding or eliminating software defects, including security vulnerabilities.

In this exercise, you will compose and perform a play with the ability to automatically upgrade a VyOS network device.

Outcomes

You should be able to:

- Compose a play with the ability to upgrade a VyOS network device.
- Perform the play in check-only mode to determine what effect it would have in change mode.

Before You Begin

Open a terminal window on the **workstation** VM and change to the `~/proj/` directory.

- 1. Compose a play with the ability to upgrade a VyOS network device. Create a playbook named **vyos-upgrade.yml** with the following contents:

```
---
- name: a play that upgrades a VyOS device
  hosts: vyos
  vars:
    sysimg_url: https://downloads.vyos.io/release/1.1.8/vyos-1.1.8-amd64.iso

  tasks:
    - name: abort unless target host has ansible_network_os == 'vyos'
      assert:
        that: "ansible_network_os == 'vyos'"

    - name: get old system image information
      vyos_command:
        commands:
          - show system image
      register: old_system_image

    - name: show old system image information
      debug:
        msg: "{{ old_system_image.stdout }}"

    - name: download fresh system image
      vyos_command:
        commands:
          - add system image {{ sysimg_url }}
```

```
- name: get new system image information
  vyos_command:
    commands:
      - show system image
  register: new_system_image

- name: show new system image information
  debug:
    msg: "{{ new_system_image.stdout }}"

- name: reboot the system
  vyos_command:
    commands:
      - command: reboot now
  ignore_errors: yes
  when: old_system_image.stdout != new_system_image.stdout

- name: wait for restart
  wait_for_connection:
    delay: 20
    timeout: 120
  when: old_system_image.stdout != new_system_image.stdout
```

- ▶ 2. Perform the play in check-only mode to determine what effect it would have in change mode. Execute the **ansible-playbook** command using the **--check** or **-C** option. Limit it to **spine01**.

```
[student@workstation proj]$ ansible-playbook -C -l spine01 vyos-upgrade.yml

PLAY [a play that upgrades a VyOS device] ****

TASK [abort unless target host has ansible_network_os == 'vyos'] ****
ok: [spine01] => {
    "changed": false,
    "msg": "All assertions passed"
}

TASK [get old system image information] ****
ok: [spine01]

TASK [show old system image information] ****
ok: [spine01] => {
    "msg": [
        "The system currently has the following image(s) installed:\n\n"
        "  1: 1.1.8 (default boot)"
    ]
}

TASK [download fresh system image] ****
[WARNING]: only show commands are supported when using check mode, not
executing `add system image
https://downloads.vyos.io/release/1.1.8/vyos-1.1.8-amd64.iso`

ok: [spine01]
```

```
TASK [get new system image information] ****
ok: [spine01]

TASK [show new system image information] ****
ok: [spine01] => {
    "msg": [
        "The system currently has the following image(s) installed:\n\n"
        "  1: 1.1.8 (default boot)"
    ]
}

TASK [reboot the system] ****
skipping: [spine01]

TASK [wait for restart] ****
skipping: [spine01]

PLAY RECAP ****
spine01 : ok=6    changed=0    unreachable=0    failed=0
```

This concludes the guided exercise.

▶ Lab

Automating Complex Operations

Access Control Lists are extremely useful tools. They play an important role in permitting or denying traffic. They are also used to define interesting traffic for the purpose of policy routing, and much more.

In this lab, an Ansible play is used to enable logging of inbound SSH traffic on the management interface. In the real world, logging all inbound SSH traffic could be used to detect unauthorized access.

Outcomes

You should be able to:

- Compose a playbook containing a play that:
 - Creates an extended Access Control List (ACL) that logs TCP port 22 traffic from anywhere to anywhere.
 - Applies the ACL to the management interface.
- Perform the play that creates ACL 101 and applies it to the management interface.
- Verify that the ACL is working

Before You Begin

Open a terminal window on the **workstation** VM.

1. You already have a playbook named **j2cfg.yml** that applies changes to networking devices based on configuration statements generated by Jinja2 templates. Modify the **j2/ios-config.j2** template to create ACL 101 and apply it to the management interface.
Create an extended Access Control List (ACL) that logs TCP port 22 traffic from anywhere to anywhere. To create an ACL that logs SSH traffic, append a single configuration statement that defines extended ACL 101, permitting TCP traffic for port 22 from anywhere to anywhere, logging the matches. The format of an extended ACL configuration statement that logs matches is **access list number permit/deny protocol src dst eq port log**. The IOS configuration statement that defines an ACL 101 that logs all SSH traffic from anywhere to anywhere is **access list 101 permit tcp any any eq 22**.
Apply the ACL to the management interface. The management interface of **cs01** is **GigabitEthernet1**. To apply ACL 101 to the management interface, issue the interface level configuration statement **ip access-group 101 in**.
2. Perform the play that creates ACL 101 and applies it to the management interface. Limit it to **cs01**.
3. Verify that the ACL is working.

This concludes the lab.

► Solution

Automating Complex Operations

Access Control Lists are extremely useful tools. They play an important role in permitting or denying traffic. They are also used to define interesting traffic for the purpose of policy routing, and much more.

In this lab, an Ansible play is used to enable logging of inbound SSH traffic on the management interface. In the real world, logging all inbound SSH traffic could be used to detect unauthorized access.

Outcomes

You should be able to:

- Compose a playbook containing a play that:
 - Creates an extended Access Control List (ACL) that logs TCP port 22 traffic from anywhere to anywhere.
 - Applies the ACL to the management interface.
- Perform the play that creates ACL 101 and applies it to the management interface.
- Verify that the ACL is working

Before You Begin

Open a terminal window on the **workstation** VM.

1. You already have a playbook named **j2cfg.yml** that applies changes to networking devices based on configuration statements generated by Jinja2 templates. Modify the **j2/ios-config.j2** template to create ACL 101 and apply it to the management interface.

Create an extended Access Control List (ACL) that logs TCP port 22 traffic from anywhere to anywhere. To create an ACL that logs SSH traffic, append a single configuration statement that defines extended ACL 101, permitting TCP traffic for port 22 from anywhere to anywhere, logging the matches. The format of an extended ACL configuration statement that logs matches is **access list number permit/deny protocol src dst eq port log**. The IOS configuration statement that defines an ACL 101 that logs all SSH traffic from anywhere to anywhere is **access list 101 permit tcp any any eq 22**.

Apply the ACL to the management interface. The management interface of **cs01** is **GigabitEthernet1**. To apply ACL 101 to the management interface, issue the interface level configuration statement **ip access-group 101 in**.

The resulting **j2/ios-config.j2** file should have the following content:

```
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
{% for nameserver in nameservers %}
ip name-server {{ nameserver }}
{% endfor %}
service timestamps log datetime
```

```
service timestamps debug datetime
logging {{ syslog_ipv4 }}
logging trap {{ ios_loglevel }}
access-list 1 permit {{ workstation_ipv4 | ipaddr('address') }} log
access-list 1 permit {{ tower_ipv4 | ipaddr('address') }} log
snmp-server community {{ ro_community }} RO 1
access-list 101 permit tcp any any eq 22 log
interface GigabitEthernet1
  ip access-group 101 in
```

2. Perform the play that creates ACL 101 and applies it to the management interface. Limit it to **cs01**.

```
[student@workstation proj]$ ansible-playbook -l cs01 j2cfg.yml
```

3. Verify that the ACL is working.

- 3.1. Monitor traffic on the **syslog** port of the **workstation** VM to confirm that log messages are arriving. Run **tcpdump**, listening on **eth0** port 514.

```
[student@workstation proj]$ sudo tcpdump -Xni eth0 port 514
[sudo] password for student: student
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

- 3.2. Open another terminal window on the **workstation** machine. Use SSH to connect to **cs01** as **admin** using **student** as the password.

```
[student@workstation ~]$ ssh admin@cs01
Password: student
cs01#
```

- 3.3. In your **tcpdump** terminal window, you should see a hex dump of the log message caused by the SSH connection to **cs01**.

```
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:15:16.095546 IP 172.25.250.195.50767 > 172.25.250.254.syslog: SYSLOG
  local7.info, length: 147
    0x0000:  4500 00af 0005 0000 ff11 6d43 ac19 fac3  E.....mC....
    0x0010:  ac19 fafe c64f 0202 009b e6a6 3c31 3930  .....0.....<190
    0x0020:  3e38 323a 202a 4175 6720 2035 2032 303a  >82..*Aug..5.20:
    0x0030:  3135 3a31 343a 2025 464d 414e 4650 2d36  15:14:.%FMANFP-6
    0x0040:  2d49 5041 4343 4553 534c 4f47 503a 2046  -IPACCESSLOGP:.F
    0x0050:  303a 2066 6d61 6e5f 6670 5f69 6d61 6765  0:.fman_fp_image
    0x0060:  3a20 206c 6973 7420 3130 3120 7065 726d  :..list.101.perm
    0x0070:  6974 7465 6420 7463 7020 3137 322e 3235  itted.tcp.172.25
    0x0080:  2e32 3530 2e32 3534 2835 3238 3634 2920  .250.254(52864).
    0x0090:  2d3e 2031 3732 2e32 352e 3235 302e 3139  ->.172.25.250.19
    0x00a0:  3528 3232 292c 2031 2070 6163 6b65 74      5(22),.1.packet
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
```

Use **Control+C** to break out of the **tcpdump** session.

This concludes the lab.

Chapter 7

Comprehensive Review

Goal

Review tasks from *Red Hat Ansible for Network Automation*

Objectives

- Review tasks from *Red Hat Ansible for Network Automation*

Sections

- Comprehensive Review

Lab

- Deploying Ansible
- Executing Commands and Plays
- Parameterizing Automation
- Administering Ansible
- Automating Simple Network Operations
- Automating Complex Network Operations

Comprehensive Review

Objectives

After completing this section, you should have reviewed and refreshed the knowledge and skills learned in Red Hat Ansible for Network Automation.

Reviewing Git

Labs in the Comprehensive Review make use of the Git Source Code Management system.

The following is a very brief overview of some the most basic Git subcommands:

- Obtain a local copy of a repository.

```
$ git clone username@host:/path/to/repository
```

- Add all new files, recursively, to the index of files in the local working copy.

```
$ git add -A :/
```

- Commit changes to your local working copy.

```
$ git commit -m "Commit message"
```

- Send changes to the remote repository.

```
$ git push
```

Ignoring Files

You can instruct Git not to track and remotely store particular files.

To ignore all **vault.yml** files, for instance, create a file named **.gitignore** in the project root with the following content:

```
**/vault.yml
```

This ignores files by this name located in subdirectories, too.

It does not affect files that have already been added, committed, and pushed to a repository.

Reviewing Red Hat Ansible for Network Automation

Before beginning the comprehensive review for this course, you should be comfortable with the topics covered in each chapter.

You can refer to earlier sections in the textbook for extra study.

Chapter 1, Deploying Ansible

Install and configure Ansible.

- Install Red Hat Ansible Engine on a Red Hat Enterprise Linux control node.
- Set up and validate the hosts inventory used in the labs.

Chapter 2, Running Commands and Plays

Run automated tasks on devices using plays and ad hoc commands.

- Run ad hoc commands to execute single, one-time tasks.
- Write playbooks, run plays with **ansible-playbook**, and interpret the resulting output.
- Build more complex plays that include multiple tasks.
- Write playbooks that include multiple plays.

Chapter 3, Parameterizing Automation

Perform complex tasks with loops, variables, conditions, roles, and templates.

- Define variables and use them in conditionals to control tasks.
- Parameterize playbooks and deploy customized files using Jinja2 templates.

Chapter 4, Administering Ansible

Discuss how Ansible solves administrative challenges faced by enterprises today.

- Manage advanced inventories, safeguard information with Ansible Vault.
- Define roles and manage infrastructure using Red Hat Ansible Tower.

Chapter 5, Automating Simple Network Operations

Automate simple operations on network devices

- Interrogate network devices with automation.
- Back up device configurations.
- Perform simple actions using playbooks.
- Apply simple changes using playbooks.

Chapter 6, Automating Complex Network Operations

Automate complex operations on network devices

- Apply changes to a network that involve many tasks using playbooks.
- Resolve network issues using playbooks.

▶ Lab

Deploying Ansible

In this lab, you will deploy a local Ansible hosts inventory, custom configuration, and supporting variables, to fully and safely automate the process of performing tasks on remote devices in the Lab Network.

Outcomes

You should be able to:

- Clone the **ansible-generic-project** Git repository to create a local directory for your new Ansible project.
- Create a hosts inventory file.
- Create a directory structure to hold group variables.
- Create plain and encrypted group variables files.
- Create a Vault password directory and Vault password file outside of your project directory and protect these with appropriate file-system permissions.
- Create a local **ansible.cfg** file to provide customization.
- Verify connectivity to managed network devices, omitting the servers.
- Commit your work to the Git repository.

Before You Begin

Open a terminal window on the **workstation** VM.

Instructions

Perform the following steps:

- Clone the **ansible-generic-project** Git repository to create a local directory for your new Ansible project. Use the following URL in the Lab Environment:
 - `http://git.lab.example.com:3000/student/ansible-generic-project.git`
- Create a hosts inventory file.
 - Base the hosts inventory file on the contents of *Appendix A, Table of Lab Network Hosts and Groups*
- Create a directory structure to hold group variables.
- Create plain and encrypted group files.
 - Map groups to connection and authentication variables as shown in *Appendix B, Connection and Authentication Variables*.
 - Create encrypted **vault.yml** variable files to hold sensitive data.

- Create a Vault password directory and Vault password file outside of your project directory and protect these with appropriate file-system permissions.
 - Create a local **ansible.cfg** file to provide local customization. Use **gathering = explicit**, for instance, to avoid typing **gather_facts: False** at the top of every play.
 - Verify connectivity to managed network devices, omitting the servers.
 - The **verify-access.yml** playbook is provided with **ansible-generic-project**. Use that playbook to verify access or create your own. If failing the verification, troubleshoot and resolve the problem.
- The **show-current-access-vars.yml** playbook is also provided with **ansible-generic-project**. This might be useful if connection and authentication tests are failing.
- Commit your work to the Git repository. If prompted for credentials, user and password are **student** and **student**. The commands are shown here.

```
[student@workstation ]$ git add -A :/  
[student@workstation ]$ git commit -m "ch7 lab1"  
[student@workstation ]$ git push
```

► Solution

Deploying Ansible

In this lab, you will deploy a local Ansible hosts inventory, custom configuration, and supporting variables, to fully and safely automate the process of performing tasks on remote devices in the Lab Network.

Outcomes

You should be able to:

- Clone the **ansible-generic-project** Git repository to create a local directory for your new Ansible project.
- Create a hosts inventory file.
- Create a directory structure to hold group variables.
- Create plain and encrypted group variables files.
- Create a Vault password directory and Vault password file outside of your project directory and protect these with appropriate file-system permissions.
- Create a local **ansible.cfg** file to provide customization.
- Verify connectivity to managed network devices, omitting the servers.
- Commit your work to the Git repository.

Before You Begin

Open a terminal window on the **workstation** VM.

Instructions

Perform the following steps:

- Clone the **ansible-generic-project** Git repository to create a local directory for your new Ansible project. Use the following URL in the Lab Environment:
 - `http://git.lab.example.com:3000/student/ansible-generic-project.git`
- Create a hosts inventory file.
 - Base the hosts inventory file on the contents of *Appendix A, Table of Lab Network Hosts and Groups*
- Create a directory structure to hold group variables.
- Create plain and encrypted group files.
 - Map groups to connection and authentication variables as shown in *Appendix B, Connection and Authentication Variables*.
 - Create encrypted **vault.yml** variable files to hold sensitive data.

- Create a Vault password directory and Vault password file outside of your project directory and protect these with appropriate file-system permissions.
- Create a local **ansible.cfg** file to provide local customization. Use **gathering = explicit**, for instance, to avoid typing **gather_facts: False** at the top of every play.
- Verify connectivity to managed network devices, omitting the servers.
 - The **verify-access.yml** playbook is provided with **ansible-generic-project**. Use that playbook to verify access or create your own. If failing the verification, troubleshoot and resolve the problem.

The **show-current-access-vars.yml** playbook is also provided with **ansible-generic-project**. This might be useful if connection and authentication tests are failing.

- Commit your work to the Git repository. If prompted for credentials, user and password are **student** and **student**. The commands are shown here.

```
[student@workstation ]$ git add -A :/  
[student@workstation ]$ git commit -m "ch7 lab1"  
[student@workstation ]$ git push
```

1. Clone the **ansible-generic-project** Git repository to create a local directory for your new Ansible project. Use the following URL in the Lab Environment: <http://git.lab.example.com:3000/student/ansible-generic-project.git>

```
[student@workstation ~]$ git clone \  
> http://git.lab.example.com:3000/student/ansible-generic-project.git
```

Change into the directory created by the **git clone** command.

```
[student@workstation ~]$ cd ansible-generic-project  
[student@workstation ansible-generic-project]$
```

2. Download the **example.com** hosts inventory file.

```
[student@workstation ansible-generic-project]$ wget \  
> http://materials.example.com/full/inventory
```

3. Create a directory structure to hold group variables, and create plain and encrypted group variables files.

- 3.1. Create **group_vars/** subdirectories for groups with variables listed in *Appendix B*.

```
[student@workstation ansible-generic-project]$ mkdir -p group_vars/ios  
[student@workstation ansible-generic-project]$ mkdir group_vars/local  
[student@workstation ansible-generic-project]$ mkdir group_vars/network  
[student@workstation ansible-generic-project]$ mkdir group_vars/vyos
```

- 3.2. Map groups to connection and authentication variables.

```
[student@workstation ansible-generic-project]$ cat group_vars/ios/vars.yml
ansible_network_os: ios

[student@workstation ansible-generic-project]$ cat group_vars/local/vars.yml
ansible_network_os: local

[student@workstation ansible-generic-project]$ cat group_vars/network/vars.yml
ansible_connection: network_cli

[student@workstation ansible-generic-project]$ cat group_vars/vyos/vars.yml
ansible_network_os: vyos
```

- 3.3. Create encrypted **vault.yml** variables files to hold sensitive data. Use **redhat** as the Vault password.

The **group_vars/ios/vault.yml** file contains "**ansible_user: admin**" and "**ansible_password: student**".

```
[student@workstation ansible-generic-project]$ ansible-vault create \
> group_vars/ios/vault.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

The **group_vars/vyos/vault.yml** file contains "**ansible_user: vyos**" and "**ansible_password: vyos**".

```
[student@workstation ansible-generic-project]$ ansible-vault create \
> group_vars/vyos/vault.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

4. Create a Vault password directory and Vault password file outside of your project directory and protect these with appropriate file-system permissions.



Note

You can skip this step if you already completed it in Guided Exercise 4.3.

```
[student@workstation ansible-generic-project]$ cd ..
[student@workstation ~]$ mkdir .rvh
[student@workstation ~]$ chmod 700 .rvh
[student@workstation ~]$ ls -ld .rvh
drwx----- 2 student student 25 May 29 15:42 .rvh
[student@workstation ~]$ echo redhat > .rvh/vault-secret
[student@workstation ~]$ chmod 600 .rvh/vault-secret
[student@workstation ~]$ ls -l .rvh/vault-secret
-rw----- 1 student student 7 May 29 15:43 .rvh/vault-secret
[student@workstation ~]$ cd ansible-generic-project
```

5. Create a local **ansible.cfg** file to provide customization.

```
[student@workstation ansible-generic-project]$ cat ansible.cfg
[defaults]
inventory = inventory
host_key_checking = False
gathering = explicit
vault_password_file = ../../rhv/vault-secret

[persistent_connection]
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

6. Verify connectivity to managed network devices, omitting the servers.

The **verify-access.yml** playbook is provided with the **ansible-generic-project** Git repo. Use that playbook to verify access or create your own.

```
[student@workstation ansible-generic-project]$ ansible-playbook verify-access.yml
```

If failing the verification, troubleshoot and resolve the problem.

The **show-current-access-vars.yml** playbook is provided with the **ansible-generic-project** Git repo. This might be useful if the connection and authentication tests are failing. Use **-l SUBSET** to limit the target set to a particular host or group. In this example, for instance, access variables are viewed for **spine01**.

```
[student@workstation ansible-generic-project]$ ansible-playbook -l spine01 \
> show-current-access-vars.yml

PLAY [a play that exposes the current access vars] ****

TASK [show the value of key variables] ****
ok: [spine01] => {
    "msg": "host: spine01, con: ssh, nos: vyos, user: vyos, pass: vyos\n"
}

PLAY RECAP ****
spine01 : ok=1    changed=0    unreachable=0    failed=0
```

7. Commit your work to the Git repository.

- 7.1. Add a **.gitignore** file that instructs Git not to store **vault.yml** files.

```
[student@workstation ansible-generic-project]$ cat .gitignore
**/vault.yml
```

- 7.2. Update the repository. If prompted the Git credentials are **student**, password **student**.

```
[student@workstation ansible-generic-project]$ git add -A :/
[student@workstation ansible-generic-project]$ git commit -m "ch7 lab 1"
[student@workstation ansible-generic-project]$ git push
```

▶ Lab

Executing Commands and Plays

In this lab, you will review the process of composing and executing ad hoc commands and plays.

Outcomes

You should be able to:

- Compose and execute an ad hoc command that displays the results you would see if you typed commands interactively on the console of a network device.
 - For VyOS devices, display the results of the **show interfaces** command.
 - For IOS devices, display the results of the **show ip interfaces brief** command.
- Create a playbook that does the same thing as the ad hoc commands you just executed.
- Perform the play or plays in the playbook you created.
- Compose a playbook named **ios-vyos-assert-domain.yml** that asserts that the domain name is currently set to "lab.example.com."
- Perform the play you created that asserts the value of the domain name.

Before You Begin

Open a terminal window on the **workstation** VM, and change into a directory that provides files that support Ansible connectivity to network devices in the Lab Network (that is, the ability to perform tasks on remote devices there).

Instructions

Perform the following steps:

- Choose a module that provides the ability to run commands remotely on network devices.
- Figure out the correct arguments and syntax to use in order to execute the CLI commands as Ansible ad hoc commands using the appropriate module.
- Execute the ad hoc commands.
- Compose a playbook that accomplishes the same thing.
- Perform the play or plays in the playbook you created to display interface information.
- Compose a playbook named **ios-vyos-assert-domain.yml** that asserts that the domain name is currently set to "lab.example.com." At the command line of a VyOS device, you could type "**sh conf comm | grep domain-name**" to show how the device is currently configured with respect to domain name. At the command line of an IOS device, you could type "**sh run | include domain**".
- Perform the play you created that asserts the value of the domain name.

► Solution

Executing Commands and Plays

In this lab, you will review the process of composing and executing ad hoc commands and plays.

Outcomes

You should be able to:

- Compose and execute an ad hoc command that displays the results you would see if you typed commands interactively on the console of a network device.
 - For VyOS devices, display the results of the **show interfaces** command.
 - For IOS devices, display the results of the **show ip interfaces brief** command.
- Create a playbook that does the same thing as the ad hoc commands you just executed.
- Perform the play or plays in the playbook you created.
- Compose a playbook named **ios-vyos-assert-domain.yml** that asserts that the domain name is currently set to "lab.example.com."
- Perform the play you created that asserts the value of the domain name.

Before You Begin

Open a terminal window on the **workstation** VM, and change into a directory that provides files that support Ansible connectivity to network devices in the Lab Network (that is, the ability to perform tasks on remote devices there).

Instructions

Perform the following steps:

- Choose a module that provides the ability to run commands remotely on network devices.
- Figure out the correct arguments and syntax to use in order to execute the CLI commands as Ansible ad hoc commands using the appropriate module.
- Execute the ad hoc commands.
- Compose a playbook that accomplishes the same thing.
- Perform the play or plays in the playbook you created to display interface information.
- Compose a playbook named **ios-vyos-assert-domain.yml** that asserts that the domain name is currently set to "lab.example.com." At the command line of a VyOS device, you could type "**sh conf comm | grep domain-name**" to show how the device is currently configured with respect to domain name. At the command line of an IOS device, you could type "**sh run | include domain**".
- Perform the play you created that asserts the value of the domain name.

1. Choose a module that provides the ability to run commands remotely on network devices.
 - 1.1. On VyOS devices, the module to use is **vyos_command**.
 - 1.2. On IOS devices, the module to use is **ios_command**.
2. Figure out the correct arguments and syntax to use in order to execute the CLI commands as Ansible ad hoc commands using the appropriate module.
 - 2.1. For VyOS devices this is the command:
ansible -m vyos_command -a"commands='sh int'" vyos
 - 2.2. For IOS devices this is the command:
ansible -m ios_command -a"commands='sh ip int br'" ios

3. Execute the ad hoc commands

- 3.1. For VyOS devices:

```
[student@workstation ansible-generic-project]$ ansible -m vyos_command \
> -a "commands='sh int'" vyos
```

- 3.2. For IOS devices:

```
[student@workstation ansible-generic-project]$ ansible -m ios_command \
> -a "commands='sh ip int br'" ios
```

4. Compose a playbook that accomplishes the same thing.

```
[student@workstation ansible-generic-project]$ cat ios-vyos-sh-br-int.yml
---
- name: multi-vendor play that shows interfaces
  hosts: network
  vars:
    ios_command: sh ip int br
    vyos_command: sh int

  tasks:
    - name: "{{ ios_command }} on IOS device {{ inventory_hostname }}"
      ios_command:
        commands:
        - "{{ ios_command }}"
      register: ios_result
      when: ansible_network_os == 'ios'

    - name: show IOS result
      debug:
        var: ios_result
      when: ansible_network_os == 'ios'

    - name: "{{ vyos_command }} on IOS device {{ inventory_hostname }}"
      vyos_command:
        commands:
```

```
- "{{ vyos_command }}"
register: vyos_result
when: ansible_network_os == 'vyos'

- name: show VyOS result
  debug:
    var: vyos_result
  when: ansible_network_os == 'vyos'
```

5. Perform the play or plays in the playbook you created.

```
[student@workstation ansible-generic-project]$ ansible-playbook \
> ios-vyos-sh-br-int.yml
```

▶ Lab

Parameterizing Automation

In this lab, you will set host and domain names and nameservers on managed devices in the Lab Network. In the course of doing so, you will review the process of using variables, loops and conditionals, roles, and Jinja2 templates to affect the outcome of automated processes.

Outcomes

You should be able to:

- Add variables that provide data supporting our objectives.
- Create a Jinja2 template that transforms the data into commands.
- Compose a playbook that sources commands from the Jinja2 template you created.
- Perform the play, restricting its scope to a single target.
- Commit your work to the Git repository.

Before You Begin

Open a terminal window on the **workstation** VM. Change to a directory that contains your latest updates to your **ansible-generic** project. This directory, together with an appropriately protected Vault password file, supports the ability to perform tasks on remote devices in the Lab Network.

Instructions

Perform the following steps:

- Add group variables for domain name and name servers. The host name is automatically available by virtue of the **inventory_hostname** variable.
- Create two Jinja2 templates that convert data into commands: one for VyOS devices and one for IOS devices. Name them **vyos-hn-dn-ns.j2** and **ios-hn-dn-ns.j2**, respectively.
- Create two playbooks named **j2test-vyos-hn-dn-ns.yml** and **j2test-ios-hn-dn-ns.yml**. The connection method should be set to local in these playbooks. Each of the playbooks should contain a play consisting of a single task, which uses the template module to simply write to a file or series of files (one per **inventory_hostname**) the document produced by the corresponding template. Use these playbooks to verify that the templates do indeed generate the desired configuration statements.
- Create a playbook named **set-hn-dn-ns.yml** that sets host name, domain name, and nameservers for networking devices by sourcing configuration statements directly from an appropriate Jinja2 template. The template should be selected according to the value of the **ansible_network_os** variable.
- Perform the play in the **set-hn-dn-ns.yml** playbook, restricting it to a single target.
- Commit your work to the Git repository. If prompted for credentials, user and password are **student** and **student**. The commands are shown here.

```
[student@workstation ansible-generic-project]$ git add -A :/  
[student@workstation ansible-generic-project]$ git commit -m "ch7 lab3"  
[student@workstation ansible-generic-project]$ git push
```

► Solution

Parameterizing Automation

In this lab, you will set host and domain names and nameservers on managed devices in the Lab Network. In the course of doing so, you will review the process of using variables, loops and conditionals, roles, and Jinja2 templates to affect the outcome of automated processes.

Outcomes

You should be able to:

- Add variables that provide data supporting our objectives.
- Create a Jinja2 template that transforms the data into commands.
- Compose a playbook that sources commands from the Jinja2 template you created.
- Perform the play, restricting its scope to a single target.
- Commit your work to the Git repository.

Before You Begin

Open a terminal window on the **workstation** VM. Change to a directory that contains your latest updates to your **ansible-generic** project. This directory, together with an appropriately protected Vault password file, supports the ability to perform tasks on remote devices in the Lab Network.

Instructions

Perform the following steps:

- Add group variables for domain name and name servers. The host name is automatically available by virtue of the **inventory_hostname** variable.
- Create two Jinja2 templates that convert data into commands: one for VyOS devices and one for IOS devices. Name them **vyos-hn-dn-ns.j2** and **ios-hn-dn-ns.j2**, respectively.
- Create two playbooks named **j2test-vyos-hn-dn-ns.yml** and **j2test-ios-hn-dn-ns.yml**. The connection method should be set to local in these playbooks. Each of the playbooks should contain a play consisting of a single task, which uses the template module to simply write to a file or series of files (one per **inventory_hostname**) the document produced by the corresponding template. Use these playbooks to verify that the templates do indeed generate the desired configuration statements.
- Create a playbook named **set-hn-dn-ns.yml** that sets host name, domain name, and nameservers for networking devices by sourcing configuration statements directly from an appropriate Jinja2 template. The template should be selected according to the value of the **ansible_network_os** variable.
- Perform the play in the **set-hn-dn-ns.yml** playbook, restricting it to a single target.
- Commit your work to the Git repository. If prompted for credentials, user and password are **student** and **student**. The commands are shown here.

```
[student@workstation ansible-generic-project]$ git add -A :/
[student@workstation ansible-generic-project]$ git commit -m "ch7 lab3"
[student@workstation ansible-generic-project]$ git push
```

- Add group variables for domain name and name servers. The host name is automatically available by virtue of the `inventory_hostname` variable.

Add a `domain_name` variable and a `name_servers` variable to the `network` group.

```
[student@workstation ansible-generic-project]$ cat group_vars/network/vars.yml
ansible_connection: network_cli
domain_name: lab.example.com
name_servers:
- 8.8.8.8
- 8.8.4.4
```

- Create two Jinja2 templates that convert data into commands: one for VyOS devices and one for IOS devices. Name them `vyos-hn-dn-ns.j2` and `ios-hn-dn-ns.j2`, respectively.

- If it does not already exist, create a `j2` directory to hold Jinja2 template files.

```
[student@workstation ansible-generic-project]$ mkdir j2
```

- Create a Jinja2 template named `j2/vyos-hn-dn-ns.j2` that generates host name, domain name and nameserver configuration statements for VyOS systems.

```
[student@workstation ansible-generic-project]$ cat j2/vyos-hn-dn-ns.j2
set system host-name {{ inventory_hostname }}
set system domain-name {{ domain_name }}
{% for ns in name_servers %}
set system name-server {{ ns }}
{% endfor %}
```

- Create a Jinja2 template named `j2/ios-hn-dn-ns.j2` that generates domain name and nameserver configuration statements for IOS systems.

```
[student@workstation ansible-generic-project]$ cat j2/ios-hn-dn-ns.j2
hostname {{ inventory_hostname }}
ip domain name {{ domain_name }}
{% for ns in name_servers %}
ip name-server {{ ns }}
{% endfor %}
```

- Create two playbooks named `j2test-vyos-hn-dn-ns.yml` and `j2test-ios-hn-dn-ns.yml`. The connection method should be set to local in these playbooks. Each of the playbooks should contain a play consisting of a single task, which uses the template module to simply write to a file or series of files (one per `inventory_hostname`) the document produced by the corresponding template. Use these playbooks to verify that the templates do indeed generate the desired commands.

- 3.1. Create a directory to hold files your playbooks create.

```
[student@workstation ansible-generic-project]$ mkdir out
```

- 3.2. Create the **j2test-vyos-hn-dn-ns.yml** playbook that generates VyOS configuration statements for a representative VyOS device and writes them to a file.

```
[student@workstation ansible-generic-project]$ cat j2test-vyos-hn-dn-ns.yml
---
- name: generate VyOS hostname, domain name, and nameserver statements
  hosts: spine01
  connection: local
  vars:
    tmpl: j2/vyos-hn-dn-ns.j2
    outfile: out/vyos-hn-dn-ns-{{ inventory_hostname }}.cmd

  tasks:

    - name: read from template, write to output file
      template:
        src: "{{ tmpl }}"
        dest: "{{ outfile }}"
```

- 3.3. Perform the play in the **j2test-vyos-hn-dn-ns.yml** playbook.

```
[student@workstation ansible-generic-project]$ ansible-playbook \
> j2test-vyos-hn-dn-ns.yml
```

- 3.4. Confirm that it generates VyOS configuration statements as desired.

```
[student@workstation ansible-generic-project]$ cat out/vyos-hn-dn-ns-spine01.cmd
set system host-name spine01
set system domain-name lab.example.com
set system name-server 8.8.8.8
set system name-server 8.8.4.4
```

- 3.5. Create the **j2test-ios-hn-dn-ns.yml** playbook that generates IOS configuration statements for a representative IOS device and writes them to a file.

```
[student@workstation ansible-generic-project]$ cat j2test-ios-hn-dn-ns.yml
---
- name: generate ios hostname, domain name, and nameserver statements
  hosts: cs01
  connection: local
  vars:
    tmpl: j2/ios-hn-dn-ns.j2
    outfile: out/ios-hn-dn-ns-{{ inventory_hostname }}.cmd

  tasks:

    - name: read from template, write to output file
```

```
template:
  src: "{{ tmpl }}"
  dest: "{{ outfile }}"
```

- 3.6. Perform the play in the **j2test-ios-hn-dn-ns.yml** playbook.

```
[student@workstation ansible-generic-project]$ ansible-playbook \
> j2test-ios-hn-dn-ns.yml
```

- 3.7. Confirm that it generates IOS configuration statements as desired.

```
[student@workstation ansible-generic-project]$ cat out/ios-hn-dn-ns-cs01.cmd
set system host-name cs01
set system domain-name lab.example.com
set system name-server 8.8.8.8
set system name-server 8.8.4.4
```

4. Create a playbook named **set-hn-dn-ns.yml** that sets host name, domain name, and nameservers for networking devices by sourcing configuration statements directly from an appropriate Jinja2 template. The template should be selected according to the value of the **ansible_network_os** variable.

Create a multivendor playbook named **set-hn-dn-ns.yml** that uses the `when` conditional to perform or skip tasks based on the value of the **ansible_network_os** variable. The connection method for these should be **network_cli**, NOT **local**. The playbook consists of a play that sets host name, domain name, and nameservers for both VyOS and IOS devices:

```
[student@workstation ansible-generic-project]$ cat set-hn-dn-ns.yml
---
- name: configure hostname, domain name, and nameserver
  hosts: network
  vars:
    tmpl: j2/{{ ansible_network_os }}-hn-dn-ns.j2

  tasks:
    - name: read from template, configure IOS device
      ios_config:
        src: "{{ tmpl }}"
      when: ansible_network_os == 'ios'

    - name: read from template, configure VyOS device
      vyos_config:
        src: "{{ tmpl }}"
      when: ansible_network_os == 'vyos'
```

5. Perform the play in the **set-hn-dn-ns.yml** playbook, limiting it to a single target.

```
[student@workstation ansible-generic-project]$ ansible-playbook -l spine01 \
> set-hn-dn-ns.yml
```

6. Compose a playbook named **ios-vyos-assert-domain.yml** that asserts that the domain name is currently set to "lab.example.com".

```
[student@workstation ansible-generic-project]$ cat ios-vyos-assert-domain.yml
---
- name: multi-vendor play that asserts value of domain name
  hosts: network
  vars:
    correct_domain: lab.example.com
    ios_command: sh run | include domain
    vyos_command: sh conf comm | grep domain-name

  tasks:

    - name: "{{ ios_command }} on IOS device"
      ios_command:
        commands:
          - "{{ ios_command }}"
      register: ios_result
      when: ansible_network_os == 'ios'

    - name: "assert IOS device is in {{ correct_domain }}"
      assert:
        that: "'{{ correct_domain }}' in ios_result.stdout[0]"
      when: ansible_network_os == 'ios'

    - name: "{{ vyos_command }} on VyOS device"
      vyos_command:
        commands:
          - "{{ vyos_command }}"
      register: vyos_result
      when: ansible_network_os == 'vyos'

    - name: "assert VyOS device is in {{ correct_domain }}"
      assert:
        that: "'{{ correct_domain }}' in vyos_result.stdout[0]"
      when: ansible_network_os == 'vyos'
```

7. Perform the play you created that asserts the value of the domain name.

```
[student@workstation ansible-generic-project]$ ansible-playbook \
> ios-vyos-assert-domain.yml
```

8. Commit your work to the Git repository. If prompted for credentials, user and password are **student** and **student**.

```
[student@workstation ansible-generic-project]$ git add -A && git commit -m \
> "ch7 lab3" && git push
```

► Lab

Administering Automation

In this lab, you will employ Red Hat Ansible Tower to put into effect the play you pushed to **ansible-generic** Git repository.

Outcomes

You should be able to:

- Employ an existing inventory in Red Hat Ansible Tower (the **lab.example.com** inventory you created in Lab 4).
- Create Credentials in Red Hat Ansible Tower to support the project.
- Create a Project in Red Hat Ansible Tower
- Create a Job Template in Red Hat Ansible Tower.
- Execute the Job Template.

Before You Begin

Open a web browser window on the **workstation** VM. Log in to <http://tower.lab.example.com> as **admin** using **student** as the password.

Instructions

Perform the following steps:

- Temporarily remove the line **vault_password_file = ../../rhv/vault-secret** from your **ansible.cfg** file then commit your work to the Git repository.
- Create Credentials in Red Hat Ansible Tower to support the project.
 - Create a Credential named **vyos-access** that provides administrative access to VyOS devices as user **vyos** with password **vyos**. The Credential type is **Machine**.
 - Create a Credential named **student-scm** that provides read access to the **ansible-generic-project** Git repo as user **student** with password **student**. The Credential type is **Source Control**.
- Create a Project in Red Hat Ansible Tower named **vyos-hn-dn-ns**. The project should have SCM Type of **Git**, and be configured with SCM source URL of <http://git.lab.example.com:3000/student/ansible-generic-project.git>, using branch **master**. It should use the **student-scm** SCM Credential you create in order to connect to the Git repo.
- Create a Job Template in Red Hat Ansible Tower named **vyos-hn-dn-ns**. The **JOB TYPE** should be **Check**, the **INVENTORY** should be the **lab.example.com** inventory that was created in Lab 4. The **MACHINE CREDENTIAL** should be the **vyos-access** Credential created in step 1. The **PLAYBOOK** should be the **set-hn-dn-ns.yml** playbook that was created and pushed to the **ansible-generic-project** Git repo in Lab 7.3.
- Execute the Job Template to check your playbook.

- Add back the line `vault_password_file = ../../rhv/vault-secret` to your `ansible.cfg` file.



Important

You MUST have completed Lab 4 and created the `lab.example.com` inventory in order to complete this lab.

► Solution

Administering Automation

In this lab, you will employ Red Hat Ansible Tower to put into effect the play you pushed to **ansible-generic** Git repository.

Outcomes

You should be able to:

- Employ an existing inventory in Red Hat Ansible Tower (the **lab.example.com** inventory you created in Lab 4).
- Create Credentials in Red Hat Ansible Tower to support the project.
- Create a Project in Red Hat Ansible Tower
- Create a Job Template in Red Hat Ansible Tower.
- Execute the Job Template.

Before You Begin

Open a web browser window on the **workstation** VM. Log in to <http://tower.lab.example.com> as **admin** using **student** as the password.

Instructions

Perform the following steps:

- Temporarily remove the line **vault_password_file = ../../rhv/vault-secret** from your **ansible.cfg** file then commit your work to the Git repository.
- Create Credentials in Red Hat Ansible Tower to support the project.
 - Create a Credential named **vyos-access** that provides administrative access to VyOS devices as user **vyos** with password **vyos**. The Credential type is **Machine**.
 - Create a Credential named **student-scm** that provides read access to the **ansible-generic-project** Git repo as user **student** with password **student**. The Credential type is **Source Control**.
- Create a Project in Red Hat Ansible Tower named **vyos-hn-dn-ns**. The project should have SCM Type of **Git**, and be configured with SCM source URL of <http://git.lab.example.com:3000/student/ansible-generic-project.git>, using branch **master**. It should use the **student-scm** SCM Credential you create in order to connect to the Git repo.
- Create a Job Template in Red Hat Ansible Tower named **vyos-hn-dn-ns**. The **JOB TYPE** should be **Check**, the **INVENTORY** should be the **lab.example.com** inventory that was created in Lab 4. The **MACHINE CREDENTIAL** should be the **vyos-access** Credential created in step 1. The **PLAYBOOK** should be the **set-hn-dn-ns.yml** playbook that was created and pushed to the **ansible-generic-project** Git repo in Lab 7.3.
- Execute the Job Template to check your playbook.

- Add back the line `vault_password_file = ../../rhv/vault-secret` to your `ansible.cfg` file.

**Important**

You MUST have completed Lab 4 and created the `lab.example.com` inventory in order to complete this lab.

1. Temporarily remove the line `vault_password_file = ../../rhv/vault-secret` from your `ansible.cfg` file then commit your work to the Git repository.

```
[student@workstation ansible-generic-project]$ cat ansible.cfg
[defaults]
inventory = inventory
host_key_checking = False
gathering = explicit

[persistent_connection]
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

Update the repository. If prompted the Git credentials are `student`, password `student`.

```
[student@workstation ansible-generic-project]$ git add ansible.cfg
[student@workstation ansible-generic-project]$ git commit -m "ch7 lab 4"
[student@workstation ansible-generic-project]$ git push
```

2. Create a credential named **vyos-access** that provides administrative access to VyOS devices as user **vyos** with password **vyos**. The Credential type is **Machine**.
 - 2.1. Click the Settings icon (the gear icon at the upper right) to go to the **SETTINGS** screen.
 - 2.2. Click the Credentials card to go to the **SETTINGS / CREDENTIALS** screen.
 - 2.3. At the **SETTINGS / CREDENTIALS** screen, click **ADD** to add a credential.
 - 2.4. At the **CREATE CREDENTIAL** screen, set the **NAME** field to **vyos-access**. Set **TYPE** to **Machine**. In the **TYPE DETAILS** form, set **USERNAME** to **vyos** and **PASSWORD** to **vyos**.
 - 2.5. Click **SAVE** at the lower right to finalize the creation of the **vyos-access** Credential.
3. Create a credential named **student-scm** that provides read access to the **ansible-generic-project** Git repo as user **student** with password **student**. The Credential type is **Source Control**.
 - 3.1. Click the Settings icon (the gear icon at the upper right) to go to the **SETTINGS** screen.
 - 3.2. Click the Credentials card to go to the **SETTINGS / CREDENTIALS** screen.

- 3.3. At the **SETTINGS / CREDENTIALS** screen, click **ADD** to add a credential.
 - 3.4. At the **CREATE CREDENTIAL** screen, set the **NAME** field to **student-scm**. Set **TYPE** to **Source Control**. In the **TYPE DETAILS** form, set **USERNAME** to **student** and **PASSWORD** to **student**.
 - 3.5. Click **SAVE** at the lower right to finalize the creation of the **student-scm** Credential.
4. Create a Project in Red Hat Ansible Tower named **vyos-hn-dn-ns**. The project should have SCM Type of **Git**, and be configured with SCM source URL of `http://git.lab.example.com:3000/student/ansible-generic-project.git`, using branch **master**. It should use the **student-scm** SCM Credential you create in order to connect to the Git repo.
 - 4.1. At the landing page, click the **PROJECTS** link to go to the **PROJECTS** page.
 - 4.2. On the **PROJECTS** page, click **ADD** to add a project.
 - 4.3. Set the **NAME** field to **vyos-hn-dn-ns**. Click the magnifying glass icon to select **Default** in the **ORGANIZATION** field. Set **SCM TYPE** to **Git**. In the **SOURCE DETAILS** form, set **SCM URL** to `http://git.lab.example.com:3000/student/ansible-generic-project.git`, **SCM BRANCH** to **master**, and **SCM CREDENTIAL** to **student-scm**.
 - 4.4. Click **SAVE** to finalize the creation of the **vyos-hn-dn-ns** Project.
 5. Create a Job Template in Red Hat Ansible Tower named **vyos-hn-dn-ns**. The **JOB TYPE** should be **Check**, the **INVENTORY** should be the `lab.example.com` inventory that was created in Lab 4. The **MACHINE CREDENTIAL** should be the **vyos-access** Credential created in step 1. The **PLAYBOOK** should be the `set-hn-dn-ns.yml` playbook that was created and pushed to the **ansible-generic-project** Git repo in Lab 7.3.
 - 5.1. At the landing page, click the **TEMPLATES** link to go to the **TEMPLATE** page.
 - 5.2. On the **TEMPLATES** page, click **ADD** to add a template and select **Job Template** as the template type.
 - 5.3. Set the **NAME** field to **vyos-hn-dn-ns**. Set **JOB TYPE** to **Check**, **INVENTORY** to `lab.example.com`, and **PROJECT** to **vyos-hn-dn-ns**. For **PLAYBOOK**, select the `set-hn-dn-ns.yml` playbook that was created and pushed to the **ansible-generic-project** Git repo in Lab 7.3. Set **MACHINE CREDENTIAL** to **vyos-access**. Set Limit to **vyos**.
 - 5.4. Click **SAVE** to finalize the creation of the **vyos-hn-dn-ns** Job Template.
 6. Execute the Job Template to check your playbook.

```

12
13 TASK [read from template, write to output file] ****
14 ok: [leaf01]
15 ok: [leaf02]
16 ok: [spine01]
17 ok: [spine02]
18
19 PLAY RECAP ****
20 leaf01 : ok=2    changed=0    unreachable=0    failed=0
21 leaf02 : ok=2    changed=0    unreachable=0    failed=0
22 spine01 : ok=2    changed=0    unreachable=0    failed=0
23 spine02 : ok=2    changed=0    unreachable=0    failed=0
24

```

Copyright © 2017 Red Hat, Inc.

- Add back the line **vault_password_file = ../../rhv/vault-secret** to your **ansible.cfg** file.

```
[student@workstation ansible-generic-project]$ cat ansible.cfg
[defaults]
inventory = inventory
host_key_checking = False
gathering = explicit
vault_password_file = ../../rhv/vault-secret

[persistent_connection]
command_timeout = 180
connect_timeout = 100
connect_retry_timeout = 100
```

► Lab

Automating Simple Network Operations

This Lab provisions a new layer 3 subnet under the Adjustment scenario.

The Break Up scenario described the separation of **example.com** network into three independent networks. Routing among the networks was established by way of explicitly advertised routes under eBGP, in accordance with peering agreements between the businesses.

Under the Break Up scenario, traffic between **server01** in **AS10101** and **server02** in **AS19216** is routed through **AS17216** by way of lab environment links simulating connections across the internet.

The new subnet represents an alternate path across the internet between **AS10101** and **AS19216**. It runs between the **eth3** interface of **spine01** and the **eth3** interface of **spine02**.

The management team asks you to:

1. Ensure that OSPF is not advertising routes across the new link.
2. Apply appropriate descriptions to the interfaces.
3. Configure layer 3 on the interfaces.

This Lab focuses on interface descriptions and layer 3 addresses. A different Lab configures routing.

You will be accessing the network devices by way of management interfaces. Those interfaces is already configured with respect to layer 3 and up, and should not be changed.

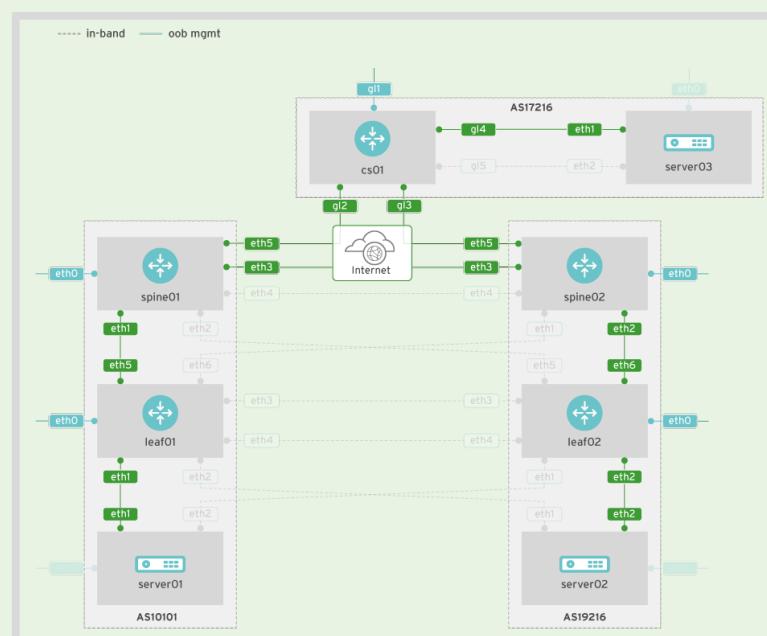


Figure 7.2: The Adjustment Phase Production Services Network of example.com

Adjustment Phase Interface Descriptions

Devices	Interfaces	Descriptions
spine01	eth3	spine02-as19216
spine02	eth3	spine01-as10101

Adjustment Phase Layer 3 Addressing (management interfaces not shown)

Devices	Interfaces	IPv4 Addresses
spine01	eth3	172.16.50.1/30
spine02	eth3	172.16.50.2/30

In this lab, you will assign IP addresses and implement interface description changes to activate a new link between autonomous systems. The changes in this lab correspond to the Adjustment phase in the growth and development of **example.com** and successor companies.

Outcomes

You should be able to:

- Create a variables file defining the variables that will be used.
- Perform a play that implements layer three address and interface description changes.
- Verify that the outcome is as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/ansible-generic-project/` directory.

Instructions

Perform the following steps:

- Create a variables file defining the variables that will be used.
 - Create a vars file named **vars/adjustment-data.yml** that defines two variables: **new_layer3_data** and **new_interface_data**. The **new_layer3_data** variable should be used in your playbook to configure interface layer 3 addresses. The **new_interface_data** variable should be used in your playbook to configure interface descriptions. See the **Adjustment Phase Interface Descriptions** and **Adjustment Phase Layer 3** Addressing tables for interface descriptions and layer 3 addresses.
- Perform a play that implements layer three address and interface description changes.
 - Compose a playbook named **adjustment.yml**. It should consist of a play that targets only **spine01** and **spine02**. The play should (1) make the new link interfaces passive with respect to OSPF, (2) apply interface descriptions to the new link interfaces, and (3) assign layer 3 addresses to the new link interfaces. The command that would be typed at the command line of a VyOS device to make an interface passive is "**set protocols ospf passive-interface interface-name**".

- Perform the play in **adjustment.yml** that implements the desired changes.
- Verify that the outcome is as desired.
 - Execute an ad hoc command to verify that interface descriptions and layer 3 addresses are correct for the new link interfaces on **spine01** and **spine02**.

► Solution

Automating Simple Network Operations

This Lab provisions a new layer 3 subnet under the Adjustment scenario.

The Break Up scenario described the separation of **example.com** network into three independent networks. Routing among the networks was established by way of explicitly advertised routes under eBGP, in accordance with peering agreements between the businesses.

Under the Break Up scenario, traffic between **server01** in **AS10101** and **server02** in **AS19216** is routed through **AS17216** by way of lab environment links simulating connections across the internet.

The new subnet represents an alternate path across the internet between **AS10101** and **AS19216**. It runs between the **eth3** interface of **spine01** and the **eth3** interface of **spine02**.

The management team asks you to:

1. Ensure that OSPF is not advertising routes across the new link.
2. Apply appropriate descriptions to the interfaces.
3. Configure layer 3 on the interfaces.

This Lab focuses on interface descriptions and layer 3 addresses. A different Lab configures routing.

You will be accessing the network devices by way of management interfaces. Those interfaces is already configured with respect to layer 3 and up, and should not be changed.

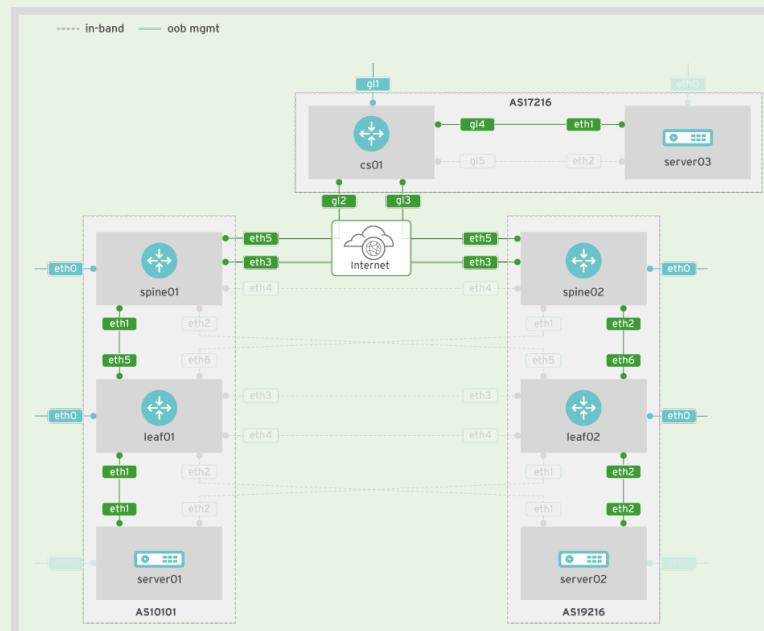


Figure 7.2: The Adjustment Phase Production Services Network of example.com

Adjustment Phase Interface Descriptions

Devices	Interfaces	Descriptions
spine01	eth3	spine02-as19216
spine02	eth3	spine01-as10101

Adjustment Phase Layer 3 Addressing (management interfaces not shown)

Devices	Interfaces	IPv4 Addresses
spine01	eth3	172.16.50.1/30
spine02	eth3	172.16.50.2/30

In this lab, you will assign IP addresses and implement interface description changes to activate a new link between autonomous systems. The changes in this lab correspond to the Adjustment phase in the growth and development of **example.com** and successor companies.

Outcomes

You should be able to:

- Create a variables file defining the variables that will be used.
- Perform a play that implements layer three address and interface description changes.
- Verify that the outcome is as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your **~/ansible-generic-project/** directory.

Instructions

Perform the following steps:

- Create a variables file defining the variables that will be used.
 - Create a vars file named **vars/adjustment-data.yml** that defines two variables: **new_layer3_data** and **new_interface_data**. The **new_layer3_data** variable should be used in your playbook to configure interface layer 3 addresses. The **new_interface_data** variable should be used in your playbook to configure interface descriptions. See the **Adjustment Phase Interface Descriptions** and **Adjustment Phase Layer 3** Addressing tables for interface descriptions and layer 3 addresses.
- Perform a play that implements layer three address and interface description changes.
 - Compose a playbook named **adjustment.yml**. It should consist of a play that targets only **spine01** and **spine02**. The play should (1) make the new link interfaces passive with respect to OSPF, (2) apply interface descriptions to the new link interfaces, and (3) assign layer 3 addresses to the new link interfaces. The command that would be typed at the command line of a VyOS device to make an interface passive is "**set protocols ospf passive-interface interface-name**".

- Perform the play in **adjustment.yml** that implements the desired changes.
- Verify that the outcome is as desired.
 - Execute an ad hoc command to verify that interface descriptions and layer 3 addresses are correct for the new link interfaces on **spine01** and **spine02**.

1. Create a variables file defining the variables that will be used.
 - 1.1. Create a vars file named **vars/adjustment-data.yml** that defines two variables: **new_layer3_data** and **new_interface_data**. The **new_layer3_data** variable should be used in your playbook to configure interface layer 3 addresses. The **new_interface_data** variable should be used in your playbook to configure interface descriptions. See the **Adjustment Phase Interface Descriptions** and **Adjustment Phase Layer 3 Addressing** tables for interface descriptions and layer 3 addresses.

Create the **vars/** directory if it does not already exist.

```
[student@workstation ansible-generic-project]$ mkdir -p vars
```

Create the **vars/adjustment-data.yml** file.

```
[student@workstation ansible-generic-project]$ cat > vars/adjustment-data.yml
new_layer3_data:
  spine01:
    - { name: eth3, ipv4: 172.16.50.1/30 }
  spine02:
    - { name: eth3, ipv4: 172.16.50.2/30 }
new_interface_data:
  spine01:
    eth3:
      description: spine02-as19216
  spine02:
    eth3:
      description: spine01-as10101
```

2. Perform a play that implements layer three address and interface description changes.
 - 2.1. Compose a playbook named **adjustment.yml**. It should consist of a play that targets only **spine01** and **spine02**. The play should (1) make the new link interfaces passive with respect to OSPF, (2) apply interface descriptions to the new link interfaces, and (3) assign layer 3 addresses to the new link interfaces. The command that would be typed at the command line of a VyOS device to make an interface passive is "**set protocols ospf passive-interface interface-name**".

```
[student@workstation ansible-generic-project]$ cat adjustment.yml
---
- name: adjust the network layer 3 model
  hosts: spine01, spine02
  vars_files:
```

```
- vars/adjustment-data.yml

tasks:

- name: make the new interfaces passive with respect to OSPF
  vyos_config:
    lines:
      - "set protocols ospf passive-interface 'eth3'"
  when: ansible_network_os == 'vyos'

- name: configure interface descriptions
  vyos_interface:
    name: "{{ item.key }}"
    description: "{{ item.value.description }}"
  with_dict: "{{ new_interface_data[inventory_hostname] }}"
  when: ansible_network_os == 'vyos'

- name: configure layer 3
  vyos_l3_interface:
    aggregate: "{{ new_layer3_data[inventory_hostname] }}"
  when: ansible_network_os == 'vyos'
```

2.2. Perform the play in **adjustment.yml** that implements the desired changes.

```
[student@workstation ansible-generic-project]$ ansible-playbook adjustment.yml
```

3. Verify that the outcome is as desired. Execute an ad hoc command to verify that interface descriptions and layer 3 addresses are correct for the new link interfaces on **spine01** and **spine02**.

```
[student@workstation ansible-generic-project]$ ansible -m vyos_command \
> -a "commands='sh int|grep eth3'" spines
spine01 | SUCCESS => {
  "changed": false,
  "stdout": [
    "eth3          172.16.50.1/30           u/u  spine02-as19216"
  ],
  "stdout_lines": [
    [
      "eth3          172.16.50.1/30           u/u  spine02-
as19216"
    ]
  ]
}
spine02 | SUCCESS => {
  "changed": false,
  "stdout": [
    "eth3          172.16.50.2/30           u/u  spine01-as10101"
  ],
  "stdout_lines": [
    [
      "eth3          172.16.50.2/30           u/u  spine01-
as10101"
    ]
  ]
}
```

]
]
}

► Lab

Automating Complex Network Operations

This Lab implements changes designed to route traffic across the newly provisioned link under the Adjustment scenario.

The Break Up scenario described the separation of **example.com** network into three independent networks. Routing among the networks was established by way of explicitly advertised routes under eBGP, in accordance with peering agreements between the businesses.

Under the Break Up scenario, traffic between **server01** in **AS10101** and **server02** in **AS19216** is routed through **AS17216** by way of lab environment links simulating connections across the internet.

This is an example of hub and spoke internetwork topology. It is preferred, though, for each of the three networks to consider both of the other two to be BGP neighbors and receive routes from them. When each AS advertises its subnets to both of its neighbors, traffic will not have to be routed across an intermediary (the hub). The new arrangement is an example of a mesh internetwork topology.

Under the first part of the Adjustment scenario, a new layer 3 subnet has been provisioned that provides an alternate path across the internet between AS10101 and AS19216. This Lab configures eBGP to advertise routes so traffic can flow between AS10101 and AS19216 across the alternate path, using the newly provisioned subnet.

The management team asks you to:

1. Configure eBGP to advertise routes appropriately.
2. Verify that after making the changes, spine01 and spine02 see two BGP neighbors, and have the desired routes.

You will be accessing the network devices by way of management interfaces. Those interfaces are already configured with respect to layer 3 and up, and should not be changed.

This exercise assumes that Guided Exercises 6.7 and 6.8 have been completed.

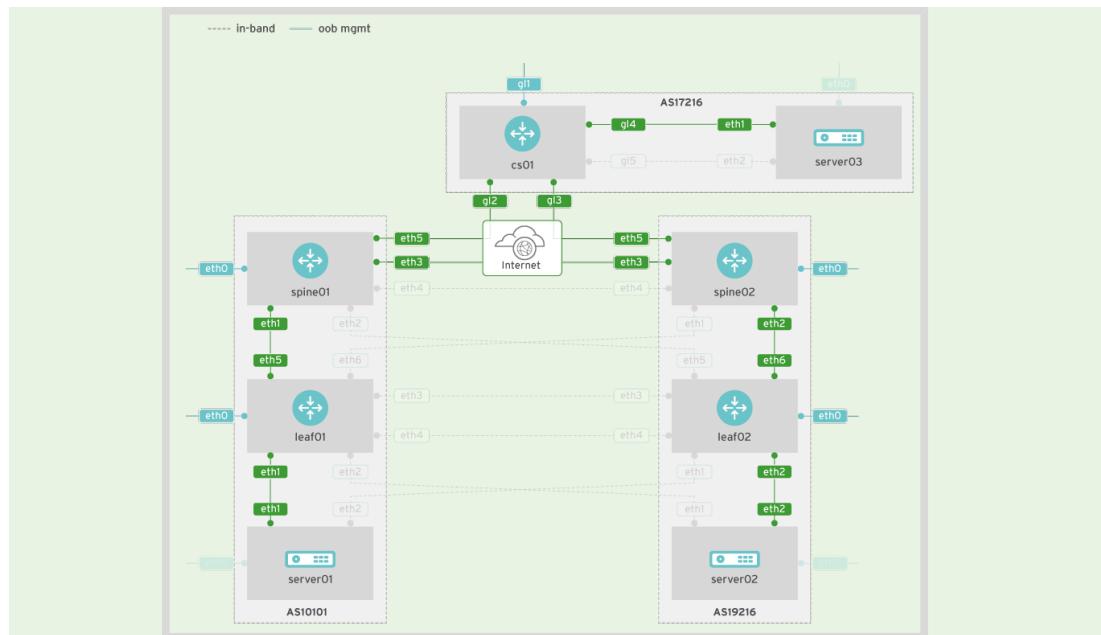


Figure 7.3: The Adjustment Phase Production Services Network of example.com

Adjustment Phase Interface Descriptions

Devices	Interfaces	Descriptions
spine01	eth3	spine02-as19216
spine02	eth3	spine01-as10101

Adjustment Phase Layer 3 Addressing (management interfaces not shown)

Devices	Interfaces	IPv4 Addresses
spine01	eth3	172.16.50.1/30
spine02	eth3	172.16.50.2/30

In this lab, you will create an eBGP neighbor relationship across the link that was added in Lab 7.5.

Outcomes

You should be able to:

- Revise the data in an existing variables file to support the new requirements
- Compose a playbook with a play that configures devices to support new eBGP neighbor relationships and a new set of network advertisements.
- Perform the play in the new playbook.
- Verify that the outcome is as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your **~/ansible-generic-project/** directory.

Instructions

Perform the following steps:

- Revise the data in an existing variables file to support the new requirements.
 - Create a vars file named **vars/ebgp-adjustment-data.yml**. It should be identical to **/home/student/proj/vars/ebgp-breakup-data.yml**, but each border router should now have two neighbors. The border routers are **spine01**, **spine02**, and **cs01**.
- Compose a playbook with a play that configures devices to support new eBGP neighbor relationships and a new set of network advertisements.

Name the new playbook **ebgp-adjustment.yml**. It does the same things as the **ebgp-breakup.yml**, but uses the **vars/ebgp-adjustment-data.yml** vars file instead of **vars/ebgp-breakup-data.yml**.

- Perform the play in the new playbook.
- Verify that the outcome is as desired.
 - Verify that each border router sees two BGP neighbors (**sh ip bgp sum**).
 - Verify that **spine01** has routes for **192.168.*** by way of **spine02** (172.16.50.2) rather than **cs01**.
 - Verify that **spine02** has routes for **10.10.*** by way of **spine01** (172.16.50.1) rather than **cs01**.

► Solution

Automating Complex Network Operations

This Lab implements changes designed to route traffic across the newly provisioned link under the Adjustment scenario.

The Break Up scenario described the separation of **example.com** network into three independent networks. Routing among the networks was established by way of explicitly advertised routes under eBGP, in accordance with peering agreements between the businesses.

Under the Break Up scenario, traffic between **server01** in **AS10101** and **server02** in **AS19216** is routed through **AS17216** by way of lab environment links simulating connections across the internet.

This is an example of hub and spoke internetwork topology. It is preferred, though, for each of the three networks to consider both of the other two to be BGP neighbors and receive routes from them. When each AS advertises its subnets to both of its neighbors, traffic will not have to be routed across an intermediary (the hub). The new arrangement is an example of a mesh internetwork topology.

Under the first part of the Adjustment scenario, a new layer 3 subnet has been provisioned that provides an alternate path across the internet between AS10101 and AS19216. This Lab configures eBGP to advertise routes so traffic can flow between AS10101 and AS19216 across the alternate path, using the newly provisioned subnet.

The management team asks you to:

1. Configure eBGP to advertise routes appropriately.
2. Verify that after making the changes, spine01 and spine02 see two BGP neighbors, and have the desired routes.

You will be accessing the network devices by way of management interfaces. Those interfaces are already configured with respect to layer 3 and up, and should not be changed.

This exercise assumes that Guided Exercises 6.7 and 6.8 have been completed.

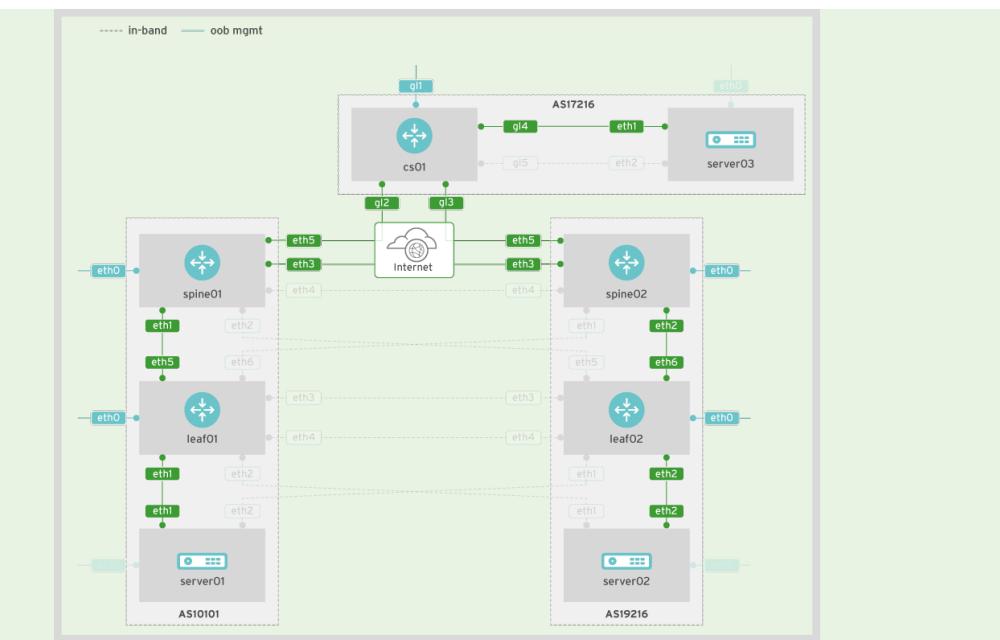


Figure 7.3: The Adjustment Phase Production Services Network of example.com

Adjustment Phase Interface Descriptions

Devices	Interfaces	Descriptions
spine01	eth3	spine02-as19216
spine02	eth3	spine01-as10101

Adjustment Phase Layer 3 Addressing (management interfaces not shown)

Devices	Interfaces	IPv4 Addresses
spine01	eth3	172.16.50.1/30
spine02	eth3	172.16.50.2/30

In this lab, you will create an eBGP neighbor relationship across the link that was added in Lab 7.5.

Outcomes

You should be able to:

- Revise the data in an existing variables file to support the new requirements
- Compose a playbook with a play that configures devices to support new eBGP neighbor relationships and a new set of network advertisements.
- Perform the play in the new playbook.
- Verify that the outcome is as desired.

Before You Begin

Open a terminal window on the **workstation** VM and change to your `~/ansible-generic-project/` directory.

Instructions

Perform the following steps:

- Revise the data in an existing variables file to support the new requirements.
 - Create a vars file named **vars/ebgp-adjustment-data.yml**. It should be identical to `/home/student/proj/vars/ebgp-breakup-data.yml`, but each border router should now have two neighbors. The border routers are **spine01**, **spine02**, and **cs01**.
- Compose a playbook with a play that configures devices to support new eBGP neighbor relationships and a new set of network advertisements.

Name the new playbook **ebgp-adjustment.yml**. It does the same things as the **ebgp-breakup.yml**, but uses the **vars/ebgp-adjustment-data.yml** vars file instead of **vars/ebgp-breakup-data.yml**.

- Perform the play in the new playbook.
- Verify that the outcome is as desired.
 - Verify that each border router sees two BGP neighbors (`sh ip bgp sum`).
 - Verify that **spine01** has routes for **192.168.*** by way of **spine02** (172.16.50.2) rather than **cs01**.
 - Verify that **spine02** has routes for **10.10.*** by way of **spine01** (172.16.50.1) rather than **cs01**.

1. Revise the data in an existing variables file to support the new requirements.

Create a vars file named **vars/ebgp-adjustment-data.yml**. It should be identical to `/home/student/proj/vars/ebgp-breakup-data.yml`, but each border router should now have two neighbors. The border routers are **spine01**, **spine02**, and **cs01**.

- 1.1. Create **vars/ebgp-adjustment-data.yml** by copying **proj/vars/ebgp-breakup-data.yml**.

```
[student@workstation ansible-generic-project]$ cp \
> ~/proj/vars/ebgp-breakup-data.yml \
> vars/ebgp-adjustment-data.yml
```

- 1.2. Add the lines shown in bold in the listing below.

```
[student@workstation ansible-generic-project]$ cp \
> vars/ebgp-adjustment-data.yml
bgp_data:
spine01:
as: 10000
router-id: 10.0.0.1/32
networks:
```

```

- 10.10.5.0/30
- 10.10.10.0/30
neighbors:
- { as: 17216, ipv4: 172.16.2.2/30 }
- { as: 19216, ipv4: 172.16.50.2/30 }

spine02:
as: 19216
router-id: 192.168.0.1/32
networks:
- 192.168.5.0/30
- 192.168.10.0/30
neighbors:
- { as: 17216, ipv4: 172.16.5.2/30 }
- { as: 10000, ipv4: 172.16.50.1/30 }

cs01:
as: 17216
router-id: 172.16.0.1/32
networks:
- 172.16.10.0/30
- 192.168.0.0/16
- 10.10.0.0/16
neighbors:
- { as: 10000, ipv4: 172.16.2.1/30 }
- { as: 19216, ipv4: 172.16.5.1/30 }

neighbors:
spine01:
- { as: 17216, ipv4: 172.16.2.2/30 }
- { as: 19216, ipv4: 172.16.50.2/30 }

spine02:
- { as: 17216, ipv4: 172.16.5.2/30 }
- { as: 10000, ipv4: 172.16.50.1/30 }

cs01:
- { as: 10000, ipv4: 172.16.2.1/30 }
- { as: 19216, ipv4: 172.16.5.1/30 }

networks:
spine01:
- 10.10.5.0/30
- 10.10.10.0/30

spine02:
- 192.168.5.0/30
- 192.168.10.0/30

cs01:
- 172.16.10.0/30

```

2. Compose a playbook with a play that configures devices to support new eBGP neighbor relationships and a new set of network advertisements.

Name the new playbook **ebgp-adjustment.yml**. It does the same things as the **ebgp-breakup.yml**, but uses the **vars/ebgp-adjustment-data.yml** vars file instead of **vars/ebgp-breakup-data.yml**.

- 2.1. Create **vars/ebgp-adjustment.yml** by copying **proj/vars/ebgp-breakup.yml**.

```
[student@workstation ansible-generic-project]$ cp \
> /home/student/proj/ebgp-breakup.yml \
> ebpg-adjustment.yml
```

- 2.2. Modify the line shown in bold in the listing below.

```
[student@workstation ansible-generic-project]$ cat ebpg-adjustment.yml
---
- name: play that configures eBGP on VyOS and IOS border routers
  hosts: border-routers
  vars:
    vyos_bgp_tpl: j2/vyos-bgp.j2
    ios_bgp_tpl: j2/ios-bgp.j2
  vars_files:
    - vars/ebgp-adjustment-data.yml

  tasks:
    - name: do not use OSPF to advertise across AS boundaries on cs01
      ios_config:
        lines:
          - passive-interface default
        parents: router ospf 1
      when: inventory_hostname == 'cs01'

    - name: do not use OSPF to advertise across AS boundaries on spines
      vyos_config:
        lines:
          - set protocols ospf passive-interface 'eth5'
          - set protocols ospf passive-interface 'eth3'
      when: inventory_hostname in groups['spines']

    - name: static routes on cs01 to support non-local BGP routes
      ios_config:
        lines:
          - ip route 10.10.0.0 255.255.0.0 GigabitEthernet2 172.16.2.1
          - ip route 192.168.0.0 255.255.0.0 GigabitEthernet3 172.16.5.1
      when: inventory_hostname == 'cs01'

    - name: "map bgp data to ios device using {{ ios_bgp_tpl }}"
      ios_config:
        src: "{{ ios_bgp_tpl }}"
      when: ansible_network_os == 'ios'

    - name: "map bgp data to vyos device using {{ vyos_bgp_tpl }}"
      vyos_config:
        src: "{{ vyos_bgp_tpl }}
```

```
vyos_config:  
  src: "{{ vyos_bgp_tpl }}"  
when: ansible_network_os == 'vyos'
```

- 2.3. The playbook is expecting to use **j2/vyos-bgp.j2** and **j2/ios-bgp.j2**. You should already have copies of these in **/home/student/proj/j2**, so put a copy in **ansible-generic-project/j2/** as well.

```
[student@workstation ansible-generic-project]$ cp \  
> /home/student/proj/j2/*-bgp.j2 j2/
```

3. Perform the play in the new playbook.

```
[student@workstation ansible-generic-project]$ ansible-playbook \  
> ebgp-adjustment.yml
```

4. Verify that the outcome is as desired.

- 4.1. Verify that each border router sees two BGP neighbors (**sh ip bgp sum**).

```
[student@workstation ansible-generic-project]$ ansible -m vyos_command \  
> -a "commands='sh ip bgp sum'" spines  
[student@workstation ansible-generic-project]$ ansible -m ios_command \  
> -a "commands='sh ip bgp sum'" cs01
```

- 4.2. Verify that **spine01** has routes for **192.168.*** by way of **spine02** (172.16.50.2) rather than **cs01**.

```
[student@workstation ansible-generic-project]$ ansible -m vyos_command \  
> -a "commands='sh ip ro 192.168.10.2'" spine01  
spine01 | SUCCESS => {  
  "changed": false,  
  "stdout": [  
    "Routing entry for 192.168.10.0/30\\n Known via \"bgp\\\", distance 20,  
    metric  
20, best\\n Last update 00:04:56 ago\\n * 172.16.50.2, via eth3"  
  ],  
  "stdout_lines": [  
    [  
      "Routing entry for 192.168.10.0/30",  
      " Known via \"bgp\\\", distance 20, metric 20, best",  
      " Last update 00:04:56 ago",  
      " * 172.16.50.2, via eth3"  
    ]  
  ]  
}
```

- 4.3. Verify that **spine02** has routes for **10.10.*** by way of **spine01** (172.16.50.1) rather than **cs01**.

```
[student@workstation ansible-generic-project]$ ansible -m vyos_command \  
> -a "commands='sh ip ro 10.10.10.2'" spine02  
spine02 | SUCCESS => {
```

```
"changed": false,  
"stdout": [  
    "Routing entry for 10.10.10.0/30\n Known via \"bgp\", distance 20, metric  
20, best\n Last update 00:07:57 ago\n * 172.16.50.1, via eth3"  
],  
"stdout_lines": [  
    [  
        "Routing entry for 10.10.10.0/30",  
        " Known via \"bgp\", distance 20, metric 20, best",  
        " Last update 00:07:57 ago",  
        " * 172.16.50.1, via eth3"  
    ]  
]
```


Appendix A

Table of Lab Network Hosts and Groups

The following table summarizes the names of all valid hosts and groups included in the Ansible inventory that is used by hands-on activities in this course.

Valid hosts and groups in the inventory

Name	Type	Member of
access-layer	group	all
border-routers	group	all
cloud-services	group	all
cs01	host	access-layer, border-routers, cloud-services, ios
ftp-server	group	all
ios	group	network
leaf01	host	access-layer, leafs, vyos
leaf02	host	access-layer, leafs, vyos
leafs	group	vyos
local	group	all
localhost	host	local
network	group	all
server01	host	servers
server02	host	servers, ftp-server
server03	host	servers
servers	group	all
spine01	host	border-routers, spines, vyos
spine02	host	border-routers, spines, vyos
spines	group	vyos
vyos	group	network



Important

The group **all** is automatically created by Ansible. Do not create a group named **all** in your inventory file.

A sample **inventory** file will look similar to the following:

```
[user@host ~]$ cat inventory
[leafs]
leaf[01:02]

[spines]
spine[01:02]

[border-routers]
spine01
spine02
cs01

[access-layer]
leaf01
leaf02
cs01

[cloud-services]
cs01

[ios]
cs01

[vyos:children]
spines
leafs

[network:children]
vyos
ios
```


Appendix B

Connection and Authentication Variables

The following table summarizes the connection and authentication variables that you need to set for specific groups in the Ansible inventory used by hands-on activities.

Variables marked "YES" in the "Encrypt?" column are sensitive data and should be contained in files protected by Ansible Vault encryption.

Connection and Authentication Variables

Group	Variable Key: Value	Encrypt?
local	<code>ansible_connection: local</code>	no
network	<code>ansible_connection: network_cli</code>	no
vyos	<code>ansible_network_os: vyos</code>	no
	<code>ansible_user: vyos</code>	YES
	<code>ansible_password: vyos</code>	YES
ios	<code>ansible_network_os: ios</code>	no
	<code>ansible_user: admin</code>	YES
	<code>ansible_password: student</code>	YES

Appendix C

Editing Files with Vim

► Guided Exercise

Editing Files with Vim

In this exercise, you will use the Vim tutorial **vimtutor** that is bundled with the Vim editor to practice basic **vim** editing techniques.

Outcomes

- Basic competency with editing text files in the **vim** text editor, an implementation of vi
- Knowledge of the **vimtutor** tutorial for future learning and practice

Before You Begin

Open a terminal on the **workstation** VM.

- ▶ 1. This exercise will use the existing tutorial program that is bundled with the Vim editor. This tutorial, **vimtutor**, opens a text file in the **vim** editor, which provides instructions on how to edit it in order to practice Vim skills. Quitting and re-running **vimtutor** will provide you with a clean copy of the tutorial.
Your virtual machine will have at least a basic version of **vim** installed, but might not have the fully enhanced version of **vim** which includes the tutorial and help files. Run the command **sudo yum install vim-enhanced** to make sure that it is installed.
- ▶ 2. Run **vimtutor**. Read the Welcome screen and perform Lesson 1.1. Many users of vim only use the keyboard arrow keys for cursor navigation. In **vi**'s early years, users could not rely on working keyboard mappings for arrow keys. Therefore, **vi** was designed to work using only standard character keys for cursor navigation, the conveniently grouped **h**, **j**, **k**, and **l**. Here is one way to remember them: hang back, jump down, kick up, leap forward.
- ▶ 3. Return to the **vimtutor** window. Perform Lesson 1.2. This lesson teaches how to quit **vim** without saving an unwanted change to your file. All changes are lost, but this can be better than leaving a critical file in an incorrect state.
- ▶ 4. Return to the **vimtutor** window. Perform Lesson 1.3.
Vim has faster, more efficient keystrokes to delete an exact amount of words, lines, sentences, and paragraphs. However, as this exercise demonstrates, any editing job can be accomplished using only **x** for single-character deletion.
- ▶ 5. Return to the **vimtutor** window. Perform Lesson 1.4.
The minimum required keystrokes to make an edit to an open file includes **i** to enter edit mode, arrow keys to move the cursor, **Backspace** to delete, and **Esc** to exit edit mode. For most edit tasks, the first key pressed is **i**.
- ▶ 6. (Optional) Return to the **vimtutor** window. Perform Lesson 1.5.
The previous lesson showed the **i** (insert) command as the keystroke to enter edit mode. This **vimtutor** lesson demonstrates that other keystrokes are available to change the initial cursor placement when insert mode is entered.

- ▶ **7.** Return to the vimtutor window. Perform Lesson 1.6.

Save the file by **w**riting and **q**uitting. This completes the minimum set of skills you need to be able to accomplish any text editing task with **vim**.

- ▶ **8.** Return to the **vimtutor** window. Finish by reading the Lesson 1 Summary.

There are six more multi-step lessons in **vimtutor**. None are assigned as further lessons for this exercise, but feel free to use **vimtutor** on your own to learn more about Vim.

Appendix D

IOS Minimal Management Configuration

Restoring IOS Configuration Settings

This is a minimal configuration for the IOS device named **cs01** that is used in the hands-on exercises, so that it provides connectivity to the Management Network.

Older versions of the Cisco CSR1000V instance used in the exercises had an issue that occasionally caused the virtual router to drop its configuration on startup. More information is available at <https://quickview.cloudapps.cisco.com/quickview/bug/CSCut96108>.

You can use the web-based classroom interface to connect directly to the management console of the router in order to enter these commands.



Important

The version that is currently used in this course is not affected by this issue. However, this information might still be useful if the configuration is dropped for some reason.

```
Router>en
Router#config t
Router(config)#service password-encryption
Router(config)#hostname cs01
cs01(config)#ip domain-name lab.example.com
cs01(config)#username admin privilege 15 secret student
cs01(config)#interface GigabitEthernet1
cs01(config-if)#ip address 172.25.250.195 255.255.255.0
cs01(config-if)#no shut
cs01(config-if)#exit
cs01(config)#
cs01(config)#line vty 0 4
cs01(config-line)#transport input ssh
cs01(config-line)#transport output none
cs01(config-line)#login local
cs01(config-line)#exec-timeout 5 0
cs01(config-line)#exit
cs01(config)#crypto key generate rsa general-keys modulus 2048
cs01(config)#ip ssh version 2
cs01(config)#end
cs01#wr mem
cs01#more nvram:startup-config
cs01#sh ver | include register
Configuration register is 0x2102
cs01#
```

Appendix E

Layer 3 Addresses for Lab Network

The following table summarizes the layer 3 IP addresses for the interfaces on network devices and servers in the lab network.

Lab network interfaces and IP addresses

Device	Interface	IPv4	IPv6
cs01	Loopback1	172.16.0.1/32	
	GigabitEthernet2	172.16.2.2/30	fd42:e5a1:ef5d:6030:0:0:0:2/64
	GigabitEthernet3	172.16.5.2/30	fdcd:b497:c72:de1c:0:0:0:2/64
	GigabitEthernet4	172.16.10.1/30	fdfb:ede0:bdf2:c094:0:0:0:1/64
leaf01	lo	10.0.0.11/32	
	eth1	10.10.10.1/30	fdbc:bda:8486:7118:0:0:0:1/64
	eth5	10.10.5.2/30	fdb5:4b4e:4574:c6bb:0:0:0:2/64
leaf02	lo	192.168.0.2/32	
	eth2	192.168.10.1/30	fdea:230f:c3cf:c287:0:0:0:1/64
	eth6	192.168.5.2/30	fdaa:d398:13a1:5a42:0:0:0:2/64
spine01	lo	10.0.0.1/32	
	eth1	10.10.5.1/30	fdb5:4b4e:4574:c6bb:0:0:0:1/64
	eth5	172.16.2.1/30	fd42:e5a1:ef5d:6030:0:0:0:1/64
spine02	lo	192.168.0.1/32	
	eth2	192.168.5.1/30	fdaa:d398:13a1:5a42:0:0:0:1/64
	eth5	172.16.5.1/30	fdcd:b497:c72:de1c:0:0:0:1/64
server01	eth1	10.10.10.2/30	
server02	eth2	192.168.10.2/30	
server03	eth1	172.16.10.2/30	

Appendix F

Ansible Variable Types

Ansible Variable Types and Precedence

This is a list of the different types of variables that are available in Ansible.

The list is arranged in reverse order of precedence. If the same variable name is set to different values by different methods from this list, the methods later in the list override values set by the earlier ones. A value set by the second method on the list overrides a value set by the first, the third overrides the second, and variables of the last type in this list ("extra variables" set on the command line) override values set all of the other methods.

1. Role defaults are variables defined as defaults within roles.

```
$ cat roles/myrole/defaults/main.yml  
a_variable_name: myvalue
```

2. Inventory file group variables are group variables that are defined in an inventory file.

```
[atlanta]  
host1  
host2  
  
[atlanta:vars]  
ntp_server=ntp.atlanta.example.com  
proxy=proxy.atlanta.example.com
```

3. Inventory **group_vars/all** variables are defined within an **all** group variables file found within the **group_vars** directory tree adjacent to the current inventory file.

```
.  
|__ group_vars  
|   |__ all  
|__ inventory
```

```
[user@host ~]# cat group_vars/all  
a_variable_name: myvalue
```

4. Playbook **group_vars/all** variables are defined within an **all** group variables file found within the **group_vars** directory tree adjacent to the current playbook.

```
.  
|__ group_vars  
|   |__ all  
|__ myplaybook.yml
```

```
[user@host ~]# cat group_vars/all  
a_variable_name: myvalue
```

5. Inventory **group_vars/*** variables are defined with a particular, group variables file other than **all** within the **group_vars** directory tree adjacent to the current inventory file.

Appendix F | Ansible Variable Types

```
.  
└── group_vars  
    └── some-group  
└── inventory  
  
[user@host ~]# cat group_vars/some-group  
a_variable_name: myvalue
```

6. Playbook **group_vars/*** variables are defined with a particular group variables file other than **all** within the **group_vars** directory tree adjacent to the current inventory file.

```
.  
└── group_vars  
    └── some-group  
└── myplaybook.yml  
  
[user@host ~]# cat group_vars/some-group  
a_variable_name: myvalue
```

7. Inventory file host variables are host variables defined in an inventory file.

```
[atlanta]  
host1 http_port=80 maxRequestsPerChild=808  
host2 http_port=303 maxRequestsPerChild=909
```

8. Inventory **host_vars/*** variables are variables that are defined in a host-specific variables file under the **host_vars** directory that is adjacent to the current inventory file.

```
.  
└── host_vars  
    └── some-host  
└── inventory  
  
[user@host ~]# cat host_vars/some-host  
a_variable_name: myvalue
```

9. Playbook **host_vars/*** variables are variables that are defined in a host-specific variables file under the **host_vars** directory that is adjacent to the current playbook file.

```
.  
└── host_vars  
    └── some-host  
└── myplaybook.yml  
  
[user@host ~]# cat host_vars/some-host  
a_variable_name: myvalue
```

10. Host facts are variables associated with a host and are defined in a particular way (as facts).

Appendix F | Ansible Variable Types

```
- set_fact:  
  one_fact: foo  
  other_fact: bar
```

11. Play variables are defined in the **vars** block that appears at the top of a play.

```
---  
- name: this is a play  
  hosts: groupname  
  vars:  
    foo: foovalue  
    bar: barvalue
```

12. Play **vars_prompt** variables are defined in the **vars_prompt** block that appears at the top of a play.

```
---  
- hosts: all  
  remote_user: root  
  
  vars:  
    from: "camelot"  
  
  vars_prompt:  
    - name: "name"  
      prompt: "what is your name?"  
    - name: "quest"  
      prompt: "what is your quest?"  
    - name: "favcolor"  
      prompt: "what is your favorite color?"
```

13. Play **vars_files** variables are defined in a file that is loaded by virtue of having its file path listed in the **vars_files** block that appears at the top of a play.

```
$ cat vars/example.yml  
foovar: fooval  
barvar: barval  
  
$ cat myplaybook.yml  
---  
- name: my playbook  
  hosts: myhosts  
  vars_files:  
    - vars/example.yml
```

14. Role variables are defined in **role/vars/main.yml**.

```
$ cat roles/myrole/vars/main.yml  
foovar: fooval  
barvar: barval
```

Appendix F | Ansible Variable Types

15. Block variables are defined under the **vars** heading within a block of Ansible statements that are logically grouped using the **block** module.

```
- block:  
  - debug:  
    var: var_for_block  
  
  vars:  
    var_for_block: "value for var_for_block"
```

16. Role parameters are defined under the **vars** heading that appears as an argument in conjunction with **include_role**.

```
---  
- hosts: webservers  
  tasks:  
  - include_role:  
    name: foo_app_instance  
  vars:  
    dir: '/opt/a'  
    app_port: 5000
```

17. Include parameters are variables defined in conjunction with an **include** statement.

```
---  
- name: This is an example play  
  hosts: agroup  
  vars:  
    foo: bar  
  
- include: amazon.yml  
  vars:  
    application: FooServer  
    instance_type: t2.micro  
    instance_count: 1  
- include: amazon.yml  
  vars:  
    application: BarServer  
    instance_type: t2.micro  
    instance_count: 1
```

18. Extra variables (extra vars) are defined at the command line using the **--extra-vars (-e)** option.

```
$ ansible-playbook -e "version=1.23.45 another_variable=foobarino" myplaybook.yml
```


Appendix G

Changing the Screen Resolution

Changing the Screen Resolution

You may want to change the screen resolution on your graphical desktop (workstation) to accommodate certain applications. After logging in as student on the workstation console, navigate to **Applications** → **System Tools** → **Settings**. From that folder, choose **Devices**, then choose **Displays**. From the list of displays, choose **Unknown Display**. From the **Resolution** drop-down menu, choose a setting that works for you and the application. Press the **Apply** button, then press the **Keep Changes** button.

Remember that if you reset the workstation VM, these display settings will be lost and you will need to redo them.