



Red Hat Training and Certification

Student Workbook (ROLE)

Red Hat Enterprise Linux 9.0 RH294

Red Hat Enterprise Linux Automation with Ansible

Edition 2





Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



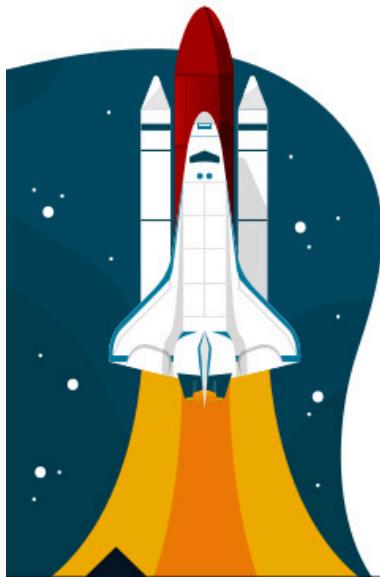
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Red Hat Enterprise Linux Automation with Ansible



Red Hat Enterprise Linux 9.0 RH294
Red Hat Enterprise Linux Automation with Ansible
Edition 2 20221117
Publication date 20221117

Authors: Mike Kelly, Ed Parenti, Morgan Weetman
Course Architect: Steven Bonneville
DevOps Engineer: Dan Kolepp
Editors: Nicole Muller, David O'Brien, Sam Ffrench

Copyright © 2022 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2022 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, OpenShift, Fedora, Hibernate, Ansible, CloudForms, RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries 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 American, Inc. and/or its affiliates.

XFS® is a registered trademark of Hewlett Packard Enterprise Development LP or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is a trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

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.

Contributors: Artur Glogowski, Travis Michette, Samik Sanyal, Michael Phillips, Patrick Gomez, Roberto Velazquez

Portions of this course were adapted from the Ansible Lightbulb project. The material from that project is available from <https://github.com/ansible/lightbulb> under the MIT License.

Document Conventions	ix
	ix
Introduction	xi
Red Hat Enterprise Linux Automation with Ansible	xi
Orientation to the Classroom Environment	xii
Obtaining a Trial Subscription to Red Hat Ansible Automation Platform	xvii
1. Introducing Ansible	1
Automating Linux Administration with Ansible	2
Quiz: Automating Linux Administration with Ansible	8
Installing Ansible	10
Guided Exercise: Installing Ansible	16
Summary	18
2. Implementing an Ansible Playbook	19
Building an Ansible Inventory	20
Guided Exercise: Building an Ansible Inventory	26
Managing Ansible Configuration Files	31
Guided Exercise: Managing Ansible Configuration Files	39
Writing and Running Playbooks	45
Guided Exercise: Writing and Running Playbooks	53
Implementing Multiple Plays	58
Guided Exercise: Implementing Multiple Plays	67
Lab: Implementing an Ansible Playbook	74
Summary	81
3. Managing Variables and Facts	83
Managing Variables	84
Guided Exercise: Managing Variables	93
Managing Secrets	99
Guided Exercise: Managing Secrets	105
Managing Facts	108
Guided Exercise: Managing Facts	117
Lab: Managing Variables and Facts	124
Summary	137
4. Implementing Task Control	139
Writing Loops and Conditional Tasks	140
Guided Exercise: Writing Loops and Conditional Tasks	152
Implementing Handlers	156
Guided Exercise: Implementing Handlers	159
Handling Task Failure	165
Guided Exercise: Handling Task Failure	170
Lab: Implementing Task Control	178
Summary	187
5. Deploying Files to Managed Hosts	189
Modifying and Copying Files to Hosts	190
Guided Exercise: Modifying and Copying Files to Hosts	196
Deploying Custom Files with Jinja2 Templates	205
Guided Exercise: Deploying Custom Files with Jinja2 Templates	210
Lab: Deploying Files to Managed Hosts	213
Summary	221
6. Managing Complex Plays and Playbooks	223
Selecting Hosts with Host Patterns	224
Guided Exercise: Selecting Hosts with Host Patterns	233

Including and Importing Files	249
Guided Exercise: Including and Importing Files	255
Lab: Managing Complex Plays and Playbooks	261
Summary	270
7. Simplifying Playbooks with Roles and Ansible Content Collections	271
Describing Role Structure	272
Quiz: Describing Role Structure	280
Creating Roles	282
Guided Exercise: Creating Roles	288
Deploying Roles from External Content Sources	295
Guided Exercise: Deploying Roles from External Content Sources	302
Getting Roles and Modules from Content Collections	309
Guided Exercise: Getting Roles and Modules from Content Collections	316
Reusing Content with System Roles	324
Guided Exercise: Reusing Content with System Roles	333
Lab: Simplifying Playbooks with Roles and Ansible Content Collections	339
Summary	352
8. Troubleshooting Ansible	353
Troubleshooting Playbooks	354
Guided Exercise: Troubleshooting Playbooks	362
Troubleshooting Ansible Managed Hosts	369
Guided Exercise: Troubleshooting Ansible Managed Hosts	377
Lab: Troubleshooting Ansible	384
Summary	393
9. Automating Linux Administration Tasks	395
Managing Software and Subscriptions	396
Guided Exercise: Managing Software and Subscriptions	406
Managing Users and Authentication	414
Guided Exercise: Managing Users and Authentication	421
Managing the Boot Process and Scheduled Processes	428
Guided Exercise: Managing the Boot Process and Scheduled Processes	433
Managing Storage	443
Guided Exercise: Managing Storage	451
Managing Network Configuration	456
Guided Exercise: Managing Network Configuration	464
Lab: Automating Linux Administration Tasks	469
Summary	480
10. Comprehensive Review: Red Hat Enterprise Linux Automation with Ansible	481
Comprehensive Review	482
Lab: Deploying Ansible	485
Lab: Creating Playbooks	495
Lab: Managing Linux Hosts and Using System Roles	503
Lab: Creating Roles	518

Document Conventions

This section describes various conventions and practices that are used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:



References

References describe where to find external documentation that is 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 something that makes your life easier.



Important

They provide details of information that is easily missed: configuration changes that apply only to the current session, or services that need restarting before an update applies. Ignoring these admonitions will not cause data loss, but might cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services that are covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Red Hat Enterprise Linux Automation with Ansible

Red Hat Enterprise Linux Automation with Ansible (RH294) is intended for Linux system administrators and developers who need to automate provisioning, configuration, application deployment, and orchestration.

Students learn how to install and configure Ansible on a management workstation and prepare managed hosts for automation. Students write Ansible Playbooks to automate tasks, and run them to ensure servers are correctly deployed and configured. Examples of approaches to automate common Linux system administration tasks are explored.

Course Objectives

- Automate common Red Hat Enterprise Linux system administration tasks by using Ansible.
- Install and configure automation content navigator from Red Hat Ansible Automation Platform to run Ansible Playbooks in a container-based automation execution environment.
- Create and manage inventories of managed hosts, and prepare the hosts for connections from Ansible.
- Write effective Ansible Playbooks.
- Reuse code and simplify playbook development with Ansible Roles and Ansible Content Collections.

Audience

- Linux system administrators, DevOps engineers, Site Reliability Engineers, infrastructure automation engineers, and developers responsible for: automating configuration management; ensuring consistent and repeatable application deployment; the provisioning and deployment of development, testing, and production servers; and integrating DevOps continuous integration/continuous delivery workflows.

Prerequisites

- Red Hat Certified System Administrator (EX200/RHCSA) certification or equivalent Red Hat Enterprise Linux knowledge and experience.

Orientation to the Classroom Environment

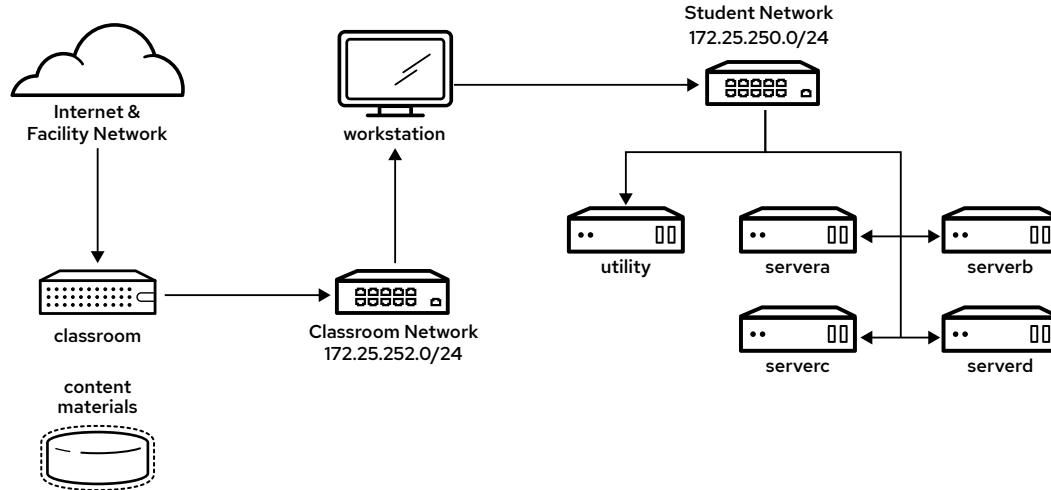


Figure 0.1: Classroom environment

In this course, the main computer system used for hands-on learning activities is **workstation**. Four other machines are also used by students for these activities: **servera**, **serverb**, **serverc**, and **serverd**. All these five systems are in the `lab.example.com` DNS domain.

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

Classroom Machines

Machine name	IP addresses	Role
<code>bastion.lab.example.com</code>	172.25.250.254	Gateway system to connect student private network to classroom server (must always be running)
<code>utility.lab.example.com</code>	172.25.250.8	System with utility services required for the classroom
<code>workstation.lab.example</code>	172.25.250.9	Graphical workstation used for system administration
<code>servera.lab.example.com</code>	172.25.250.10	Host managed with Ansible
<code>serverb.lab.example.com</code>	172.25.250.11	Host managed with Ansible
<code>serverc.lab.example.com</code>	172.25.250.12	Host managed with Ansible
<code>serverd.lab.example.com</code>	172.25.250.13	Host managed with Ansible

Introduction

The primary function of **bastion** is that it acts as a router between the network that connects the student machines and the classroom network. If **bastion** is down, other student machines will only be able to access systems on the individual student network.

Several systems in the classroom provide supporting services. Two servers, **content.example.com** and **materials.example.com**, are sources for software and lab materials used in hands-on activities. Information on how to use these servers is provided in the instructions for those activities. These are provided by the **classroom.example.com** virtual machine. Both **classroom** and **bastion** should always be running for proper use of the lab environment.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning (ROLE) classroom. Self-paced courses are accessed through a web application that is hosted at rol.redhat.com [<http://rol.redhat.com>]. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

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

The screenshot shows a web-based interface for managing a lab environment. At the top, there are tabs for 'Table of Contents', 'Course' (which is selected), 'Lab Environment', and icons for a star and a question mark. Below the tabs, there's a section for 'SSH Private Key' with a download button. A 'Lab Controls' section contains instructions for creating and deleting the lab environment, along with a 'WATCH TUTORIAL' button. The main area displays a table of virtual machines:

Name	Status	Action	Open Console
bastion	active	ACTION -	OPEN CONSOLE
classroom	active	ACTION -	OPEN CONSOLE
servera	active	ACTION -	OPEN CONSOLE
serverb	active	ACTION -	OPEN CONSOLE
utility	active	ACTION -	OPEN CONSOLE
workstation	active	ACTION -	OPEN CONSOLE

Figure 0.2: An example course Lab Environment management page

Machine States

Virtual Machine State	Description
building	The virtual machine is being created.
active	The virtual machine is running and available. If it just started, it still might be starting services.
stopped	The virtual machine is completely shut down. On starting, the virtual machine boots into the same state it was in before shutdown. The disk state is preserved.

Classroom Actions

Button or Action	Description
CREATE	Create the ROLE classroom. Creates and starts all the virtual machines needed for this classroom. Creation can take several minutes to complete.
CREATING	The ROLE classroom virtual machines are being created. Creates and starts all the virtual machines that are needed for this classroom. Creation can take several minutes to complete.
DELETE	Delete the ROLE classroom. Destroys all virtual machines in the classroom. All saved work on those systems' disks is lost.
START	Start all virtual machines in the classroom.
STARTING	All virtual machines in the classroom are starting.
STOP	Stop all virtual machines in the classroom.

Machine Actions

Button or Action	Description
OPEN CONSOLE	Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the workstation virtual machine only, and from there, 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 disk contents.
ACTION > Power Off	Forcefully shut down the virtual machine, while still preserving disk contents. This is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset associated storage to its initial state. All saved work on that system's disks is lost.

Introduction

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

At the start of an exercise, if instructed to reset all virtual machines, click **ACTION > Reset** on every virtual machine in the list.

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



Warning

The **DELETE** operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom environment when the appropriate timer expires.

To adjust the timers, locate the two + buttons at the bottom of the course management page. Click the auto-stop + button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Performing Lab Exercises

You might see four types of lab activities in this course:

- A *guided exercise* is a hands-on practice exercise that follows a presentation section. It takes you step-by-step through a procedure to perform.
- A *quiz* is typically used when checking knowledge-based learning, or when a hands-on activity is impractical for some other reason.
- An *end-of-chapter lab* is a gradable hands-on activity to help you check your learning. You are provided with a set of high-level steps to perform, based on the guided exercises in that chapter, but the steps do not walk you through every command. You are also given a solution that provides a step-by-step walk-through.
- A *comprehensive review lab* is used at the end of the course. It is also a gradable hands-on activity, but it covers content from throughout the entire course. You are provided with a specification that details what you need to accomplish in the activity, but not the specific steps to do so. Again, you are given a solution that provides a step-by-step walk-through that meets the specification.

To prepare your lab environment at the start of each hands-on activity, run the `lab start` command with an activity name specified by the activity's instructions. Likewise, at the end of each hands-on activity, run the `lab finish` command with that same activity name to clean up after the activity. Each hands-on activity has a unique name within a course.

The syntax for running an exercise script is as follows:

```
[student@workstation ~]$ lab action exercise
```

The **action** is a choice of **start**, **grade**, or **finish**. All exercises support **start** and **finish** actions. Only end-of-chapter labs and comprehensive review labs support the **grade** action.

start

The **start** action verifies the required resources to begin an exercise. It might include configuring settings, creating resources, checking prerequisite services, and verifying necessary outcomes from previous exercises. You can take an exercise at any time, even without taking preceding exercises.

grade

For gradable activities, the **grade** action directs the **lab** command to evaluate your work, and displays a list of grading criteria with a **PASS** or **FAIL** status for each. To achieve a **PASS** status for all criteria, fix any failures and rerun the **grade** action.

finish

The **finish** action cleans up resources configured during the exercise. You can take an exercise as many times as you want.

The **lab** command supports tab completion. For example, to list all exercises that you can start, enter **lab start** and then press the Tab key twice.

Obtaining a Trial Subscription to Red Hat Ansible Automation Platform

Objectives

Get a trial subscription to Red Hat Ansible Automation Platform and access cloud-based services.

Evaluating Red Hat Ansible Automation Platform

This section provides information on one way to get Red Hat Ansible Automation Platform software for evaluation outside the context of this course.



Important

You do not need to do anything in this section to complete this course.

This section provides information on one way to get access to Red Hat Ansible Automation Platform for your own evaluation and study outside the lab environment.

The necessary software is already available to you in this course's lab environment.

Accessing the Red Hat Hybrid Cloud Console

The Red Hat Hybrid Cloud Console (<https://console.redhat.com>) is a Software-as-a-Service (SaaS) offering that hosts services and applications available to customers.

The platform provides services for several Red Hat products. For example, you can use the OpenShift Cluster application to monitor your clusters and access reporting tools. Insights for Red Hat Enterprise Linux can alert you to security or stability issues with your systems.

For Ansible Automation Platform, the Red Hat Hybrid Cloud Console offers several services:

- The *automation hub* service hosts supported Ansible Content Collections from Red Hat and its partners.
- The *Red Hat Insights for Red Hat Ansible Automation Platform* service, named **Insights** in the web interface, collects data from your automation controller systems and generates graphical reports that can help you better understand automation utilization in your organization.

Authenticating to the Red Hat Hybrid Cloud Console

Use your customer portal username and password to authenticate to the Red Hat Hybrid Cloud Console. To access the Ansible Automation Platform services, you need a valid Ansible Automation Platform subscription.

Creating a Personal Account and Acquiring a Trial Subscription

If you use a corporate account to access the Red Hat Hybrid Cloud Console, then all the users within your organization share your configuration. For example, if you register an automation controller system with Red Hat Insights, then all the users within the organization see that system.

**Warning**

Never use your corporate account for testing purposes. The test configuration you perform might break your existing organization configuration.

If you do not have a customer portal account or do not want to use your corporate account, then create a personal account. Navigate to <https://access.redhat.com/>, click the user icon at the upper left, and then click **Register**.

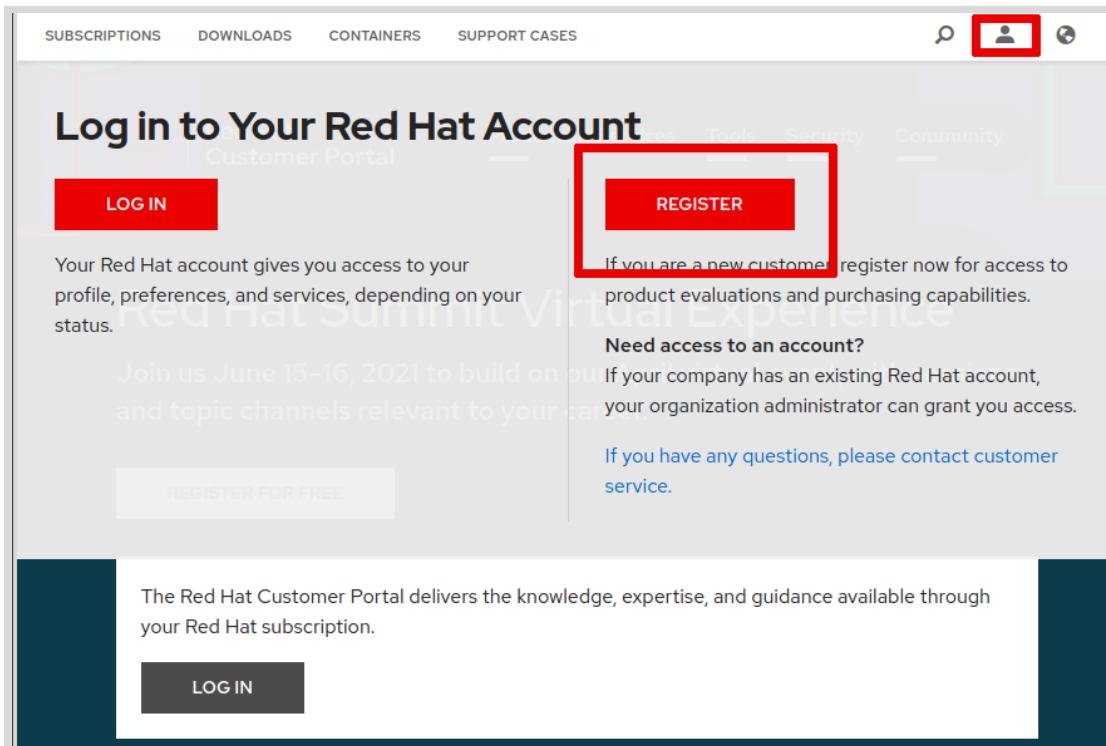


Figure 0.3: Creating a personal account

Select **Personal** for the **Account Type** and then enter your personal information.

Alternatively, you can create your personal account from <https://developers.redhat.com/>. Click **Log In** and then click **Don't have an account? Create one now**. Enter your personal information and then click **Create my Account**.

To get an Ansible Automation Platform subscription, enroll in the Ansible Automation Platform trial. That subscription allows you to access the Ansible Automation Platform services on the Red Hat Hybrid Cloud Console. Navigate to <https://console.redhat.com/ansible> and request an evaluation.

Ansible Automation Platform services requires a valid subscription ×



Get analytics and knowledge of your automation, access to certified content, and more with a Red hat Ansible Automation Platform subscription.

Try it Learn More Not now

Figure 0.4: Enrolling in the Ansible Automation Platform trial



Note

It might take up to two hours for your trial subscription to be available. During that period, the preceding trial window displays every time you access the Ansible Automation Platform services.



References

Try Red Hat Ansible Automation Platform

<https://www.redhat.com/en/technologies/management/ansible/try-it>

Chapter 1

Introducing Ansible

Goal

Describe the fundamental concepts of Ansible and how it is used, and install development tools from Red Hat Ansible Automation Platform.

Objectives

- Describe the motivation for automating Linux administration tasks with Ansible, fundamental Ansible concepts, and the basic architecture of Ansible.
- Install Ansible on a control node and describe the distinction between community Ansible and Red Hat Ansible Automation Platform.

Sections

- Automating Linux Administration with Ansible (and Quiz)
- Installing Ansible (and Guided Exercise)

Automating Linux Administration with Ansible

Objective

- Describe the motivation for automating Linux administration tasks with Ansible, fundamental Ansible concepts, and the basic architecture of Ansible.

Automation and Linux System Administration

For many years, most system administration and infrastructure management has relied on manual tasks performed through graphical or command-line user interfaces. System administrators often work from checklists, other documentation, or a memorized routine to perform standard tasks.

This approach is error-prone. It is easy for a system administrator to skip a step or make a mistake on a step. Verification that the steps were performed properly or that they result in the expected outcome is often limited.

Furthermore, by managing each server manually and independently, it is very easy for many servers that are supposed to be identical in configuration to be different in minor (or major) ways. This can make maintenance more difficult and introduce errors or instability into the IT environment.

Automation can help avoid the problems caused by manual system administration and infrastructure management. As a system administrator, you can use automation to ensure that all your systems are quickly and correctly deployed and configured. Consequently, you can automate the repetitive tasks in your daily schedule, freeing up your time and enabling you to focus on more critical tasks. For your organization, automation can help you to more quickly roll out the next version of an application or updates to a service.

Infrastructure as Code

A good automation system allows you to implement *Infrastructure as Code* practices.

Infrastructure as Code means that you can use a machine-readable automation language to define and describe the required state of your IT infrastructure. Ideally, this automation language should also be easy for humans to read, because then you can more easily understand the current state and make changes to it. This code is then applied to your infrastructure to ensure that it is actually in that state.

If the automation language is represented as simple text files, it is easy to manage in a version control system. The advantage of this is that every change can be checked into the version control system, ensuring that you have an ongoing history of changes. If you want to revert to an earlier known-good configuration, you can check out that version and apply it to your infrastructure.

This builds a foundation to help you follow best practices in DevOps. Developers can define their desired configuration in the automation language. Operators can review those changes more easily to provide feedback, and use that automation to reproducibly ensure that systems are in the state that developers expect.

Mitigating Human Error

Reducing the number of tasks performed manually on servers by using task automation and Infrastructure as Code practices can help ensure that your servers are consistently configured more often.

This means that you need to become accustomed to making changes by updating your automation code, rather than manually applying them to your servers. Otherwise, you run the risk of losing manually applied changes the next time you apply changes using automation.

You can use code review, peer review by multiple subject matter experts, and document the procedure within the automation content to reduce your operational risks.

Ultimately, you can enforce that changes to your IT infrastructure be made through automation to mitigate human error.

What Is Ansible?

Ansible is an open source automation platform. It is a *simple automation language* that can accurately describe an IT application infrastructure in Ansible Playbooks. It is also an *automation engine* that runs Ansible Playbooks.

Ansible can manage powerful automation tasks and can adapt to many workflows and environments. At the same time, new users of Ansible can very quickly use it to become productive.

Ansible Is Simple

Ansible Playbooks provide human-readable automation. This means that playbooks are automation tools that are also easy for humans to read, comprehend, and change. No special coding skills are required to write them. Playbooks execute tasks in order. The simplicity of playbook design makes them usable by every team, which allows people new to Ansible to get productive quickly.

Ansible Is Powerful

You can use Ansible to deploy applications for configuration management, for workflow automation, and for network automation. You can use Ansible to orchestrate the entire application lifecycle.

Ansible Is Agentless

Ansible is built around an *agentless architecture*. Typically, Ansible connects to the hosts it manages by using OpenSSH or WinRM and runs tasks, often (but not always) by pushing out small programs called *Ansible modules* to those hosts. These programs are used to put the system in a specific desired state. Any modules that are pushed are removed when Ansible has finished its tasks. You can start using Ansible almost immediately because no special agents need to be approved for use and then deployed to the managed hosts. Because there are no agents and no additional custom security infrastructure, Ansible is more efficient and more secure than other alternatives.

Ansible has a number of important strengths:

- *Cross platform support:* Ansible provides agentless support for Linux, Windows, UNIX, and network devices, in physical, virtual, cloud, and container environments.

- *Human-readable automation:* Ansible Playbooks, written as YAML text files, are easy to read and help ensure that everyone understands what they do.
- *Precise application descriptions:* Every change can be made by Ansible Playbooks, and every aspect of your application environment can be described and documented.
- *Easy to manage in version control:* Ansible Playbooks and projects are plain text. They can be treated like source code and placed in your existing version control system.
- *Support for dynamic inventories:* The list of machines that Ansible manages can be dynamically updated from external sources to capture the correct, current list of all managed servers all the time, regardless of infrastructure or location.
- *Orchestration that integrates easily with other systems:* HP SA, Puppet, Jenkins, Red Hat Satellite, and other systems that exist in your environment can be leveraged and integrated into your Ansible workflow.

Ansible: The Language of DevOps



Figure 1.1: Ansible across the application lifecycle

Communication is the key to DevOps. Ansible is the first automation language that can be read and written across IT.

Ansible Concepts and Architecture

The Ansible architecture consists of two types of machines: *control nodes* and *managed hosts*. Ansible is installed and run from a control node, and this machine also has copies of your Ansible project files.

Managed hosts are listed in an *inventory*, which also organizes those systems into groups for easier collective management. You can define the inventory statically in a text file, or dynamically using scripts that obtain group and host information from external sources.

Instead of writing complex scripts, Ansible users create high-level *plays* to ensure that a host or group of hosts is in a particular state. A play performs a series of *tasks* on the hosts, in the order specified by the play. These plays are expressed in YAML format in a text file. A file that contains one or more plays is called a *playbook*.

Each task runs a *module*, a small piece of code (written in Python, PowerShell, or some other language), with specific arguments. Each module is essentially a tool in your toolkit. Ansible ships with hundreds of useful modules that can perform a wide variety of automation tasks. They can act on system files, install software, or make API calls.

When used in a task, a module generally ensures that some particular aspect of the machine is in a particular state. For example, a task using one module might ensure that a file exists and has particular permissions and content. A task using a different module might ensure that a particular

file system is mounted. If the system is not in that state, the task should put it in that state, or do nothing. If a task fails, the default Ansible behavior is to abort the rest of the playbook for the hosts that had a failure and continue with the remaining hosts.

Tasks, plays, and playbooks are designed to be *idempotent*. This means that you can safely run a playbook on the same hosts multiple times. When your systems are in the correct state, the playbook makes no changes when you run it. Numerous modules are available that you can use to run arbitrary commands. However, you must use those modules with care to ensure that they run in an idempotent way.

Ansible also uses *plug-ins*. Plug-ins are code that you can add to Ansible to extend it and adapt it to new uses and platforms.

The Ansible architecture is agentless. Typically, when an administrator runs an Ansible Playbook, the control node connects to the managed host by using SSH (by default) or WinRM. This means that you do not need to have an Ansible-specific agent installed on managed hosts, and do not need to permit any additional communication between the control node and managed hosts.

Getting Support for Ansible

Red Hat Ansible Automation Platform is a fully supported version of Ansible that enables enterprises to manage their automation at scale.

It provides the following benefits:

- Official support for the core Ansible toolset.
- Red Hat Certified Ansible Content Collections to help you accelerate adoption of Ansible automation with supported code.
- On-premise tools to help you centralize delivery of automation content, manage automation tasks, and scale distribution of automation execution.
- Cloud services to help you discover certified Ansible content, facilitate team collaboration, and provide operational analytics to automate mixed, hybrid environments.

For example, its automation controller component (formerly called Red Hat Ansible Tower) is an enterprise framework that you can use to control who has access to run playbooks on which hosts, share the use of SSH credentials without allowing users to transfer them or see their contents, log all your Ansible jobs, and manage inventory, among many other things. It provides a browser-based user interface (web UI) and a RESTful API. The upstream Ansible community does not automatically include this component with the community Ansible distribution, but it is developed as open source and is provided and supported as part of the Red Hat Ansible Automation Platform product.

The Ansible Way

The following goals were used during the design of Ansible.

Complexity Kills Productivity

Simpler is better. Ansible is designed so that its tools are simple to use and automation is simple to write and read. You should take advantage of this to strive for simplification in how you create your automation.

Optimize for Readability

The Ansible automation language is built around simple, declarative, text-based files that are easy for humans to read. Written properly, Ansible Playbooks can clearly document your workflow automation.

Think Declaratively

Ansible is a *desired-state engine*. It approaches the problem of how to automate IT deployments by expressing them in terms of the state that you want your systems to be in. The goal of Ansible is to put your systems into the desired state, only making changes that are necessary. Trying to treat Ansible like a scripting language is not the right approach.

Use Cases

Unlike some other tools, Ansible combines orchestration with configuration management, provisioning, and application deployment in one easy-to-use platform.

Some use cases for Ansible include:

Configuration Management

Centralizing configuration file management and deployment is a common use case for Ansible, and it is how many power users are first introduced to the Ansible automation platform.

Application Deployment

When you define your application with Ansible, and manage the deployment with automation controller, development teams can effectively manage the entire application lifecycle from development to production.

Provisioning

Applications have to be deployed or installed on systems. Ansible and automation controller can help streamline the process of provisioning systems, whether you are PXE booting and kickstarting bare-metal servers or virtual machines, or creating virtual machines or cloud instances from templates.

Continuous Delivery

Creating a CI/CD pipeline requires coordination and buy-in from numerous teams. You cannot do it without a simple automation platform that everyone in your organization can use. Ansible Playbooks keep your applications properly deployed and managed throughout their entire lifecycle.

Security and Compliance

When your security policy is defined in Ansible Playbooks, scanning for and remediation of potential security issues can be integrated into other automated processes. Instead of being an afterthought, it is an integral part of everything that is deployed.

Orchestration

Configurations alone do not define your environment. You need to define how multiple configurations interact, and ensure that the disparate pieces can be managed as a whole.



References

Ansible

<https://www.ansible.com>

How Ansible Works

<https://www.ansible.com/how-ansible-works>

Red Hat Ansible Automation Platform

<https://www.redhat.com/en/technologies/management/ansible>

Introducing Ansible Automation Platform 2

<https://www.ansible.com/blog/introducing-ansible-automation-platform-2>

Announcing the Community Ansible 3.0.0 Package

<https://www.ansible.com/blog/announcing-the-community-ansible-3.0.0-package>

► Quiz

Automating Linux Administration with Ansible

Choose the correct answer to the following questions:

- ▶ 1. Which term best describes the Ansible architecture?
 - a. Agentless
 - b. Client/Server
 - c. Event-driven
 - d. Stateless

- ▶ 2. Which network protocol does Ansible use by default to communicate with managed nodes?
 - a. HTTP
 - b. HTTPS
 - c. SNMP
 - d. SSH

- ▶ 3. Which file defines the actions that Ansible performs on managed nodes?
 - a. Host inventory
 - b. Manifest
 - c. Playbook
 - d. Script

- ▶ 4. Which syntax is used to define Ansible Playbooks?
 - a. Bash
 - b. Perl
 - c. Python
 - d. YAML

► Solution

Automating Linux Administration with Ansible

Choose the correct answer to the following questions:

- ▶ 1. **Which term best describes the Ansible architecture?**
 - a. Agentless
 - b. Client/Server
 - c. Event-driven
 - d. Stateless

- ▶ 2. **Which network protocol does Ansible use by default to communicate with managed nodes?**
 - a. HTTP
 - b. HTTPS
 - c. SNMP
 - d. SSH

- ▶ 3. **Which file defines the actions that Ansible performs on managed nodes?**
 - a. Host inventory
 - b. Manifest
 - c. Playbook
 - d. Script

- ▶ 4. **Which syntax is used to define Ansible Playbooks?**
 - a. Bash
 - b. Perl
 - c. Python
 - d. YAML

Installing Ansible

Objectives

Install Ansible on a control node and describe the distinction between community Ansible and Red Hat Ansible Automation Platform.

Ansible and Red Hat Ansible Automation Platform

You can obtain Ansible software in different ways, each with their own level of support.

- From the upstream community
- As part of Red Hat Enterprise Linux, with limited support
- With the fully supported Red Hat Ansible Automation Platform product

This course focuses on the last of these three, using the tools provided with Red Hat Ansible Automation Platform. However, the Ansible language and basic concepts are the same no matter how you obtain the software.

Community Ansible

The upstream Ansible community develops Ansible and distributes versions of it in two ways.

The first of these is *Ansible Core*. This is a minimalist component that consists of the core runtime that can interpret Ansible content and a set of commonly used Ansible modules (included as the `ansible.builtin` Ansible Content Collection). This runtime is structured so that the control node acts as the execution environment for Ansible code.

The second is *community Ansible*. This is a distribution of Ansible Core plus a selection of other Ansible Content Collections selected by the open source community, adding additional Ansible modules and roles.

Both are provided by the upstream developers as Python pip packages; neither community version is supported by Red Hat.

Ansible Core in Red Hat Enterprise Linux

Red Hat provides Ansible Core as an RPM package, `ansible-core`, included with Red Hat Enterprise Linux 9 in the AppStream repository. It is intended to enable support for automation code provided or generated by Red Hat. It is supported, but the scope of support is limited to any Ansible Playbooks, roles, or modules that are included with or generated by a Red Hat product, such as the system roles included in the `rhel-system-roles` package, Red Hat Insights remediation playbooks, and OpenSCAP compliance Ansible Playbooks. Other use cases, including using the other Ansible modules and plug-ins included with Ansible Core 2.13, are outside the scope of support.

For more information, see the Knowledgebase article "Using Ansible in RHEL 9" [<https://access.redhat.com/articles/6393321>].

Red Hat Ansible Automation Platform

Red Hat provides a fully supported version of Ansible through Red Hat Ansible Automation Platform. Ansible Automation Platform provides a supported version of the Ansible Core toolset plus additional certified and supported content, tools, components, and cloud services. Customers with a valid subscription can use its RPM repository, install the additional tools, and consume certified content from the cloud services.



Important

This course uses Red Hat Ansible Automation Platform 2.2, which includes Ansible Core 2.13. This version of Ansible Automation Platform is roughly similar to the community Ansible 6 distribution, although Ansible Automation Platform includes different Ansible Content Collections and additional tools and components.

The course teaches you how to write and run Ansible automation code, and the skills you learn here help you with community Ansible as well.

Installing the server components of Ansible Automation Platform, such as automation controller and automation hub, is beyond the scope of this course.

Red Hat Ansible Automation Platform 2 Overview

Red Hat Ansible Automation Platform 2 includes a number of distinct components that together provide a complete and integrated set of automation tools and resources.

Ansible Core

Ansible Core provides the fundamental functionality used to run Ansible Playbooks. It defines the automation language that is used to write Ansible Playbooks in YAML text files. It provides the key functions such as loops, conditionals, and other Ansible imperatives needed for automation code. It also provides the framework and basic command-line tools to drive automation.

Red Hat Ansible Automation Platform 2.2 provides Ansible Core 2.13 in the `ansible-core` RPM package and in its `ee-minimal-rhel8` and `ee-supported-rhel8` automation execution environments.

Ansible Content Collections

Historically, Ansible provided a large number of modules as part of the core package; an approach referred to in the Ansible community as "batteries included". However, with the success and rapid growth of Ansible, the number of modules included with Ansible grew exponentially. This led to certain challenges with support, especially because users sometimes wanted to use earlier or later versions of modules than were packaged with a particular version of Ansible.

The upstream developers decided to reorganize most modules into separate *Ansible Content Collections* made up of related modules, roles, and plug-ins that are supported by the same group of developers. Ansible Core itself is limited to a small set of modules provided by the `ansible.builtin` Ansible Content Collection, which is always part of Ansible Core.

Red Hat provides access to more than 120 certified content collections with a Red Hat Ansible Automation Platform 2 subscription. Many community-supported collections are also available on Ansible Galaxy.

Automation Content Navigator

Red Hat Ansible Automation Platform 2 provides a new top-level tool to develop and test Ansible Playbooks, the *automation content navigator* (`ansible-navigator`). This tool replaces and extends the functionality of several command-line Ansible utilities, including `ansible-playbook`, `ansible-inventory`, `ansible-config`, and so on.

In addition, it separates the control node on which you run Ansible from the automation execution environment that runs it, by running your playbooks in a container. This makes it easier for you to provide a complete working environment for your automation code for deployment to production.

Automation Execution Environments

An *automation execution environment* is a container image that contains Ansible Core, Ansible Content Collections, and any Python libraries, executables, or other dependencies needed to run your playbook.

When you run a playbook with `ansible-navigator`, you can select an automation execution environment for it to use to run that playbook. When your code is working, you can provide the playbook and the automation execution environment to automation controller and know that it has everything needed to correctly run your playbook.

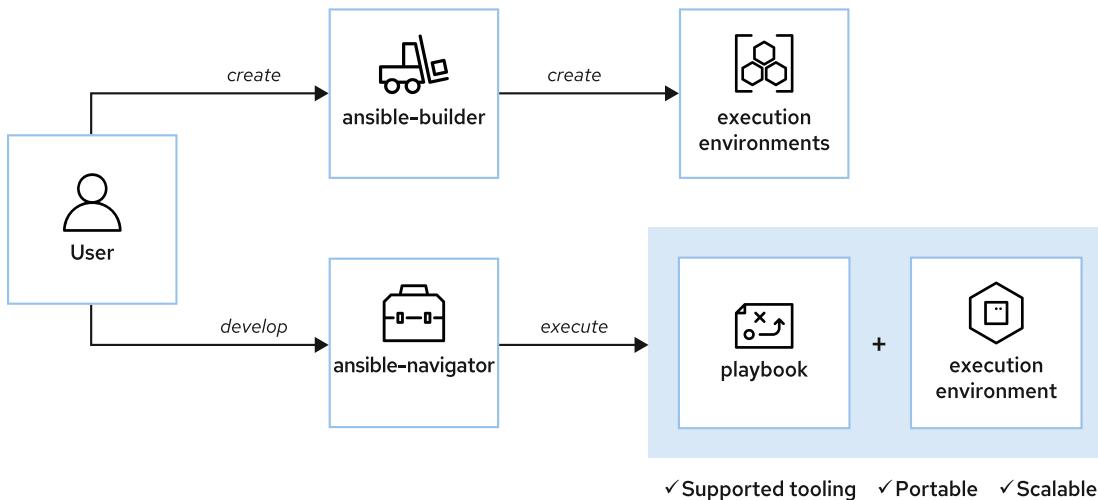


Figure 1.2: User experience: Adapting execution environments to your needs

Automation Controller

Automation controller, formerly called Red Hat Ansible Tower, is the component of Red Hat Ansible Automation Platform that provides a central point of control to run your enterprise automation code. It provides a web UI and a REST API that can be used to configure, run, and evaluate your automation jobs.

Automation Hub

A public service at `console.redhat.com` provides access to Red Hat Certified Ansible Content Collections that you can download and use with `ansible-galaxy` (for `ansible-navigator`) and with automation controller.

Preparing a Control Node

To run Ansible Playbooks, install automation content navigator (`ansible-navigator`) on your control node and download an execution environment. Hosts that are managed by Ansible do not

need to have `ansible-navigator` installed; you only need to install that tool on the control node from which you run Ansible Playbooks.

Python 3.8 or later needs to be installed on the control node before installing the `ansible-core` package.

You need a valid Red Hat Ansible Automation Platform subscription to install automation content navigator on your control node.

If you have activated Simple Content Access for your organization in the Red Hat Customer Portal, then you do not need to attach the subscription to your system.

The installation process is as follows:

**Note**

You do not need to run these exact steps in your classroom environment because it is preconfigured to download the `ee-supported-rhel8` execution environment.

- Install automation content navigator on your control nodes.

```
[user@controlnode ~]$ sudo dnf install ansible-navigator
```

- Verify that automation content navigator is installed on the system.

```
[user@controlnode ~]$ ansible-navigator --version
ansible-navigator 2.1.0
```

- Log in to the container registry.

```
[user@controlnode ~]$ podman login registry.redhat.io
Username: your-registry-username
Password: your-registry-password
Login Succeeded!
```

- Download the container image for the execution environment that you plan to use with automation content navigator. (Automation content navigator might also automatically download the default execution environment when you run the `ansible-navigator` command.)

```
[user@controlnode ~]$ podman pull \
> registry.redhat.io/ansible-automation-platform-22/ee-supported-rhel8:latest
```

- Display the list of locally available container images to verify that the image was downloaded.

```
[user@controlnode ~]$ ansible-navigator images
  Image           Tag     Execution environment      Created
  Size
  0|ee-supported-rhel8    latest    True                  5 weeks ago
  1.32 GB
```

**Note**

If you require access to the `ansible-playbook` command, which uses your control node as the execution environment (and does not use container-based execution environments), you can install the `ansible-core` package as well:

```
[user@controlnode ~]$ sudo dnf install ansible-core
```

However, `ansible-navigator` generally provides a better development experience and makes it easier for you to develop Ansible Playbooks that you can later migrate to automation controller for use by other members of your organization.

Preparing Managed Hosts

One of the benefits of Ansible is that managed hosts do not need to have a special agent installed. The Ansible control node connects to managed hosts by using a standard network protocol to ensure that the systems are in the specified state.

Managed hosts might have some requirements depending on how the control node connects to them and what modules are run on them.

- Linux and UNIX managed hosts need to have Python 3.8 or later installed for most modules to work. For Red Hat Enterprise Linux 8, you might be able to depend on the `platform-python` package. You can also enable and install the `python38` application stream.
- If SELinux is enabled on the managed hosts, ensure that the `python3-libselinux` package is installed before using modules that are related to any copy, file, or template functions. If the other Python components are installed, you can use Ansible modules such as `ansible.builtin.dnf` or `ansible.builtin.package` to ensure that this package is also installed.
- Ansible needs to be able to connect to the machine by using SSH, and if it connects as a regular user it needs to be able to use a mechanism such as `sudo` to get superuser access.

**Note**

Some modules might have their own additional requirements. For example, the `ansible.builtin.dnf` module, which can be used to install packages on current Fedora systems, requires the `python3-dnf` package.

Microsoft Windows Managed Hosts

The `ansible.windows` Ansible Content Collection that is part of the default automation execution environment includes a number of modules that are specifically designed for Microsoft Windows managed hosts.

Most of the modules specifically designed for Microsoft Windows managed hosts require PowerShell 3.0 or later on the managed host rather than Python. In addition, the managed hosts need to have Windows PowerShell remoting configured.

Ansible also requires .NET Framework 4.0 or later to be installed on Microsoft Windows managed hosts.

This course uses Linux-based managed hosts in its examples, and does not go into great depth on the specific differences and adjustments needed when managing Microsoft Windows managed hosts.

More information on managing Microsoft Windows managed hosts is available on the Ansible website at https://docs.ansible.com/ansible/latest/user_guide/windows.html, or in the Red Hat training course *Microsoft Windows Automation with Red Hat Ansible Automation Platform* (DO417).

Managed Network Devices

You can also use Ansible automation to configure managed network devices such as routers and switches. Ansible includes many modules specifically designed for this purpose. This includes support for Cisco IOS, IOS XR, and NX-OS; Juniper Junos; Arista EOS; and VyOS-based networking devices, among others.

You can write Ansible Playbooks for network devices using the same basic techniques that you use when writing playbooks for servers. Because most network devices cannot run Python, Ansible runs network modules on the control node, not on the managed hosts. Special connection methods are also used to communicate with network devices, typically using either CLI over SSH, XML over SSH, or API over HTTP(S).

This course does not cover the automation of network device management in any depth. For more information on this topic, see *Ansible for Network Automation* [<https://docs.ansible.com/ansible/latest/network/index.html>] on the Ansible community website, or the Red Hat training course *Network Automation with Red Hat Ansible Automation Platform* (DO457).



References

Simple Content Access

<https://access.redhat.com/articles/simple-content-access>

Red Hat Ansible Automation Platform Installation Guide Red Hat Ansible Automation Platform 2.2 | Red Hat Customer Portal

https://access.redhat.com/documentation/en-us/red_hat_ansible_automation_platform/2.2/html/red_hat_ansible_automation_platform_installation_guide/

Windows Guides – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/windows.html

Ansible for Network Automation – Ansible Documentation

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

► Guided Exercise

Installing Ansible

In this exercise, you install automation content navigator on a control node that runs Red Hat Enterprise Linux.

Outcomes

- You should be able to install automation content navigator on a control node.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start intro-install
```

Instructions

- 1. On the `workstation` machine, install the `ansible-navigator` RPM package that provides automation content navigator so that you can use that machine as your control node.

```
[student@workstation ~]$ sudo dnf install ansible-navigator
[sudo] password for student: student
Last metadata expiration check: 0:12:47 ago on Fri 29 Jul 2022 11:10:54 AM EDT.
Dependencies resolved.
...output omitted...
Is this ok [y/d/N]: y
...output omitted...
```



Note

This lab environment is already configured with the remote RPM package repository needed to install `ansible-navigator`. In a production setting, you would need to use `subscription-manager` to register your system with Red Hat Subscription Management and enable the `ansible-automation-platform-2.2-for-rhel-9-x86_64-rpms` repository first.

- 2. Verify that automation content navigator is installed on the system. Run the `ansible-navigator` command with the `--version` option.

```
[student@workstation ~]$ ansible-navigator --version
ansible-navigator 2.1.0
```

- 3. Log in to the container registry. Use the `podman login` command to log in to the automation hub registry at `utility.lab.example.com`. Use `admin` as the username and `redhat` as the password.

```
[student@workstation ~]$ podman login utility.lab.example.com
Username: admin
Password: redhat
Login Succeeded!
```

- 4. Download the execution environment container image. Run the `ansible-navigator images` command to make automation content navigator download the execution environment image and display a list of the available images.

```
[student@workstation ~]$ ansible-navigator images
...output omitted...
Running the command: podman pull utility.lab.example.com/ee-supported-rhel8:latest
...output omitted...
```

After the `ee-supported-rhel8:latest` image is downloaded, `ansible-navigator` displays the list of images in interactive mode:

Image	Tag	Execution environment	Created	Size
0 ee-supported-rhel8	latest	True	5 weeks ago	1.32 GB

```
^b/PgUp page up ^f/PgDn page down ↑ scroll esc back [0-9] goto :help help
```

Press Esc to exit the image list.

Finish

On the `workstation` machine, run the `lab finish intro-install` script to clean up this exercise.

```
[student@workstation ~]$ lab finish intro-install
```

This concludes the guided exercise.

Summary

- Automation helps you mitigate human error and ensure that your IT infrastructure is in a consistent, correct state.
- Ansible is an open source automation platform that can adapt to many workflows and environments.
- Red Hat Ansible Automation Platform is a fully supported version of Ansible that also includes a number of additional components and tools to help you develop, deploy, and manage your automation code.
- Ansible can be used to manage many types of systems, including servers running Linux, servers running Microsoft Windows, and network devices.
- Ansible Playbooks are human-readable text files that describe the desired state of an IT infrastructure.
- Ansible connects to managed hosts using standard network protocols such as SSH, and runs code or commands on the managed hosts to ensure that they are in the state specified.
- Ansible is built around an agentless architecture in which the Ansible software is only installed on a control node and in automation execution environments.
- Automation content navigator (`ansible-navigator`) is a key tool that helps you develop and run your Ansible automation code.

Chapter 2

Implementing an Ansible Playbook

Goal

Create an inventory of managed hosts, write a simple Ansible Playbook, and run the playbook to automate tasks on those hosts.

Objectives

- Describe Ansible inventory concepts and manage a static inventory file.
- Describe where Ansible configuration files are located, how Ansible selects them, and edit them to apply changes to default settings.
- Write a basic Ansible Playbook and run it using the automation content navigator.
- Write a playbook that uses multiple plays with per-play privilege escalation, and effectively use automation content navigator to find new modules in available Ansible Content Collections and use them to implement tasks for a play.

Sections

- Building an Ansible Inventory (and Guided Exercise)
- Managing Ansible Configuration Files (and Guided Exercise)
- Writing and Running Playbooks (and Guided Exercise)
- Implementing Multiple Plays (and Guided Exercise)

Lab

- Implementing an Ansible Playbook

Building an Ansible Inventory

Objectives

- Describe Ansible inventory concepts and manage a static inventory file.

Defining the Inventory

An *inventory* defines a collection of hosts that Ansible manages. These hosts can also be assigned to *groups*, which can be managed collectively. Groups can contain child groups, and hosts can be members of multiple groups. The inventory can also set variables that apply to the hosts and groups that it defines.

There are two ways to define host inventories. Use a text file to define a *static* host inventory. Use an Ansible plug-in to generate a *dynamic* host inventory as needed, using external information providers.

Specifying Managed Hosts with a Static Inventory

A static inventory file is a text file that specifies the managed hosts that Ansible targets. You can write this file using a number of different formats, including INI-style or YAML. The INI-style format is very common and is used for most examples in this course.



Note

Ansible supports multiple static inventory formats. This section focuses on the most common one, the INI-style format.

In its simplest form, an INI-style static inventory file is a list of hostnames or IP addresses of managed hosts, each on a single line:

```
web1.example.com
web2.example.com
db1.example.com
db2.example.com
192.0.2.42
```

Normally, however, you organize managed hosts into *host groups*. Host groups allow you to more effectively run Ansible against a collection of systems. In this case, each section starts with a host group name enclosed in square brackets ([]). This is followed by the hostname or an IP address for each managed host in the group, each on a single line.

In the following example, the host inventory defines two host groups: `webservers` and `db-servers`.

```
[webservers]
web1.example.com
web2.example.com
192.0.2.42
```

```
[db-servers]
db1.example.com
db2.example.com
```

Hosts can be in multiple groups. In fact, the recommended practice is to organize your hosts into multiple groups, possibly organized in different ways depending on the role of the host, its physical location, whether it is in production or not, and so on. This allows you to easily apply Ansible plays to specific sets of hosts based on their characteristics, purpose, or location.

```
[webservers]
web1.example.com
web2.example.com
192.0.2.42
```

```
[db-servers]
db1.example.com
db2.example.com
```

```
[east-datacenter]
web1.example.com
db1.example.com
```

```
[west-datacenter]
web2.example.com
db2.example.com
```

```
[production]
web1.example.com
web2.example.com
db1.example.com
db2.example.com
```

```
[development]
192.0.2.42
```



Important

Two host groups always exist:

- The `all` host group contains every host explicitly listed in the inventory.
- The `ungrouped` host group contains every host explicitly listed in the inventory that is not a member of any other group.

Defining Nested Groups

Ansible host inventories can include groups of host groups. This is accomplished by creating a host group name with the :children suffix. The following example creates a new group called `north-america`, which includes all hosts from `usa` and `canada` groups.

```
[usa]
washington1.example.com
washington2.example.com

[canada]
ontario01.example.com
ontario02.example.com

[north-america:children]
canada
usa
```

A group can have both managed hosts and child groups as members. For example, in the previous inventory you could add a `[north-america]` section that has its own list of managed hosts. That list of hosts would be merged with the additional hosts that the `north-america` group inherits from its child groups.

Simplifying Host Specifications with Ranges

You can specify ranges in the hostnames or IP addresses to simplify Ansible host inventories. You can specify either numeric or alphabetic ranges. Ranges have the following syntax:

```
[START:END]
```

Ranges match all values from `START` to `END`, inclusively. Consider the following examples:

- `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).
- `server[01:20].example.com` matches all hosts named `server01.example.com` through `server20.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`.

If leading zeros are included in numeric ranges, they are used in the pattern. The second example above does not match `server1.example.com` but does match `server07.example.com`.

To illustrate this, the following example uses ranges to simplify the `[usa]` and `[canada]` group definitions from the preceding example:

```
[usa]
washington[1:2].example.com

[canada]
ontario[01:02].example.com
```

Verifying the Inventory

When in doubt, use the `ansible-navigator inventory` command to verify a machine's presence in the inventory. In the following example, `ansible-navigator inventory` is run in stdout mode. The first query matches a host in the inventory, and the second does not.

```
[user@controlnode ~]$ ansible-navigator inventory -m stdout \
> --host washington1.example.com
{}

[user@controlnode ~]$ ansible-navigator inventory -m stdout \
> --host washington01.example.com
[WARNING]: Could not match supplied host pattern, ignoring:
washington01.example.com
...output omitted...
```

The following command lists all hosts in the inventory.

```
[user@controlnode ~]$ ansible-navigator inventory -m stdout --list
{
  "_meta": {
    "hostvars": {}
  },
  "all": {
    "children": [
      "canada",
      "ungrouped",
      "usa"
    ]
  },
  "canada": {
    "hosts": [
      "ontario01.example.com",
      "ontario02.example.com"
    ]
  },
  "usa": {
    "hosts": [
      "washington1.example.com",
      "washington2.example.com"
    ]
  }
}
```

The following command lists all hosts in a group.

```
[user@controlnode ~]$ ansible-navigator inventory -m stdout --graph canada
@canada:
|--ontario01.example.com
|--ontario02.example.com
```

Run the `ansible-navigator inventory` command to interactively browse inventory hosts and groups:

```
[user@controlnode ~]$ ansible-navigator inventory
  Title          Description
0|Browse groups   Explore each inventory group and group members members
1|Browse hosts     Explore the inventory with a list of all hosts
```

Type :0 to select "Browse Groups":

Name	Taxonomy	Type
0 canada	all	group
1 ungrouped	all	group
2 usa	all	group

Press the ESC key to exit the Groups menu. Type :1 to select "Browse Hosts"

```
Inventory hostname
0|ontario01.example.com
1|ontario02.example.com
2|washington1.example.com
3|washington2.example.com
```

Press the ESC key twice to exit `ansible-navigator inventory`.



Important

If the inventory contains a host and a host group with the same name, the `ansible-navigator inventory` command prints a warning.

Ensure that host groups do not use the same names as hosts in the inventory.

Overriding the Location of the Inventory

The `/etc/ansible/hosts` file is considered the system's default static inventory file. However, normal practice is not to use that file but to specify a different location for your inventory files.

The `ansible-navigator` commands that you use to run playbooks can specify the location of an inventory file on the command line with the `--inventory PATHNAME` or `-i PATHNAME` option, where `PATHNAME` is the path to the desired inventory file.



Note

You can also define a different default location for the inventory file in your Ansible configuration file.

Dynamic Inventories

Ansible inventory information can also be dynamically generated, using information provided by external databases. The open source community has written a number of dynamic inventory plug-ins that are available from the upstream Ansible project. If those Ansible plug-ins do not meet your needs, you can also write your own.

For example, a dynamic inventory program could contact your Red Hat Satellite server or Amazon EC2 account, and use information stored there to construct an Ansible inventory. Because the

program does this when you run Ansible, it can populate the inventory with up-to-date information provided by the service as new hosts are added, and old hosts are removed.

How to use a dynamic inventory is beyond the scope of this section.



References

How to build your inventory: Ansible Documentation

http://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html

► Guided Exercise

Building an Ansible Inventory

In this exercise, you create a new static inventory containing hosts and groups.

Outcomes

- You should be able to create default and custom static inventories.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start playbook-inventory
```

Instructions

- 1. Change into the `/home/student/playbook-inventory/` directory.

```
[student@workstation ~]$ cd playbook-inventory
[student@workstation playbook-inventory]$
```

- 2. Create a custom static inventory file named `inventory` in the `/home/student/playbook-inventory` working directory.

Information about your four managed hosts is listed in the following table. Assign each host to multiple groups for management purposes based on the purpose of the host, the city where it is located, and the deployment environment to which it belongs.

In addition, groups for US cities (Raleigh and Mountain View) must be set up as children of the `us` group so that hosts in the United States can be managed as a group.

Server Inventory Specifications

Host name	Purpose	Location	Environment
<code>servera.lab.example.com</code>	Web server	Raleigh	Development
<code>serverb.lab.example.com</code>	Web server	Raleigh	Testing
<code>serverc.lab.example.com</code>	Web server	Mountain View	Production
<code>serverd.lab.example.com</code>	Web server	London	Production

- 2.1. Create an inventory file in the /home/student/playbook-inventory working directory. Using the Server Inventory Specifications table as a guide, edit the inventory file so that it contains the following content:

```
[webservers]
server[a:d].lab.example.com

[raleigh]
servera.lab.example.com
serverb.lab.example.com

[mountainview]
serverc.lab.example.com

[london]
serverd.lab.example.com

[development]
servera.lab.example.com

[testing]
serverb.lab.example.com

[production]
serverc.lab.example.com
serverd.lab.example.com

[us:children]
raleigh
mountainview
```

- 3. Use variations of the `ansible-navigator inventory` command to verify the managed hosts and groups in the custom /home/student/playbook-inventory/inventory inventory file.



Important

Your `ansible-navigator inventory` command must include the `-i` option to specify the location of your inventory file, as shown in the following steps.

- 3.1. List all managed hosts in the inventory by using the `ansible-navigator inventory -i inventory -m stdout --list` command.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory -m stdout --list
{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": [
            "development",
            "london",
```

```
"production",
"testing",
"ungrouped",
"us",
"webservers"
]
},
"development": {
  "hosts": [
    "servera.lab.example.com"
  ]
},
"london": {
  "hosts": [
    "serverd.lab.example.com"
  ]
},
"mountainview": {
  "hosts": [
    "serverc.lab.example.com"
  ]
},
"production": {
  "hosts": [
    "serverc.lab.example.com",
    "serverd.lab.example.com"
  ]
},
"raleigh": {
  "hosts": [
    "servera.lab.example.com",
    "serverb.lab.example.com"
  ]
},
"testing": {
  "hosts": [
    "serverb.lab.example.com"
  ]
},
"us": {
  "children": [
    "mountainview",
    "raleigh"
  ]
},
"webservers": {
  "hosts": [
    "servera.lab.example.com",
    "serverb.lab.example.com",
    "serverc.lab.example.com",
    "serverd.lab.example.com"
  ]
}
}
```

- 3.2. Graph all managed hosts in the inventory file that are not part of a group by running the `ansible-navigator inventory -i inventory -m stdout --graph ungrouped` command. No ungrouped managed hosts exist in this inventory file.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory -m stdout --graph ungrouped
@ungrouped:
```

- 3.3. Graph all managed hosts in the `development` group by using the `ansible-navigator inventory -i inventory -m stdout --graph development` command.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory -m stdout --graph development
@development:
|--servera.lab.example.com
```

- 3.4. Graph all managed hosts in the `testing` group by using the `ansible-navigator inventory -i inventory -m stdout --graph testing` command.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory -m stdout --graph testing
@testing:
|--serverb.lab.example.com
```

- 3.5. Graph all managed hosts in the `production` group by using the `ansible-navigator inventory -i inventory -m stdout --graph production` command.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory -m stdout --graph production
@production:
|--serverc.lab.example.com
|--serverd.lab.example.com
```

- 3.6. Graph all managed hosts in the `us` group by using the `ansible-navigator inventory -i inventory -m stdout --graph us` command.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory -m stdout --graph us
@us:
|--@mountainview:
|   |--serverc.lab.example.com
|--@raleigh:
|   |--servera.lab.example.com
|   |--serverb.lab.example.com
```

- 3.7. Run `ansible-navigator inventory -i inventory` in interactive mode. Browse groups and managed host entries in the inventory file.

Type :0 to browse groups. Type :1 to browse hosts. Type :q to exit `ansible-navigator`.

```
[student@workstation playbook-inventory]$ ansible-navigator inventory \
> -i inventory

  Title          Description
0|Browse groups   Explore each inventory group and group members members
1|Browse hosts     Explore the inventory with a list of all hosts
```

Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish playbook-inventory
```

This concludes the section.

Managing Ansible Configuration Files

Objectives

- Describe where Ansible configuration files are located, how Ansible selects them, and edit them to apply changes to default settings.

Configuring Ansible

You can create and edit two files in each of your Ansible project directories that configure the behavior of Ansible and the `ansible-navigator` command:

- `ansible.cfg`, which configures the behavior of several Ansible tools.
- `ansible-navigator.yml`, which changes default options for the `ansible-navigator` command.

Managing Ansible Settings

You can create an `ansible.cfg` file in your Ansible project's directory to apply settings that apply to multiple Ansible tools.

The Ansible configuration file consists of several sections, with each section containing settings defined as key-value pairs. Section titles are enclosed in square brackets. For basic operation, use the following two sections:

- [`defaults`], which sets defaults for Ansible operation
- [`privilege_escalation`], which configures how Ansible performs privilege escalation on managed hosts

For example, the following is a typical `ansible.cfg` file:

```
[defaults]
inventory = ./inventory
remote_user = user
ask_pass = false

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = false
```

The following table explains these parameters:

Directive	Description
<code>inventory</code>	Specifies the path to the inventory file.

Directive	Description
remote_user	Specifies the username that Ansible uses to connect to the managed hosts. If not specified, the current user's name is used. (In a container-based automation execution environment run by <code>ansible-navigator</code> , this is always <code>root</code> .)
ask_pass	Specifies whether to prompt for an SSH password. (Can be <code>false</code> , which is the default, if using SSH public key authentication.)
become	Specifies whether to automatically switch users on the managed host (typically to <code>root</code>) after connecting. This can also be specified by a play.
become_method	Specifies how to switch users (typically <code>sudo</code> , which is the default, but <code>su</code> is an option).
become_user	Specifies which user to switch to on the managed host (typically <code>root</code> , which is the default).
become_ask_pass	Specifies whether to prompt for a password for the <code>become_method</code> parameter. Defaults to <code>false</code> .

Determining Current Configuration Settings

The `ansible-navigator config` command displays the current Ansible configuration. The command displays the actual value that Ansible uses for each parameter and from which source it retrieves that value, configuration file, or environment variable.

The following output is from the `ansible-navigator config` command run from a `/home/student/project/` directory that contains an `ansible.cfg` file:

Name	Default	Source	Current
<i>...output omitted...</i>			
44 Default ask pass	True	default	False
45 Default ask vault pass	True	default	False
46 Default become	False	/home/student/project/ansible.cfg	True
47 Default become ask pass	False	/home/student/project/ansible.cfg	True
<i>...output omitted...</i>			
50 Default become method	False	/home/student/project/ansible.cfg sudo	
51 Default become user	False	/home/student/project/ansible.cfg root	
<i>...output omitted...</i>			

In the preceding example, each line describes an Ansible configuration parameter.

- The `Default ask pass` and `Default ask vault pass` parameters use their default values, indicated by the `True` value in the `Default` column.
- The `Default become` and `Default become ask pass` parameters have been manually configured to `True` in the `/home/student/project/ansible.cfg` configuration file. The `Default` column is `False` for these two parameters. The `Source` column provides the path to the configuration file which defines these parameters, and the `Current` column shows that the value for these two parameters is `True`.

- The `Default become method` parameter has the current value of `sudo`, and the `Default become user` parameter has the current value of `root`.

To exit the interactive mode of the `ansible-navigator config` command, press Esc or type :q.



Note

If the project does not include an `ansible.cfg` file, then Ansible tries to use a `~/.ansible.cfg` file in the home directory of the user running Ansible, and if that does not exist, it tries to use an `/etc/ansible/ansible.cfg` file.

However, if you are using `ansible-navigator`, the `ansible-navigator` command looks for these files *inside the automation execution environment*. On the current Red Hat-supported execution environments, these files do not exist or are empty.

If you are using the earlier `ansible-playbook` command or other Ansible commands that do not use automation execution environments, then Ansible looks for these files on your workstation.

Managing Settings for Automation Content Navigator

You can create a configuration file (or settings file) for `ansible-navigator` to override the default values of its configuration settings. The settings file can be in JSON (`.json`) or YAML (`.yml` or `.yaml`) format. This discussion focuses on the YAML format.

Automation content navigator looks for a settings file in the following order and uses the first file that it finds:

- If the `ANSIBLE_NAVIGATOR_CONFIG` environment variable is set, then use the configuration file at the location it specifies.
- An `ansible-navigator.yml` file in your current Ansible project directory.
- A `~/.ansible-navigator.yml` file (in your home directory). Notice that it has a "dot" at the start of its file name.

Just like the Ansible configuration file, each project can have its own automation content navigator settings file.

The following `ansible-navigator.yml` file configures some common settings:

```
ansible-navigator:  
  execution-environment: ①  
    image: utility.lab.example.com/ee-supported-rhel8:latest ②  
    pull:  
      policy: missing ③  
  playbook-artifact:  
    enable: false ④
```

- ① The `execution-environment` section configures settings for the execution environment that the `ansible-navigator` command uses.
- ② The `image` section defines the container image name to use for the execution environment.

- ③ The `policy` section nested below the `pull` section states to only pull the container image if it does not already exist on the local machine.
- ④ The `enable` section nested below the `playbook-artifact` section disables generating playbook artifacts when using the `ansible-navigator run` command. Playbook artifacts must be disabled when you require a prompt for a password when running a playbook.

**Note**

More complex configurations for automation content navigator are covered in the course *Developing Advanced Automation with Red Hat Ansible Automation Platform (DO374)*. See <https://ansible-navigator.readthedocs.io/en/latest/settings/> for more documentation on the settings that you can use in this file.

Configuring Connections

Ansible needs to know how to communicate with its managed hosts. One of the most common reasons to change the configuration file is to control which methods and users Ansible uses to administer managed hosts. The following are some examples of required information:

- The location of the inventory that lists the managed hosts and host groups
- The connection protocol to use to communicate with the managed hosts (by default, SSH), and whether a nonstandard network port is needed to connect to the server
- The remote user to use on the managed hosts; this could be `root` or it could be an unprivileged user
- If the remote user is unprivileged, Ansible needs to know if it should try to escalate privileges to `root` and how to do it (for example, by using `sudo`)
- Whether to prompt for an SSH password or `sudo` password to log in or gain privileges

Inventory Location

In the `[defaults]` section of the `ansible.cfg` file, the `inventory` parameter can point directly to a static inventory file, or to a directory containing multiple static inventory files and dynamic inventory scripts.

```
[defaults]
inventory = ./inventory
```

Connection Settings

By default, Ansible connects to managed hosts using the SSH protocol. The most important parameters that control how Ansible connects to the managed hosts are set in the `[defaults]` section.

If not configured otherwise, Ansible attempts to connect to the managed host using the same username as the local user running the Ansible commands. To specify a different remote user, set the `remote_user` parameter to that username. 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 you do not have SSH key-based authentication configured for your remote user, it is possible to use password-based authentication. However, to use password-based SSH authentication with

`ansible-navigator`, you need to configure `ansible-navigator` so that it does not generate playbook artifacts (log files that record information about playbook runs).

The following example is a minimal `ansible-navigator.yml` file that disables the generation of playbook artifacts:

```
ansible-navigator:  
  playbook-artifact:  
    enable: false
```

With this configuration, you can run `ansible-navigator run` with the `-m stdout` and `--ask-pass` options in addition to the playbook that you want to run, so that the `ansible-navigator` command prompts you for the remote user's SSH password.



Important

You can set the `ask_pass = true` parameter in the defaults section of the `ansible.cfg` file so that you are always prompted by Ansible tools for the SSH password.

If you do not disable playbook artifact generation for `ansible-navigator`, or if you do not use `-m stdout` when running `ansible-navigator`, the `ansible-navigator` command might hang or otherwise fail to run.

If you have password-based SSH authentication set up, you can configure key-based SSH authentication.

The first step is to make sure that the user on the control node has an SSH key pair configured in `~/.ssh`. You can run the `ssh-keygen` command to generate a key pair.

For a single existing managed host, you can install your public key on the managed host and use the `ssh-copy-id` command to populate your local `~/.ssh/known_hosts` file with its host key, as follows:

```
[user@controlnode ~]$ ssh-copy-id root@web1.example.com  
The authenticity of host 'web1.example.com (192.168.122.181)' can't be  
established.  
ECDSA key fingerprint is 70:9c:03:cd:de:ba:2f:11:98:fa:a0:b3:7c:40:86:4b.  
Are you sure you want to continue connecting (yes/no)? yes  
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter  
out any that are already installed  
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted  
now it is to install the new keys  
root@web1.example.com's password:  
  
Number of key(s) added: 1  
  
Now try logging into the machine, with: "ssh 'root@web1.example.com'"  
and check to make sure that only the key(s) you wanted were added.
```

**Note**

You can also use an Ansible Playbook to deploy your public key to the `remote_user` account on *all* managed hosts using the `authorized_key` module.

This course has not covered Ansible Playbooks in detail yet. A play that ensures that your public key is deployed to the managed hosts' `root` accounts might read as follows:

```
- name: Public key is deployed to managed hosts for Ansible
  hosts: all

  tasks:
    - name: Ensure key is in root's ~/.ssh/authorized_hosts
      ansible.posix.authorized_key:
        user: root
        state: present
        key: '{{ item }}'
      with_file:
        - ~/ssh/id_rsa.pub
```

Because the managed host would not have SSH key-based authentication configured yet, you would have to run the playbook using the `ansible-navigator run` command with the `--ask-pass` option for the command to authenticate as the remote user.

Escalating Privileges

For security and auditing reasons, Ansible might need to connect to remote hosts as an unprivileged user before escalating privileges to get administrative access as the `root` user. This can be set up in the `[privilege_escalation]` section of the Ansible configuration file.

To enable privilege escalation by default, set the `become = true` parameter in the configuration file. Even if this is set by default, various methods exist to override it when running ad hoc commands or Ansible Playbooks. (For example, there might be times when you want to run a task or play that does not escalate privileges.)

The `become_method` parameter specifies how to escalate privileges. Several options are available, but the default is to use `sudo`. Likewise, the `become_user` parameter specifies which user to escalate to, but the default is `root`.

If the `become_method` mechanism chosen requires the user to enter a password to escalate privileges, you can set the `become_ask_pass = true` parameter in the configuration file.

**Important**

If you have `become_ask_pass = true` set when you use `ansible-navigator`, you also need to disable playbook artifact generation and use `-m stdout` as previously discussed in this section.

**Note**

On Red Hat Enterprise Linux 8 and 9, the default configuration of `/etc/sudoers` grants all users in the `wheel` group the ability to use `sudo` to become `root` after entering their password.

One way to enable a user (`someuser` in the following example) to use `sudo` to become `root` without a password is to install a file with the appropriate parameters into the `/etc/sudoers.d` directory (owned by `root`, with octal permissions `0400`):

```
## password-less sudo for Ansible user
someuser ALL=(ALL) NOPASSWD:ALL
```

Think through the security implications of whatever approach you choose for privilege escalation. Different organizations and deployments might have different tradeoffs to consider.

The following example `ansible.cfg` file assumes that you can connect to the managed hosts as `someuser` using SSH key-based authentication, and that `someuser` can use `sudo` to run commands as `root` without entering a password:

```
[defaults]
inventory = ./inventory
remote_user = someuser
ask_pass = false

[privilegeEscalation]
become = true
becomeMethod = sudo
becomeUser = root
becomeAskPass = false
```

**Note**

Setting `become = true` means that all tasks in the plays in playbooks that you run are run as the user specified by the `become_user` parameter, usually the `root` user. From a security perspective, rather than setting `become = true` in the `ansible.cfg` file, it might be a better practice to configure privilege escalation for specific plays in your playbooks, or on a task-by-task basis as needed.

Configuration File Comments

Both the `ansible.cfg` file and `ansible-navigator.yml` support the number sign (#) at the start of a line as a comment character. The number sign at the start of a line comments out the entire line.

In addition, the `ansible.cfg` file supports the semicolon (;) as a comment character. The semicolon character comments out everything to the right of it on the line.



References

ssh-keygen(1), and ssh-copy-id(1) man pages

Configuring Ansible: Ansible Documentation

https://docs.ansible.com/ansible/latest/installation_guide/intro_configuration.html

ansible-navigator settings: Ansible Navigator Documentation

<https://ansible-navigator.readthedocs.io/en/latest/settings/>

► Guided Exercise

Managing Ansible Configuration Files

In this exercise, you edit Ansible configuration files to customize your Ansible environment.

Outcomes

- You should be able to create configuration files to configure your Ansible environment with persistent custom settings.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start playbook-manage
```

Instructions

- 1. Change into the `/home/student/playbook-manage` directory.

```
[student@workstation ~]$ cd ~/playbook-manage  
[student@workstation playbook-manage]$
```

- 2. Configure automation content navigator.

- 2.1. Create the `/home/student/playbook-manage/ansible-navigator.yml` file. Configure automation content navigator to use the execution environment image `utility.lab.example.com/ee-supported-rhel8:latest` and to only pull the image if it is missing. Also configure automation content navigator to disable playbook artifacts. The file should consist of the following content:

```
---  
ansible-navigator:  
  execution-environment:  
    image: utility.lab.example.com/ee-supported-rhel8:latest  
    pull:  
      policy: missing  
  playbook-artifact:  
    enable: false
```

- 2.2. Run the `ansible-navigator images` command to list the available execution environment images.

```
[student@workstation playbook-manage]$ ansible-navigator images
-----
Execution environment image and pull policy overview
-----
Execution environment image name: utility.lab.example.com/ee-supported-rhel8:latest
Execution environment image tag: latest
Execution environment pull arguments: None
Execution environment pull policy: missing
Execution environment pull needed: True
-----
Updating the execution environment
...output omitted...
Running the command: podman pull utility.lab.example.com/ee-supported-rhel8:latest
Trying to pull utility.lab.example.com/ee-supported-rhel8:latest...
...output omitted...
```

- 2.3. After automation content navigator pulls the execution environment image you should see it in the list:

Image	Tag	Execution environment	Created	Size
0 ee-supported-rhel8	latest	True	3 weeks ago	1.34 GB

^b/PgUp page up ^f/PgDn page down ↑ scroll esc back [0-9] goto :help help

Press Esc to exit the image list.

- 3. In your /home/student/playbook-manage directory, start editing a new file named `ansible.cfg`. Create a `[defaults]` section in that file. In that section, add a line that uses the `inventory` directive to specify the `./inventory` file as the default inventory.

```
[defaults]
inventory = ./inventory
```

Save your work and exit the text editor.

- 4. In the /home/student/playbook-manage directory, start editing the new static inventory file, `inventory`.

The static inventory should contain four host groups:

- `[myself]` should contain the `workstation` host.
- `[intranetweb]` should contain the `servera.lab.example.com` host.
- `[internetweb]` should contain the `serverb.lab.example.com` host.
- `[web]` must contain the `intranetweb` and `internetweb` host groups.

- 4.1. In /home/student/playbook-manage/inventory, create the `myself` host group by adding the following lines:

```
[myself]
workstation
```

- 4.2. In `/home/student/playbook-manage/inventory`, create the `intranetweb` host group by adding the following lines:

```
[intranetweb]  
servera.lab.example.com
```

- 4.3. In `/home/student/playbook-manage/inventory`, create the `internetweb` host group by adding the following lines:

```
[internetweb]  
serverb.lab.example.com
```

- 4.4. In `/home/student/playbook-manage/inventory`, create the `web` host group by adding the following lines:

```
[web:children]  
intranetweb  
internetweb
```

- 4.5. The final `inventory` file should consist of the following content:

```
[myself]  
workstation  
  
[intranetweb]  
servera.lab.example.com  
  
[internetweb]  
serverb.lab.example.com  
  
[web:children]  
intranetweb  
internetweb
```

Save your work and exit the text editor.

- 5. Use the `ansible-navigator` command to run the provided playbooks and test the configuration of your inventory file's host groups.

The `ansible-navigator run` command runs an Ansible Playbook, formatted as a YAML file, that contains automation instructions to be run on managed hosts. The following `ansible-navigator` commands use the configuration files that you edited in preceding steps.

Each of the following playbooks runs the `ansible.builtin.ping` module on a host or group of hosts to determine if they are ready to be used as managed hosts by Ansible. These tests also validate whether your `inventory` file is correct.

- 5.1. Run the `/home/student/playbook-manage/ping-myself.yml` playbook to verify that the `workstation` machine is in the `myself` inventory group.

```
[student@workstation playbook-manage]$ ansible-navigator run \  
> -m stdout ping-myself.yml
```

```
PLAY [Validate inventory hosts] ****
TASK [Ping workstation] ****
ok: [workstation]

PLAY RECAP ****
workstation : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

- 5.2. Run the /home/student/playbook-manage/ping-intranetweb.yml playbook to verify that the servera.lab.example.com machine is in the intranetweb inventory group.

```
[student@workstation playbook-manage]$ ansible-navigator run \
> -m stdout ping-intranetweb.yml

PLAY [Validate inventory hosts] ****
TASK [Ping intranetweb] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

- 5.3. Run the /home/student/playbook-manage/ping-internetweb.yml playbook to verify that the serverb.lab.example.com machine is in the internetweb inventory group.

```
[student@workstation playbook-manage]$ ansible-navigator run \
> -m stdout ping-internetweb.yml

PLAY [Validate inventory hosts] ****
TASK [Ping internetweb] ****
ok: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

- 5.4. Run the /home/student/playbook-manage/ping-web.yml playbook to verify that the servera.lab.example.com and serverb.lab.example.com machines are in the web inventory group.

```
[student@workstation playbook-manage]$ ansible-navigator run \
> -m stdout ping-web.yml

PLAY [Validate inventory hosts] ****
TASK [Ping web] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]
```

```
PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
serverb.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

- 5.5. Run the /home/student/playbook-manage/ping-all.yml playbook to verify that the workstation, servera.lab.example.com, and serverb.lab.example.com machines are all in the inventory file.

```
[student@workstation playbook-manage]$ ansible-navigator run \
> -m stdout ping-all.yml

PLAY [Validate inventory hosts] ****

TASK [Ping all] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]
ok: [workstation]

PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
serverb.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
workstation      : ok=1    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

- ▶ 6. Open the /home/student/playbook-manage/ansible.cfg file in a text editor. Add a [privilege_escalation] section to configure Ansible to automatically use the sudo command to switch from student to root when running tasks on the managed hosts. Ansible should also be configured to prompt you for the password that student uses for the sudo command.
- 6.1. Create the [privilege_escalation] section in the /home/student/playbook-manage/ansible.cfg configuration file by adding the following entry:

```
[privilege_escalation]
```

- 6.2. Enable privilege escalation by setting the become directive to true.

```
become = true
```

- 6.3. Set the privilege escalation to use the sudo command by setting the become_method directive to sudo.

```
become_method = sudo
```

- 6.4. Set the privilege escalation user by setting the become_user directive to root.

```
become_user = root
```

- 6.5. Enable prompting for the privilege escalation password by setting the `become_ask_pass` directive to `true`.

```
become_ask_pass = true
```

- 6.6. The complete `ansible.cfg` file should consist of the following content:

```
[defaults]
inventory = ./inventory

[privilegeEscalation]
become = true
becomeMethod = sudo
becomeUser = root
becomeAskPass = true
```

Save your work and exit the text editor.

- 7. Use the `ansible-navigator` command to run the `/home/student/playbook-manage/ping-intranetweb.yml` playbook again to verify that you are now prompted for the sudo password.

When prompted for the sudo password, enter `student`.

```
[student@workstation playbook-manage]$ ansible-navigator run \
> -m stdout ping-intranetweb.yml
BECOME password: student

PLAY [Validate inventory hosts] ****
TASK [Ping intranetweb] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish playbook-manage
```

This concludes the section.

Writing and Running Playbooks

Objectives

- Write a basic Ansible Playbook and run it using the automation content navigator.

Ansible Playbooks

The power of Ansible is that you can use playbooks to run multiple, complex tasks against a set of targeted hosts in an easily repeatable manner.

A *task* is the application of a module to perform a specific unit of work. A *play* is a sequence of tasks to be applied, in order, to one or more hosts selected from your inventory. A *playbook* is a text file containing a list of one or more plays to run in a specific order.

Plays enable you to change a lengthy, complex set of manual administrative tasks into an easily repeatable routine with predictable and successful outcomes. In a playbook, you can save the sequence of tasks in a play into a human-readable and immediately runnable form. The tasks themselves, because of the way in which they are written, document the steps needed to deploy your application or infrastructure.

Formatting an Ansible Playbook

The following example contains one play with a single task.

```
---
- name: Configure important user consistently
  hosts: servera.lab.example.com
  tasks:
    - name: Newbie exists with UID 4000
      ansible.builtin.user:
        name: newbie
        uid: 4000
        state: present
```

A playbook is a text file written in YAML format, and is normally saved with the extension `.yml`. The playbook uses indentation with space characters to indicate the structure of its data. YAML does not place strict requirements on how many spaces are used for the indentation, but two basic rules apply:

- Data elements at the same level in the hierarchy (such as items in the same list) must have the same indentation.
- Items that are children of another item must be indented more than their parents.

You can also add blank lines for readability.

**Important**

You can only use space characters for indentation; do not use tab characters.

If you use the `vi` text editor, you can apply some settings which might make it easier to edit your playbooks. For example, you can add the following line to your `$HOME/.vimrc` file, and when `vi` detects that you are editing a YAML file, it performs a 2-space indentation when you press the Tab key and automatically indents subsequent lines.

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

A playbook usually begins with a line consisting of three dashes (---) to indicate the start of the document. It might end with three dots (...) to indicate the end of the document, although in practice this is often omitted.

Between those markers, the playbook is defined as a list of plays. An item in a YAML list starts with a single dash followed by a space. For example, a YAML list might appear as follows:

```
- apple
- orange
- grape
```

In the preceding playbook example, the line after --- begins with a dash and starts the first (and only) play in the list of plays.

The play itself is a collection of key-value pairs. Keys in the same play should have the same indentation. The following example shows a YAML snippet with three keys. The first two keys have simple values. The third has a list of three items as a value.

```
name: just an example
hosts: webservers
tasks:
  - first
  - second
  - third
```

The initial example play has three keys: `name`, `hosts`, and `tasks`. These keys all have the same indentation.

The first line of the example play starts with a dash and a space (indicating the play is the first item of a list), and then the first key, `name`. The `name` key associates an arbitrary string with the play as a label that identifies the purpose of the play. The `name` key is optional, but is recommended because it helps to document your playbook. This is especially useful when a playbook contains multiple plays.

```
- name: Configure important user consistently
```

The second key in the play is a `hosts` key, which specifies the hosts against which the play's tasks are run. The `hosts` key takes a host pattern as a value, such as the names of managed hosts or groups in the inventory.

```
hosts: servera.lab.example.com
```

Finally, the last key in the play is `tasks`, whose value specifies a list of tasks to run for this play. This example has a single task, which runs the `ansible.builtin.user` module with specific arguments (to ensure the `newbie` user exists and has UID 4000).

```
tasks:
  - name: newbie exists with UID 4000
    ansible.builtin.user:
      name: newbie
      uid: 4000
      state: present
```

The `tasks` key is the part of the play that actually lists, in order, the tasks to be run on the managed hosts. Each task in the list is itself a collection of key-value pairs.

In this example, the only task in the play has two keys:

- `name` is an optional label documenting the purpose of the task. It is a good idea to name all your tasks to help document the purpose of each step of the automation process.
- `ansible.builtin.user` is the module to run for this task. Its arguments are passed as a collection of key-value pairs, which are children of the module (`name`, `uid`, and `state`).

The following is another example of a `tasks` key with multiple tasks, each using the `ansible.builtin.service` module to ensure that a service should start at boot:

```
tasks:
  - name: Web server is enabled
    ansible.builtin.service:
      name: httpd
      enabled: true

  - name: NTP server is enabled
    ansible.builtin.service:
      name: chronyd
      enabled: true

  - name: Postfix is enabled
    ansible.builtin.service:
      name: postfix
      enabled: true
```



Important

The order in which the plays and tasks are listed in a playbook is important, because Ansible runs them in the same order.

Finding Modules for Tasks

Modules are the tools that plays use to accomplish tasks. Hundreds of modules have been written that do different things. You can usually find a tested, special-purpose module that does what you need, often as part of the default automation execution environment.

Ansible Core 2.11 and later package the modules that you use for tasks in sets called *Ansible Content Collections*. Each Ansible Content Collection contains a selection of related Ansible content, including modules and documentation.

The `ansible-core` package provides a single Ansible Content Collection named `ansible.builtin`. These modules are always available to you. Visit <https://docs.ansible.com/ansible/latest/collections/ansible/builtin/> for a list of modules contained in the `ansible.builtin` collection.

In addition, the default automation execution environment used by `ansible-navigator` in Red Hat Ansible Automation Platform 2.2, ee-rhel8-supported, includes a number of other Ansible Content Collections.

You can browse these collections by running the `ansible-navigator collections` command. In the interactive UI, you can type a colon (:) followed by the line number of a collection to get more information about it, including the list of modules and other Ansible content that it provides. You can do the same thing with the line number of a module to get documentation about that module. Press Esc to go back to the preceding list.



Note

You can also download additional Ansible Content Collections from a number of places, including:

- The automation hub offered through the Red Hat Hybrid Cloud Console at <https://content.redhat.com/ansible/automation-hub>
- A private automation hub managed by your organization
- The community's Ansible Galaxy website at <https://galaxy.ansible.com>

These can be installed in the `collections` directory of your Ansible project. Red Hat does not provide formal support for community Ansible Content Collections, only for Red Hat Certified Ansible Content Collections.

Modules are named using fully qualified collection names (FQCNs). This allows the same name to be used for different modules in two Ansible Content Collections without causing conflicts. For example, the `copy` module provided by the `ansible.builtin` Ansible Content Collection has `ansible.builtin.copy` as its FQCN.



Important

In earlier versions of Ansible, modules had to be included with Ansible and were named using just their short name, for example the `copy` module. Ansible might still try to resolve short names if your playbooks use them. However, to avoid errors, it is a good practice to use FQCNs in new playbooks.

Most modules are *idempotent*, which means that they can be run safely multiple times, and if the system is already in the correct state, they do nothing. For example, if you run the play from the preceding example a second time, it should report no changes.

Running Playbooks

The `ansible-navigator run` command is used to run playbooks. The command is executed on the control node, and the name of the playbook to be run is passed as an argument.

Running the `ansible-navigator run` command with the `-m stdout` option prints the output of the playbook run to standard output. If the `-m stdout` option is not provided, then `ansible-navigator` runs in interactive mode. Interactive mode is covered in the course *DO374: Developing Advanced Automation with Red Hat Ansible Automation Platform*.

```
[user@controlnode ~]$ ansible-navigator run \
> -m stdout site.yml
```

When you run the playbook, output is generated to show the play and tasks being executed. The output also reports the results of each task executed.

The following example shows the contents of a simple playbook, and then the result of running it.

```
[user@controlnode playdemo]$ cat webserver.yml
---
- name: Play to set up web server
  hosts: servera.lab.example.com
  tasks:
    - name: Latest httpd version installed
      ansible.builtin.dnf:
        name: httpd
        state: latest
    ...output omitted...
[user@controlnode playdemo]$ ansible-navigator run \
> -m stdout webserver.yml

PLAY [Play to set up web server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Latest httpd version installed] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

The value of the `name` key for each play and task is displayed when the playbook is run. (The `Gathering Facts` task is a special task that the `ansible.builtin.setup` module usually runs automatically at the start of a play. This is covered later in the course.) For playbooks with multiple plays and tasks, setting `name` attributes makes it easier to monitor the progress of a playbook's execution.

You should also see that the `Latest httpd version installed` task is `changed` for `servera.lab.example.com`. This means that the task changed something on that host to ensure that its specification was met. In this case, it means that the `httpd` package was not previously installed or was not the latest version.

In general, tasks in Ansible Playbooks are idempotent, and it is safe to run a playbook multiple times. If the targeted managed hosts are already in the correct state, no changes should be made. For example, assume that the playbook from the previous example is run again:

```
[user@controlnode playdemo]$ ansible-navigator run \
> -m stdout webserver.yml

PLAY [Play to set up web server] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Latest httpd version installed] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=2    changed=0    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

This time, all tasks passed with status ok and no changes were reported.



Note

Community Ansible provides an earlier tool called `ansible-playbook` that takes many of the same options as `ansible-navigator run -m stdout` and that uses your control node as the execution environment.

It cannot use automation execution environments, and is only supported by Red Hat in Red Hat Enterprise Linux 9 for narrow use cases.

Increasing Output Verbosity

The default output provided by the `ansible-navigator run` command does not provide detailed task execution information. The `-v` option provides additional information, with up to four levels.

Configuring the Output Verbosity of Playbook Execution

Option	Description
<code>-v</code>	Displays task results.
<code>-vv</code>	Displays task results and task configuration.
<code>-vvv</code>	Displays extra information about connections to managed hosts.
<code>-vvvv</code>	Adds extra verbosity options to the connection plug-ins, including users being used on the managed hosts to execute scripts, and what scripts have been executed.

Syntax Verification

Before executing a playbook, it is good practice to validate its syntax. You can use the `ansible-navigator run --syntax-check` command to validate the syntax of a playbook. The following example shows the successful syntax validation of a playbook.

```
[user@controlnode playdemo]$ ansible-navigator run \
> -m stdout webserver.yml --syntax-check
playbook: /home/user/playdemo/webserver.yml
```

When syntax validation fails, a syntax error is reported. The output also includes the approximate location of the syntax issue in the playbook. The following example shows the failed syntax validation of a playbook where the space separator is missing after the name attribute for the play.

```
[user@controlnode playdemo]$ ansible-navigator run \
> -m stdout webserver.yml --syntax-check

ERROR! Syntax Error while loading YAML.
      mapping values are not allowed in this context

The error appears to have been in ...output omitted... line 3, column 8, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

- name:Play to set up web server
  hosts: servera.lab.example.com
        ^ here
```

Executing a Dry Run

You can use the `--check` option to run a playbook in *check mode*, which performs a "dry run" of the playbook. This causes Ansible to report what changes would have occurred if the playbook were executed, but does not make any actual changes to managed hosts.

The following example shows the dry run of a playbook containing a single task for ensuring that the latest version of the `httpd` package is installed on a managed host. In this case, the dry run reports that the task would make a change on the managed host.

```
[user@controlnode playdemo]$ ansible-navigator run \
> -m stdout webserver.yml --check

PLAY [Play to set up web server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Latest httpd version installed] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=2      changed=1      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0
```



References

Intro to playbooks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

Working with playbooks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks.html

Validating tasks: check mode and diff mode – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_checkmode.html

► Guided Exercise

Writing and Running Playbooks

In this exercise, you write and run an Ansible Playbook.

Outcomes

- You should be able to write a playbook using basic YAML syntax and Ansible Playbook structure, and successfully run it with the `ansible-navigator run` command.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start playbook-basic
```

Instructions

The `/home/student/playbook-basic` Ansible project directory has been created on the `workstation` machine for this exercise. This directory already has an `ansible.cfg` configuration file. It also has an inventory file named `inventory` that defines a `web` group, which includes the managed hosts `serverc.lab.example.com` and `serverd.lab.example.com` as members.

This exercise has you create a playbook named `site.yml` in the project directory. This playbook contains one play, which targets members of the `web` host group. The playbook uses tasks that run modules to ensure that the following conditions are met on the managed hosts:

- The `httpd` package is present, using the `ansible.builtin.dnf` module.
- The local `files/index.html` file is copied to `/var/www/html/index.html` on each managed host, using the `ansible.builtin.copy` module.
- The `httpd` service is started and enabled, using the `ansible.builtin.service` module.

You can use the `ansible-navigator doc` command to help you understand the keywords needed for each of the modules.

After the playbook is written, validate its syntax and then use the `ansible-navigator run` command to run the playbook to implement the configuration.

► 1. Change into the `/home/student/playbook-basic` directory.

```
[student@workstation ~]$ cd ~/playbook-basic  
[student@workstation playbook-basic]$
```

- 2. Create a new playbook called /home/student/playbook-basic/site.yml. Start writing a play that targets the hosts in the web host group.
- 2.1. Create and open ~/playbook-basic/site.yml. The first line of the file should be three dashes to indicate the start of the playbook.

```
---
```

- 2.2. The next line starts the play. It needs to start with a dash and a space before the first keyword in the play. Name the play with an arbitrary string documenting the play's purpose, using the name keyword.

```
---
```

```
- name: Install and start Apache HTTPD
```

- 2.3. Add a hosts keyword-value pair to specify that the play run on hosts in the inventory's web host group. Make sure that the hosts keyword is indented two spaces so it aligns with the name keyword in the preceding line.

The complete site.yml file should consist of the following content:

```
---
```

```
- name: Install and start Apache HTTPD
  hosts: web
```

- 3. Continue to edit the /home/student/playbook-basic/site.yml file, and add a tasks keyword and the three tasks for your play that were specified in the instructions.

- 3.1. Add a tasks keyword indented by two spaces (aligned with the hosts keyword) to start the list of tasks. The file should now consist of the following content:

```
---
```

```
- name: Install and start Apache HTTPD
  hosts: web
  tasks:
```

- 3.2. Add the first task. Indent by four spaces, and start the task with a dash and a space, and then give the task a name, such as Ensure httpd package is present. Use the ansible.builtin.dnf module for this task. Indent the module keywords two more spaces; set the package name to httpd and the package state to present. The task should consist of the following content:

```
- name: Ensure httpd package is present
  ansible.builtin.dnf:
    name: httpd
    state: present
```

- 3.3. Add the second task. Match the format of the previous task, and give the task a name, such as Correct index.html is present. Use the ansible.builtin.copy module. Configure the module keywords to set the src key to files/index.html and the dest key to /var/www/html/index.html. The task should consist of the following content:

```
- name: Correct index.html is present
  ansible.builtin.copy:
    src: files/index.html
    dest: /var/www/html/index.html
```

- 3.4. Add the third task to start and enable the `httpd` service. Match the format of the previous two tasks, and give the new task a name, such as `Ensure httpd is started`. Use the `ansible.builtin.service` module for this task. Set the `name` key of the service to `httpd`, the `state` key to `started`, and the `enabled` key to `true`. The task should consist of the following content:

```
- name: Ensure httpd is started
  ansible.builtin.service:
    name: httpd
    state: started
    enabled: true
```

- 3.5. Your entire `site.yml` Ansible Playbook should match the following example. Make sure that the indentation of your play's keywords, the list of tasks, and each task's keywords are all correct.

```
---
- name: Install and start Apache HTTPD
  hosts: web
  tasks:
    - name: Ensure httpd package is present
      ansible.builtin.dnf:
        name: httpd
        state: present

    - name: Correct index.html is present
      ansible.builtin.copy:
        src: files/index.html
        dest: /var/www/html/index.html

    - name: Ensure httpd is started
      ansible.builtin.service:
        name: httpd
        state: started
        enabled: true
```

Save the file and exit.

- 4. Before running your playbook, run the `ansible-navigator run --syntax-check` command to validate its syntax. Correct any reported errors before continuing. You should see output similar to the following:

```
[student@workstation playbook-basic]$ ansible-navigator run \
> -m stdout site.yml --syntax-check
playbook: /home/student/playbook-basic/site.yml
```

- ▶ 5. Run your playbook. Read through the output generated to ensure that all tasks completed successfully.

```
[student@workstation playbook-basic]$ ansible-navigator run \
> -m stdout site.yml

PLAY [Install and start Apache HTTPD] *****

TASK [Gathering Facts] *****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

TASK [Ensure httpd package is present]
*****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

TASK [Correct index.html is present] *****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

TASK [Ensure httpd is started]
*****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

PLAY RECAP *****
serverc.lab.example.com    : ok=4      changed=3      unreachable=0      failed=0
    skipped=0    rescued=0    ignored=0
serverd.lab.example.com    : ok=4      changed=3      unreachable=0      failed=0
    skipped=0    rescued=0    ignored=0
```

- ▶ 6. If all went well, you should be able to run the playbook a second time and see all tasks complete with no changes to the managed hosts.

```
[student@workstation playbook-basic]$ ansible-navigator run \
> -m stdout site.yml

PLAY [Install and start Apache HTTPD] *****

TASK [Gathering Facts] *****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

TASK [Ensure httpd package is present]
*****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

TASK [Correct index.html is present] *****
ok: [serverc.lab.example.com]
ok: [serverd.lab.example.com]
```

```
TASK [Ensure httpd is started]
*****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

PLAY RECAP ****
serverc.lab.example.com    : ok=4    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
serverd.lab.example.com    : ok=4    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
```

- ▶ 7. Use the `curl` command to verify that both `serverc.lab.example.com` and `serverd.lab.example.com` are configured as an HTTPD server.

```
[student@workstation playbook-basic]$ curl serverc.lab.example.com
This is a test page.
[student@workstation playbook-basic]$ curl serverd.lab.example.com
This is a test page.
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish playbook-basic
```

This concludes the section.

Implementing Multiple Plays

Objectives

- Write a playbook that uses multiple plays with per-play privilege escalation, and effectively use automation content navigator to find new modules in available Ansible Content Collections and use them to implement tasks for a play.

Writing Multiple Plays

A playbook is a YAML file containing a list of one or more plays. Remember that a single play is an ordered list of tasks to execute against hosts selected from the inventory. Therefore, if a playbook contains multiple plays, each play might apply its tasks to a separate set of hosts.

This can be very useful when orchestrating a complex deployment which might involve different tasks on different hosts. You can write a playbook that runs one play against one set of hosts, and when that finishes, runs another play against another set of hosts.

Writing a playbook that contains multiple plays is very straightforward. Each play in the playbook is written as a top-level list item in the playbook. Each play is a list item containing the usual play keywords.

The following example shows a simple playbook with two plays. The first play runs against `web.example.com`, and the second play runs against `database.example.com`.

```
---
# This is a simple playbook with two plays

- name: First play
  hosts: web.example.com
  tasks:
    - name: First task
      ansible.builtin.dnf:
        name: httpd
        state: present

    - name: Second task
      ansible.builtin.service:
        name: httpd
        enabled: true

- name: Second play
  hosts: database.example.com
  tasks:
    - name: First task
      ansible.builtin.service:
        name: mariadb
        enabled: true
```

Remote Users and Privilege Escalation in Plays

Plays can use different remote users or privilege escalation settings than is specified by the defaults or the current configuration file. You can override these settings in the play itself at the same level as the hosts or tasks keywords.

User Attributes

Tasks in playbooks are normally executed through a network connection to the managed hosts. Ansible has to connect to each managed host as some user to run those tasks.

By default, if you use `ansible-navigator run` to run a playbook, then Ansible connects to the managed host as the current user inside the automation execution environment, `root`.



Note

If you run an Ansible command that does not use execution environments, such as `ansible-playbook`, then by default Ansible tries to authenticate to the remote managed host using the username of the account you used to run the command.

You can set a `remote_user` directive in your project's `ansible.cfg` file to configure Ansible to use a different user account on the managed hosts when it initially logs in. If you still need the tasks to run as `root`, then you can use privilege escalation to switch to that user after the initial remote connection.

However, you can also specify the remote user that Ansible uses on a play-by-play basis. If the remote user defined in the Ansible configuration for task execution is not suitable, it can be overridden by the `remote_user` keyword within a play.

```
remote_user: remoteuser
```



Important

Ansible determines which user account to use when connecting to a managed host based on the following list, selecting the first username it finds in this order:

- The `ansible_user` variable set for the host or group, if set.
- The `remote_user` from the current play, if set.
- The `remote_user` from the `ansible.cfg` configuration file, if set.

If no value has been set for any of the preceding settings, and you are running playbooks by using `ansible-navigator` with an execution environment, Ansible uses `root`. (If you are using `ansible-playbook`, Ansible uses the name of the user that ran the command.)

Privilege Escalation Attributes

You can also configure privilege escalation in a play. Use the `become` Boolean keyword to enable or disable privilege escalation for an individual play or task. This overrides the setting in the `ansible.cfg` configuration file. It can take `yes` or `true` to enable privilege escalation, or `no` or `false` to disable it.

```
become: true
```

If privilege escalation is enabled, use the `become_method` keyword in the play to specify the privilege escalation method to use for that play. The example below specifies `sudo` as the method for privilege escalation.

```
become_method: sudo
```

Additionally, with privilege escalation enabled, you can use the `become_user` keyword in the play to define the user account to use for privilege escalation in that specific play.

```
become_user: privileged_user
```

The following example demonstrates some of these keywords in a play:

```
- name: /etc/hosts is up-to-date
  hosts: datacenter-west
  remote_user: automation
  become: true

  tasks:
    - name: server.example.com in /etc/hosts
      ansible.builtin.lineinfile:
        path: /etc/hosts
        line: '192.0.2.42 server.example.com server'
        state: present
```

Selecting Modules

The large number of modules packaged with Ansible provides administrators with many tools for common administrative tasks.

The following table lists a small number of useful modules as examples. Many others exist.

Ansible Modules

Category	Modules
Files	<ul style="list-style-type: none"> <code>ansible.builtin.copy</code>: Copy a local file to the managed host. <code>ansible.builtin.file</code>: Set permissions and other properties of files. <code>ansible.builtin.lineinfile</code>: Ensure a particular line is or is not in a file. <code>ansible.posix.synchronize</code>: Synchronize content using <code>rsync</code>.

Category	Modules
Software	<ul style="list-style-type: none"> <code>ansible.builtin.package</code>: Manage packages using the automatically detected package manager native to the operating system. <code>ansible.builtin.dnf</code>: Manage packages using the DNF package manager. <code>ansible.builtin.apt</code>: Manage packages using the APT package manager. <code>ansible.builtin.pip</code>: Manage Python packages from PyPI.
System	<ul style="list-style-type: none"> <code>ansible.posix.firewalld</code>: Manage arbitrary ports and services using firewalld. <code>ansible.builtin.reboot</code>: Reboot a machine. <code>ansible.builtin.service</code>: Manage services. <code>ansible.builtin.user</code>: Add, remove, and manage user accounts.
Net Tools	<ul style="list-style-type: none"> <code>ansible.builtin.get_url</code>: Download files over HTTP, HTTPS, or FTP. <code>ansible.builtin.uri</code>: Interact with web services.

Module Documentation

To see a list of the modules available in your current automation execution environment, run the `ansible-navigator doc -l` command. This displays a list of module names and a synopsis of their functions.

```
[user@controlnode ~]$ ansible-navigator doc -l
add_host                                     Add a host (and alt...
amazon.aws.aws_az_facts                      Gather information ...
amazon.aws.aws_az_info                       Gather information ...
amazon.aws.aws_caller_info                   Get information abo...
amazon.aws.aws_s3                           manage objects in S...
...output omitted...
vyos.vyos.vyos_user                         Manage the collecti...
vyos.vyos.vyos_vlan                          Manage VLANs on Vy0...
wait_for                                    Waits for a conditi...
wait_for_connection                         Waits until remote ...
yum                                         Manages packages wi...
yum_repository                            Add or remove YUM r...
```



Important

The `ansible-navigator doc -l` command displays the short names of modules in the `ansible.builtin` Ansible Content Collection instead of their FQCNs.

Use the `ansible-navigator doc module_name` command to display detailed documentation for a module. If you specify the `-m stdout` option, formatted documentation is displayed to your terminal. If you do not specify that option, leaving `ansible-navigator` in interactive mode, then you can scroll through the documentation in YAML format.

As an alternative, you can run the `ansible-navigator collections` command in interactive mode and explore the documentation for the collections in the current automation execution environment, and their modules.

The module documentation includes a description of what the module is for, a list of the attributes that you can use to control the module in a task, examples of how to use the module, and other metadata.

You can also view a summary of all the attributes you can use with a module by running the `ansible-navigator -s doc module_name` command. This output can serve as a starter template, which can be included in a playbook to implement the module for task execution. Comments are included in the output to explain how to use each attribute. The following example shows this output for the `ansible.builtin.dnf` module.



Note

If you are using the `ansible-playbook` command provided with limited support in Red Hat Enterprise Linux, or from community Ansible, it uses your control node as an execution environment. In that case, you can use an `ansible-doc` command to view documentation for modules installed on the control node, which works with the same options as `ansible-navigator doc`.

However, the `ansible-doc` command on your control node cannot be used to inspect documentation for an automation execution environment being used by `ansible-navigator`.

Running Arbitrary Commands on Managed Hosts

If a module does not exist to automate some task, special modules are available that can run arbitrary commands on your managed hosts.

The `ansible.builtin.command` module is the simplest of these commands. Its `cmd` argument specifies the command that you want to run.

The following example task runs `/opt/bin/makedb.sh` on managed hosts.

```
- name: Run the /opt/bin/makedb.sh command
  ansible.builtin.command:
    cmd: /opt/bin/makedb.sh
```

Unlike most modules, `ansible.builtin.command` is not idempotent. Every time the task is specified in a play, it runs and it reports that it changed something on the managed host, even if nothing needed to be changed.

You can try to make the task safer by configuring it only to run based on the existence of a file. The `creates` option causes the task to run only if a file is missing; the assumption is that if the task runs, it creates that file. The `removes` option causes the task to run only if a file is present; the assumption is that if the task runs, it removes that file.

For example, the following task only runs if `/opt/db/database.db` is not present:

```
- name: Initialize the database
ansible.builtin.command:
  cmd: /opt/bin/makedb.sh
  creates: /opt/db/database.db
```

The `ansible.builtin.command` module cannot access shell environment variables or perform shell operations such as input/output redirection or pipelines. When you need to perform shell processing, you can use the `ansible.builtin.shell` module. Like the `ansible.builtin.command` module, you pass the commands to be executed as arguments to the module.

Both `ansible.builtin.command` and `ansible.builtin.shell` modules require a working Python installation on the managed host. A third module, `ansible.builtin.raw`, can run commands directly using the remote shell, bypassing the module subsystem. This is useful when you are managing systems that cannot have Python installed (for example, a network router). It can also be used to install Python on a managed host.



Important

When possible, try to avoid the `ansible.builtin.command`, `ansible.builtin.shell`, and `ansible.builtin.raw` modules in playbooks, even though they might seem simple to use. Because these run arbitrary commands on the managed hosts, it is very easy to write non-idempotent playbooks with these modules.

If you must use them, it is probably best to use the `ansible.builtin.command` module first, resorting to `ansible.builtin.shell` or `ansible.builtin.raw` only if you need their special features.

As another example, the following task using the `ansible.builtin.shell` module is not idempotent. Every time the play is run, it rewrites `/etc/resolv.conf` even if it already consists of the line `nameserver 192.0.2.1`.

```
- name: Non-idempotent approach with shell module
ansible.builtin.shell:
  cmd: echo "nameserver 192.0.2.1" > /etc/resolv.conf
```

You can create idempotent tasks in several ways using the `ansible.builtin.shell` module, and sometimes making those changes and using `ansible.builtin.shell` is the best approach. But in this case, a better solution would be to use `ansible-navigator doc` to discover the `ansible.builtin.copy` module and use that to get the desired effect.

The following example does not rewrite the `/etc/resolv.conf` file if it already consists of the correct content:

```
- name: Idempotent approach with copy module
ansible.builtin.copy:
  dest: /etc/resolv.conf
  content: "nameserver 192.0.2.1\n"
```

The `ansible.builtin.copy` module tests to see if the state has already been met, and if so, it makes no changes. The `ansible.builtin.shell` module allows a lot of flexibility, but also requires more attention to ensure that it runs with idempotency.

You can run idempotent playbooks repeatedly to ensure systems are in a particular state without disrupting those systems if they already are.

YAML Syntax

The last part of this section investigates some variations of YAML or Ansible Playbook syntax that you might encounter.

YAML Comments

Comments can also be used to aid readability. In YAML, everything to the right of the number sign (#) is a comment. If there is content to the left of the comment, precede the hash with a space.

```
# This is a YAML comment  
  
some data # This is also a YAML comment
```

YAML Strings

Strings in YAML do not normally need to be put in quotation marks even if the string contains no spaces. You can enclose strings in either double or single quotation marks.

```
this is a string  
  
'this is another string'  
  
"this is yet another a string"
```

You can write multiline strings in either of two ways. You can use the vertical bar (|) character to denote that newline characters within the string are to be preserved.

```
include_newlines: |  
    Example Company  
    123 Main Street  
    Atlanta, GA 30303
```

You can also write multiline strings using the greater-than (>) character to indicate that newline characters are to be converted to spaces and that leading white spaces in the lines are to be removed. This method is often used to break long strings at space characters so that they can span multiple lines for better readability.

```
fold_newlines: >  
    This is an example  
    of a long string,  
    that will become  
    a single sentence once folded.
```

YAML Dictionaries

You have seen collections of key-value pairs written as an indented block, as follows:

```
name: svcrole
svbservice: httpd
svcport: 80
```

Dictionaries can also be written in an inline block format enclosed in braces, as follows:

```
{name: svcrole, svbservice: httpd, svcport: 80}
```

Avoid the inline block format because it is harder to read. However, there is at least one situation in which it is more commonly used. The use of *roles* is discussed later in this course. When a playbook includes a list of roles, it is more common to use this syntax to make it easier to distinguish roles included in a play from the variables being passed to a role.

YAML Lists

You have also seen lists written with the normal single-dash syntax:

```
hosts:
  - servera
  - serverb
  - serverc
```

Lists also have an inline format enclosed in square braces, as follows:

```
hosts: [servera, serverb, serverc]
```

You should avoid this syntax because it is usually harder to read.

Obsolete Playbook Shorthand

Some playbooks might use an earlier shorthand method to define tasks by putting the key-value pairs for the module on the same line as the module name. For example, you might see this syntax:

```
tasks:
  - name: Shorthand form
    ansible.builtin.service: name=httpd enabled=true state=started
```

Normally you would write the same task as follows:

```
tasks:
  - name: Normal form
    ansible.builtin.service:
      name: httpd
      enabled: true
      state: started
```

You should generally avoid the shorthand form and use the normal form.

The normal form has more lines, but it is easier to work with. The task's keywords are stacked vertically and are easier to differentiate. Your eyes can move straight down the play with less left-to-right motion, making it easier to read.

Also, the normal syntax is native YAML; the shorthand form is not. Syntax highlighting tools in text editors can help you more effectively if you use the normal format than if you use the shorthand format.

You might see this syntax in documentation and earlier playbooks from other people, and the syntax does still function.



References

Intro to playbooks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

Working with playbooks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks.html

Module Maintenance & Support – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/modules_support.html

Adding modules and plugins locally – Ansible Documentation

https://docs.ansible.com/ansible/latest/dev_guide/developing_locally.html

YAML Syntax – Ansible Documentation

https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html

Ansible.Builtin – Ansible Documentation

<https://docs.ansible.com/ansible/latest/collections/ansible/builtin/index.html>

► Guided Exercise

Implementing Multiple Plays

In this exercise, you write and use a playbook containing multiple plays.

Outcomes

- You should be able to construct and execute a playbook to manage configuration and perform administration of a managed host.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start playbook-multi
```

Instructions

- 1. The `/home/student/playbook-multi` directory has been created on the `workstation` machine for your Ansible project. The directory has already been populated with an `ansible.cfg` configuration file and an inventory file named `inventory`. The managed host, `servera.lab.example.com`, is already defined in this inventory file.

Create a new playbook named `/home/student/playbook-multi/intranet.yml` and add the lines needed to start the first play. It should target the managed host `servera.lab.example.com` and enable privilege escalation.

- 1.1. Change into the `/home/student/playbook-multi` directory.

```
[student@workstation ~]$ cd ~/playbook-multi  
[student@workstation playbook-multi]$
```

- 1.2. Use a text editor to create a new playbook named `/home/student/playbook-multi/intranet.yml`. Add a line consisting of three dashes to the beginning of the file to indicate the start of the YAML file.

```
---
```

- 1.3. Add the following line to the `/home/student/playbook-multi/intranet.yml` file to denote the start of a play named `Enable intranet services`.

```
- name: Enable intranet services
```

14. Add the following line to indicate that the play applies to the `servera.lab.example.com` managed host. Indent the line with two spaces (aligning with the `name` keyword above it) to indicate that it is part of the first play.

```
hosts: servera.lab.example.com
```

15. Add the following line to enable privilege escalation. Indent the line with two spaces (aligning with the keywords above it) to indicate it is part of the first play.

```
become: true
```

16. Add the following line to define the beginning of the `tasks` list. Indent the line with two spaces (aligning with the keywords above it) to indicate that it is part of the first play.

```
tasks:
```

- ▶ 2. As the first task in the first play, define a task that ensures that the `httpd` and `firewalld` packages are up-to-date.

Indent the first line of the task with four spaces. Under the `tasks` keyword in the first play, add the following lines:

```
- name: Latest version of httpd and firewalld installed ①
  ansible.builtin.dnf: ②
    name: ③
      - httpd
      - firewalld
    state: latest ④
```

- ① A descriptive name for the task.
- ② Indented six spaces and calls the `ansible.builtin.dnf` module.
- ③ The `name` keyword, which is a parameter of the `ansible.builtin.dnf` module, and is indented eight spaces. The `name` keyword specifies which packages the module should ensure are up-to-date. The `name` keyword (which is different from the task `name`) can take a list of packages, which are indented ten spaces on the two following lines.
- ④ After the list of packages, the `state` keyword specifies that the module should ensure that the latest version of the packages is installed. The `state` keyword is also a parameter of the `ansible.builtin.dnf` module, and is indented eight spaces.

- ▶ 3. Add a task to the first play's list that ensures that the correct content is in the `/var/www/html/index.html` file.

Add the following lines to define the content for the `/var/www/html/index.html` file. Indent the first line four spaces.

```
- name: Test html page is installed
  ansible.builtin.copy:
    content: "Welcome to the example.com intranet!\n"
    dest: /var/www/html/index.html
```

- The first line provides a descriptive name for the task.
 - The second line is indented six spaces and calls the `ansible.builtin.copy` module.
 - The remaining lines are indented eight spaces and pass the necessary arguments to ensure that the correct content is in the web page.
- ▶ 4. Define two more tasks in the play to ensure that the `firewalld` service is running and starts on boot, and allows connections to the `httpd` service.
- Add the following lines to ensure that the `firewalld` service is enabled and running. Indent the first line four spaces.
- ```
- name: Firewall enabled and running
 ansible.builtin.service:
 name: firewalld
 enabled: true
 state: started
```
- The first line provides a descriptive name for the task.
  - The second line is indented eight spaces and calls the `ansible.builtin.service` module.
  - The remaining lines are indented ten spaces and pass the necessary arguments to ensure that the `firewalld` service is enabled and started.
- Add the following lines to ensure that `firewalld` allows HTTP connections from remote systems. Indent the first line four spaces.
- ```
- name: Firewall permits access to httpd service
  ansible.posix.firewalld:
    service: http
    permanent: true
    state: enabled
    immediate: yes
```
- The first line provides a descriptive name for the task.
 - The second line is indented six spaces and calls the `ansible.posix.firewalld` module.
 - The remaining lines are indented eight spaces and pass the necessary arguments to ensure that remote HTTP connections are permanently allowed.
- ▶ 5. Add a final task to the first play that ensures that the `httpd` service is running and starts at boot.

Add the following lines to ensure that the `httpd` service is enabled and running. Indent the first line four spaces.

```
- name: Web server enabled and running
  ansible.builtin.service:
    name: httpd
    enabled: true
    state: started
```

- The first line provides a descriptive name for the task.
- The second line is indented six spaces and calls the `ansible.builtin.service` module.
- The remaining lines are indented eight spaces and pass the necessary arguments to ensure that the `httpd` service is enabled and running.

- ▶ 6. In the `/home/student/playbook-multi/intranet.yml` file, define a second play that targets `localhost` and tests the intranet web server. (Plays that run on `localhost` are run inside the automation execution environment by `ansible-navigator`, and not directly on your control node.) It does not need privilege escalation.
- 6.1. Add the following line to define the start of a second play. Note that there is no indentation.

```
- name: Test intranet web server
```

- 6.2. Add the following line to indicate that the play runs on the automation execution environment, `localhost`. Indent the line two spaces to indicate that it is contained by the second play.

```
hosts: localhost
```

- 6.3. Add the following line to disable privilege escalation. Align the indentation with the `hosts` keyword above it.

```
become: false
```

- 6.4. Add the following line to the `/home/student/playbook-multi/intranet.yml` file to define the beginning of the `tasks` list. Indent the line two spaces to indicate that it is contained by the second play.

```
tasks:
```

- ▶ 7. Add a single task to the second play, and use the `ansible.builtin.uri` module to request content from `http://servera.lab.example.com`. The task should verify a return HTTP status code of 200. Configure the task to place the returned content in the task results variable.

Add the following lines to create the task for verifying the web service from the control node. Indent the first line four spaces.

```
- name: Connect to intranet web server
  ansible.builtin.uri:
    url: http://servera.lab.example.com
    return_content: yes
    status_code: 200
```

- The first line provides a descriptive name for the task.
- The second line is indented with six spaces and calls the `ansible.builtin.uri` module.
- The remaining lines are indented with eight spaces and pass the necessary arguments to execute a query for web content from the control node to the managed host and verify the status code received.
- The `return_content` keyword ensures that the server's response is added to the task results.

► 8. Verify that the final `/home/student/playbook-multi/intranet.yml` playbook reflects the following structured content, and then save and close the file.

```
---
- name: Enable intranet services
  hosts: servera.lab.example.com
  become: true
  tasks:
    - name: Latest version of httpd and firewalld installed
      ansible.builtin.dnf:
        name:
          - httpd
          - firewalld
        state: latest

    - name: Test html page is installed
      ansible.builtin.copy:
        content: "Welcome to the example.com intranet!\n"
        dest: /var/www/html/index.html

    - name: Firewall enabled and running
      ansible.builtin.service:
        name: firewalld
        enabled: true
        state: started

    - name: Firewall permits access to httpd service
      ansible.posix.firewalld:
        service: http
        permanent: true
        state: enabled
        immediate: yes

    - name: Web server enabled and running
      ansible.builtin.service:
```

```

name: httpd
enabled: true
state: started

- name: Test intranet web server
  hosts: localhost
  become: false
  tasks:
    - name: Connect to intranet web server
      ansible.builtin.uri:
        url: http://servera.lab.example.com
        return_content: yes
        status_code: 200

```

- ▶ 9. Run the `ansible-navigator run --syntax-check` command to validate the syntax of the `/home/student/playbook-multi/intranet.yml` playbook.

```
[student@workstation playbook-multi]$ ansible-navigator run \
> -m stdout intranet.yml --syntax-check
playbook: /home/student/playbook-multi/intranet.yml
```

- ▶ 10. Run the playbook using the `ansible-navigator run` command. Read through the generated output to ensure that all tasks completed successfully. Verify that an HTTP GET request to `http://servera.lab.example.com` provides the correct content.

- 10.1. Run the playbook using the `ansible-navigator run` command.

```
[student@workstation playbook-multi]$ ansible-navigator run \
> -m stdout intranet.yml

PLAY [Enable intranet services] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Latest version of httpd and firewalld installed] ****
changed: [servera.lab.example.com]

TASK [Test html page is installed] ****
changed: [servera.lab.example.com]

TASK [Firewall enabled and running] ****
changed: [servera.lab.example.com]

TASK [Firewall permits access to httpd service] ****
changed: [servera.lab.example.com]

TASK [Web server enabled and running] ****
changed: [servera.lab.example.com]

PLAY [Test intranet web server] ****
TASK [Gathering Facts] ****
```

```
ok: [localhost]

TASK [Connect to intranet web server] ****
ok: [localhost]

PLAY RECAP ****
localhost : ok=2    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
servera.lab.example.com : ok=6    changed=5    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
```

- 10.2. Use the `curl` command to verify that an HTTP GET request to `http://servera.lab.example.com` provides the correct content.

```
[student@workstation playbook-multi]$ curl http://servera.lab.example.com
Welcome to the example.com intranet!
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish playbook-multi
```

This concludes the section.

► Lab

Implementing an Ansible Playbook

In this lab, you configure and perform administrative tasks on managed hosts using a playbook.

Outcomes

- You should be able to construct and run an Ansible Playbook to install, configure, and verify the status of web and database services on a managed host.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start playbook-review
```

The `/home/student/playbook-review` working directory has been created on the `workstation` machine for the Ansible project. The directory has already been populated with an `ansible.cfg` configuration file and an `inventory` file. The managed host, `serverb.lab.example.com`, is already defined in this inventory file.

Instructions

1. Change into the `/home/student/playbook-review` directory and create a new playbook called `internet.yml`. Add the necessary entries to start a first play named `Enable internet services` and specify its intended managed host, `serverb.lab.example.com`. Add the necessary entry to enable privilege escalation, and one to start a task list.
2. Add the necessary entries to the `/home/student/playbook-review/internet.yml` file to define a task that installs the latest versions of the `firewalld`, `httpd`, `mariadb-server`, `php`, and `php-mysqld` packages. Indent the beginning of the entry four spaces.
3. Add the necessary entries to the `/home/student/playbook-review/internet.yml` file to define the firewall configuration tasks. They should ensure that the `firewalld` service is enabled and running, and that access is allowed to the `http` service. Indent the beginning of these entries four spaces.
4. Add the necessary entries to ensure the `httpd` and `mariadb` services are enabled and running. Indent the beginning of these entries four spaces.
5. Add the necessary entry that uses the `ansible.builtin.copy` module to copy the `/home/student/playbook-review/index.php` file to the `/var/www/html/` directory on the managed host. Ensure the file mode is set to 0644. Indent the beginning of these entries four spaces.
6. Define another play in the `/home/student/playbook-review/internet.yml` file for a task to be performed on the control node. This play tests access to the web server that

should be running on the `serverb.lab.example.com` managed host. This play does not require privilege escalation, and runs on the `workstation.lab.example.com` managed host.

7. Add the necessary entry that tests the web service running on `serverb` from the control node using the `ansible.builtin.uri` module. Look for a return status code of 200. Indent the beginning of the entry four spaces.
8. Validate the syntax of the `internet.yml` playbook.
9. Use the `ansible-navigator run` command to run the playbook. Read through the generated output to ensure that all tasks completed successfully.

Evaluation

Grade your work by running the `lab grade playbook-review` command from your `workstation` machine. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade playbook-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish playbook-review
```

This concludes the section.

► Solution

Implementing an Ansible Playbook

In this lab, you configure and perform administrative tasks on managed hosts using a playbook.

Outcomes

- You should be able to construct and run an Ansible Playbook to install, configure, and verify the status of web and database services on a managed host.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start playbook-review
```

The `/home/student/playbook-review` working directory has been created on the `workstation` machine for the Ansible project. The directory has already been populated with an `ansible.cfg` configuration file and an `inventory` file. The managed host, `serverb.lab.example.com`, is already defined in this inventory file.

Instructions

- Change into the `/home/student/playbook-review` directory and create a new playbook called `internet.yml`. Add the necessary entries to start a first play named `Enable internet services` and specify its intended managed host, `serverb.lab.example.com`. Add the necessary entry to enable privilege escalation, and one to start a task list.
 - Add the following entry to the beginning of the `/home/student/playbook-review/internet.yml` file to begin the YAML format.

- Add the following entry to denote the start of a play named `Enable internet services`.

```
- name: Enable internet services
```

- Add the following entry to indicate that the play applies to the `serverb.lab.example.com` managed host. Make sure that the beginning of the entry is indented two spaces.

```
hosts: serverb.lab.example.com
```

- 1.4. Add the following entry to enable privilege escalation. Indent the beginning of the entry two spaces.

```
become: true
```

- 1.5. Add the following entry to define the beginning of the tasks list. Indent the beginning of the entry two spaces.

```
tasks:
```

2. Add the necessary entries to the /home/student/playbook-review/internet.yml file to define a task that installs the latest versions of the firewalld, httpd, mariadb-server, php, and php-mysqlnd packages. Indent the beginning of the entry four spaces.

```
- name: Latest version of all required packages installed
  ansible.builtin.dnf:
    name:
      - firewalld
      - httpd
      - mariadb-server
      - php
      - php-mysqlnd
    state: latest
```

3. Add the necessary entries to the /home/student/playbook-review/internet.yml file to define the firewall configuration tasks. They should ensure that the firewalld service is enabled and running, and that access is allowed to the http service. Indent the beginning of these entries four spaces.

```
- name: firewalld enabled and running
  ansible.builtin.service:
    name: firewalld
    enabled: true
    state: started

- name: firewalld permits http service
  ansible.posix.firewalld:
    service: http
    permanent: true
    state: enabled
    immediate: yes
```

4. Add the necessary entries to ensure the httpd and mariadb services are enabled and running. Indent the beginning of these entries four spaces.

```
- name: httpd enabled and running
  ansible.builtin.service:
    name: httpd
    enabled: true
    state: started

- name: mariadb enabled and running
  ansible.builtin.service:
```

```
name: mariadb
enabled: true
state: started
```

5. Add the necessary entry that uses the `ansible.builtin.copy` module to copy the `/home/student/playbook-review/index.php` file to the `/var/www/html/` directory on the managed host. Ensure the file mode is set to 0644. Indent the beginning of these entries four spaces.

```
- name: Test php page is installed
  ansible.builtin.copy:
    src: index.php
    dest: /var/www/html/index.php
    mode: 0644
```

6. Define another play in the `/home/student/playbook-review/internet.yml` file for a task to be performed on the control node. This play tests access to the web server that should be running on the `serverb.lab.example.com` managed host. This play does not require privilege escalation, and runs on the `workstation.lab.example.com` managed host.

- 6.1. Add the following entry to denote the start of a second play named `Test internet web server`.

```
- name: Test internet web server
```

- 6.2. Add the following entry to indicate that the play applies to the `workstation` managed host. Indent the beginning of the entry two spaces.

```
hosts: workstation
```

- 6.3. Add the following entry after the `hosts` keyword to disable privilege escalation for the second play. Indent the beginning of the entry two spaces.

```
become: false
```

- 6.4. Add the following entry to the `/home/student/playbook-review/internet.yml` file to define the beginning of the `tasks` list. Indent the beginning of the entry two spaces.

```
tasks:
```

7. Add the necessary entry that tests the web service running on `serverb` from the control node using the `ansible.builtin.uri` module. Look for a return status code of 200. Indent the beginning of the entry four spaces.

```
- name: Connect to internet web server
  ansible.builtin.uri:
    url: http://serverb.lab.example.com
    status_code: 200
```

8. Validate the syntax of the `internet.yml` playbook.

```
[student@workstation playbook-review]$ ansible-navigator run \
> -m stdout internet.yml --syntax-check
playbook: /home/student/playbook-review/internet.yml
```

9. Use the `ansible-navigator run` command to run the playbook. Read through the generated output to ensure that all tasks completed successfully.

```
[student@workstation playbook-review]$ ansible-navigator run \
> -m stdout internet.yml
PLAY [Enable internet services] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Latest version of all required packages installed] ****
changed: [serverb.lab.example.com]

TASK [firewalld enabled and running] ****
ok: [serverb.lab.example.com]

TASK [firewalld permits http service] ****
changed: [serverb.lab.example.com]

TASK [httpd enabled and running] ****
changed: [serverb.lab.example.com]

TASK [mariadb enabled and running] ****
changed: [serverb.lab.example.com]

TASK [Test php page is installed]
 ****
changed: [serverb.lab.example.com]

PLAY [Test internet web server] ****
TASK [Gathering Facts] ****
ok: [workstation]

TASK [Connect to internet web server] ****
ok: [workstation]

PLAY RECAP ****
serverb.lab.example.com    : ok=7      changed=5      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0
workstation                 : ok=2      changed=0      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0
```

Evaluation

Grade your work by running the `lab grade playbook-review` command from your workstation machine. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade playbook-review
```

Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish playbook-review
```

This concludes the section.

Summary

- A *play* is an ordered list of tasks that run against hosts selected from the inventory.
- A *playbook* is a text file that contains a list of one or more plays to run in order.
- Ansible Playbooks are written in YAML format.
- Ansible plays are idempotent, which means they avoid making any changes if they detect that the current state matches the desired final state.
- YAML files are structured using space indentation to represent the data hierarchy.
- Tasks are implemented using standardized code packaged as Ansible modules.
- Ansible modules are packaged into *Ansible Content Collections*, which are a distribution format for Ansible content that can include playbooks, roles, modules, and plug-ins.
- The `ansible-navigator doc` command can list modules in your automation execution environments, and provide documentation and example code snippets of how to use them in playbooks.
- The `ansible-navigator run` command is used to run playbooks and validate playbook syntax.

Chapter 3

Managing Variables and Facts

Goal

Write playbooks that use variables to simplify management of the playbook and facts to reference information about managed hosts.

Objectives

- Create and reference variables that affect particular hosts or host groups, the play, or the global environment, and describe how variable precedence works.
- Encrypt sensitive variables using Ansible Vault, and run playbooks that reference Vault-encrypted variable files.
- Reference data about managed hosts using Ansible facts, and configure custom facts on managed hosts.

Sections

- Managing Variables (and Guided Exercise)
- Managing Secrets (and Guided Exercise)
- Managing Facts (and Guided Exercise)

Lab

- Managing Variables and Facts

Managing Variables

Objectives

- Create and reference variables that affect particular hosts or host groups, the play, or the global environment, and describe how variable precedence works.

Introduction to Ansible Variables

Ansible supports variables that can be used to store values that can then be reused throughout files in an Ansible project. This can simplify the creation and maintenance of a project and reduce the number of errors.

Variables provide a convenient way to manage dynamic values for a given environment in your Ansible project. Examples of values that variables might contain include:

- Users to create
- Packages to install
- Services to restart
- Files to remove
- Archives to retrieve from the internet

Naming Variables

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

The following table illustrates the difference between invalid and valid variable names.

Examples of Invalid and Valid Ansible Variable Names

Invalid variable names	Valid variable names
web server	web_server
remote.file	remote_file
1st file	file_1 file1
remoteserver\$1	remote_server_1 remote_server1

Defining Variables

Variables can be defined in a variety of places in an Ansible project. If a variable is set using the same name in two places, and those settings have different values, *precedence* determines which value is used.

You can set a variable that affects a group of hosts or only individual hosts. Some variables are *facts* that can be set by Ansible based on the configuration of a system. Other variables can be set inside the playbook, and affect one play in that playbook, or only one task in that play. You can also set *extra variables* on the `ansible-navigator run` command line by using the `--extra-vars` or `-e` option and specifying those variables, and they override all other values for that variable name.

The following simplified list shows ways to define a variable, ordered from the lowest precedence to the highest:

- Group variables defined in the inventory
- Group variables defined in files in a `group_vars` subdirectory in the same directory as the inventory or the playbook
- Host variables defined in the inventory
- Host variables defined in files in a `host_vars` subdirectory in the same directory as the inventory or the playbook
- Host facts, discovered at runtime
- Play variables in the playbook (`vars` and `vars_files`)
- Task variables
- Extra variables defined on the command line

For example, a variable that is set to affect the `all` host group is overridden by a variable that has the same name and is set to affect a single host.

One recommended practice is to choose globally unique variable names, so that you do not have to consider precedence rules. However, sometimes you might want to use precedence to cause different hosts or host groups to get different settings than your defaults.

If the same variable name is defined at more than one level, the level with the highest precedence wins. A narrow scope, such as a host variable or a task variable, takes precedence over a wider scope, such as a group variable or a play variable. Variables defined by the inventory are overridden by variables defined by the playbook. Extra variables defined on the command line with the `--extra-vars`, or `-e`, option have the highest precedence.

A detailed and more precise discussion of variable precedence is available in the Ansible documentation at "Variable precedence: Where should I put a variable?" [https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable].

Variables in Playbooks

Variables play an important role in Ansible Playbooks because they ease the management of variable data in a playbook.

Defining Variables in Playbooks

When writing plays, you can define your own variables and then invoke those values in a task. For example, you can define a variable named `web_package` with a value of `httpd`. A task can then call the variable using the `ansible.builtin.dnf` module to install the `httpd` package.

You can define playbook variables in multiple ways. One common method is to place a variable in a `vars` block at the beginning of a play:

```
- hosts: all
  vars:
    user: joe
    home: /home/joe
```

It is also possible to define playbook variables in external files. In this case, instead of using a `vars` block in a play in the playbook, you can use the `vars_files` directive followed by a list of names for external variable files relative to the location of the playbook:

```
- hosts: all
  vars_files:
    - vars/users.yml
```

The playbook variables are then defined in those files in YAML format:

```
user: joe
home: /home/joe
```

Using Variables in Playbooks

After you have declared variables, you can use the variables in tasks. Variables are referenced by placing the variable name in double braces (`{{ }}`). Ansible substitutes the variable with its value when the task is executed.

```
vars:
  user: joe

tasks:
  # This line will read: Creates the user joe
  - name: Creates the user {{ user }}
    user:
      # This line will create the user named Joe
      name: "{{ user }}"
```

**Important**

When a variable is used as the first element to start a value, quotes are mandatory. This prevents Ansible from interpreting the variable reference as starting a YAML dictionary. The following message appears if quotes are missing:

```
ansible.builtin.dnf:
    name: {{ service }}
        ^ 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 }}"
```

Host Variables and Group Variables

Inventory variables that apply directly to hosts fall into two broad categories: *host variables* apply to a specific host, and *group variables* apply to all hosts in a host group or in a group of host groups. Host variables take precedence over group variables, but variables defined by a playbook take precedence over both.

One way to define host variables and group variables is to do it directly in the inventory file.

**Note**

This is an earlier approach to defining host and group variables, but you might see it used because it puts all the inventory information and variable settings for hosts and host groups in one file.

- Defining the `ansible_user` host variable for `demo.example.com`:

```
[servers]
demo.example.com  ansible_user=joe
```

- Defining the `user` group variable for the `servers` host group.

```
[servers]
demo1.example.com
demo2.example.com

[servers:vars]
user=joe
```

- Defining the `user` group variable for the `servers` group, which consists of two host groups each with two servers.

```
[servers1]
demo1.example.com
demo2.example.com

[servers2]
demo3.example.com
demo4.example.com

[servers:children]
servers1
servers2

[servers:vars]
user=joe
```

Some disadvantages of this approach are that it makes the inventory file more difficult to work with, it mixes information about hosts and variables in the same file, and it uses an obsolete syntax.

Using Directories to Populate Host and Group Variables

You can define variables for hosts and host groups by creating two directories, `group_vars` and `host_vars`, in the same working directory as the inventory file or playbook. These directories contain files defining group variables and host variables, respectively.



Important

The recommended practice is to define inventory variables using `host_vars` and `group_vars` directories, and not to define them directly in the inventory files.

To define group variables for the `servers` group, you would create a YAML file named `group_vars/servers`, and then the contents of that file would set variables to values using the same syntax as in a playbook:

```
user: joe
```

Likewise, to define host variables for a particular host, create a file with a name matching the host in the `host_vars` directory to contain the host variables.

The following examples illustrate this approach in more detail. Consider a scenario where you need to manage two data centers, and the data center hosts are defined in the `~/project/inventory` inventory file:

```
[datacenter1]
demo1.example.com
demo2.example.com

[datacenter2]
demo3.example.com
demo4.example.com
```

```
[datacenters:children]
datacenter1
datacenter2
```

- If you need to define a general value for all servers in both data centers, set a group variable for the `datacenters` host group:

```
[admin@station project]$ cat ~/project/group_vars/datacenters
package: httpd
```

- If you need to define difference values for each data center, set a group variable for each data center host group:

```
[admin@station project]$ cat ~/project/group_vars/datacenter1
package: httpd
[admin@station project]$ cat ~/project/group_vars/datacenter2
package: apache
```

- If you need to define different values for each managed host in every data center, then define the variables in separate host variable files:

```
[admin@station project]$ cat ~/project/host_vars/demo1.example.com
package: httpd
[admin@station project]$ cat ~/project/host_vars/demo2.example.com
package: apache
[admin@station project]$ cat ~/project/host_vars/demo3.example.com
package: mariadb-server
[admin@station project]$ cat ~/project/host_vars/demo4.example.com
package: mysql-server
```

The directory structure for the example project, `project`, if it contained all the example files above, would appear as follows:

```
project
├── ansible.cfg
├── group_vars
│   ├── datacenters
│   │   ├── datacenters1
│   │   └── datacenters2
│   └── host_vars
│       ├── demo1.example.com
│       ├── demo2.example.com
│       ├── demo3.example.com
│       └── demo4.example.com
└── inventory
└── playbook.yml
```

**Note**

Ansible looks for `host_vars` and `group_vars` subdirectories relative to both the inventory file and the playbook file.

If your inventory and your playbook happen to be in the same directory, this is simple and Ansible looks in that directory for those subdirectories. If your inventory and your playbook are in separate directories, then Ansible looks in both places for `host_vars` and `group_vars` subdirectories. The playbook subdirectories have higher precedence.

Overriding Variables from the Command Line

Inventory variables are overridden by variables set in a playbook, but both kinds of variables can be overridden through arguments passed to the `ansible-navigator run` command on the command line. Variables set on the command line are called *extra variables*.

Extra variables can be useful when you need to override the defined value for a variable for a one-off run of a playbook. For example:

```
[user@demo ~]$ ansible-navigator run main.yml -e "package=apache"
```

Using Dictionaries as Variables

Instead of assigning configuration data that relates to the same element to multiple variables, administrators can use *dictionaries*. A dictionary is a data structure containing key-value pairs, where the values can also be dictionaries.

For example, consider the following snippet:

```
user1_first_name: Bob
user1_last_name: Jones
user1_home_dir: /users/bjones
user2_first_name: Anne
user2_last_name: Cook
user2_home_dir: /users/acock
```

This could be rewritten as a dictionary called `users`:

```
users:
bjones:
  first_name: Bob
  last_name: Jones
  home_dir: /users/bjones
acock:
  first_name: Anne
  last_name: Cook
  home_dir: /users/acock
```

You can then use the following variables to access user data:

```
# Returns 'Bob'
users.bjones.first_name

# Returns '/users/acook'
users.acook.home_dir
```

Because the variable is defined as a Python dictionary, an alternative syntax is available.

```
# Returns 'Bob'
users['bjones']['first_name']

# Returns '/users/acook'
users['acook']['home_dir']
```



Important

The dot notation can cause problems if the key names are the same as names of Python methods or attributes, such as `discard`, `copy`, `add`, and so on. Using the brackets notation can help avoid conflicts and errors.

Both syntaxes are valid, but to make troubleshooting easier, Red Hat recommends that you use one syntax consistently in all files throughout any given Ansible project.

Capturing Command Output with Registered Variables

You can use the `register` statement to capture the output of a command or other information about the execution of a module. The output is saved into a variable that can be used later in the playbook for either debugging purposes or to achieve something else, such as applying a particular configuration setting based on a command's output.

The following play demonstrates how to capture the output of a command for debugging purposes:

```
---
- name: Installs a package and prints the result
  hosts: all
  tasks:
    - name: Install the package
      ansible.builtin.dnf:
        name: httpd
        state: installed
      register: install_result

    - debug:
        var: install_result
```

When you run the play, the `debug` module dumps the value of the `install_result` registered variable to the terminal.

```
[user@demo ~]$ ansible-navigator run playbook.yml -m stdout
PLAY [Installs a package and prints the result] ****
```

```
TASK [setup] ****
ok: [demo.example.com]

TASK [Install the package] ****
ok: [demo.example.com]

TASK [debug] ****
ok: [demo.example.com] => {
    "install_result": {
        "changed": false,
        "msg": "",
        "rc": 0,
        "results": [
            "httpd-2.4.51-7.el9_0.x86_64 providing httpd is already installed"
        ]
    }
}

PLAY RECAP ****
demo.example.com      : ok=3      changed=0      unreachable=0      failed=0      skipped=0
                          rescued=0     ignored=0
```



References

How to build your inventory – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html

Using Variables – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html

Variable precedence: Where should I put a variable?

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

YAML Syntax – Ansible Documentation

https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html

► Guided Exercise

Managing Variables

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

Outcomes

- Define variables in a playbook.
- Create tasks that use defined variables.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start data-variables
```

Instructions

- 1. Change into the `/home/student/data-variables` directory.

```
[student@workstation ~]$ cd ~/data-variables  
[student@workstation data-variables]$
```

- 2. Over the next several steps, you create a playbook that consists of a single play that installs the Apache web server and opens the ports for the service to be reachable. The play also queries the web server to ensure it is up and running.

Create a playbook named `playbook.yml`. Create a play named "Deploy and start Apache HTTPD service", target the host group `webserver` as the managed hosts, and define the following variables in its `vars` section:

Variables

Variable	Description
web_pkg	Web server package to install
firewall_pkg	Firewall package to install
web_service	Web service to manage
firewall_service	Firewall service to manage
python_pkg	Required package for the <code>uri</code> module
rule	The service name to open

```
---
- name: Deploy and start Apache HTTPD service
  hosts: webserver
  vars:
    web_pkg: httpd
    firewall_pkg: firewalld
    web_service: httpd
    firewall_service: firewalld
    python_pkg: python3-PyMySQL
    rule: http
```

- 3. Create the `tasks` block and create the first task, using the `ansible.builtin.dnf` module to make sure the latest versions of the required packages are installed.

```
tasks:
  - name: Required packages are installed and up to date
    ansible.builtin.dnf:
      name:
        - "{{ web_pkg }}"
        - "{{ firewall_pkg }}"
        - "{{ python_pkg }}"
      state: latest
```



Note

You can use `ansible-navigator doc ansible.builtin.dnf -m stdout` to review the syntax for the `ansible.builtin.dnf` module. (If you have the `ansible-core` package installed, you can also use `ansible-doc ansible.builtin.dnf`.)

The documentation shows that the module's `name` directive can take a list of packages that the module should work with, so that you do not need separate tasks to make sure that each package is up-to-date.

- ▶ 4. Create two tasks that make sure that the `httpd` and `firewalld` services are started and enabled.

```
- name: The {{ firewall_service }} service is started and enabled
  ansible.builtin.service:
    name: "{{ firewall_service }}"
    enabled: true
    state: started

- name: The {{ web_service }} service is started and enabled
  ansible.builtin.service:
    name: "{{ web_service }}"
    enabled: true
    state: started
```

**Note**

The `ansible.builtin.service` module works differently from the `ansible.builtin.dnf` module, as documented by `ansible-doc ansible.builtin.service`. Its `name` directive takes the name of exactly one service to work with.

You can write a single task that ensures both services are started and enabled, using the `loop` keyword covered later in this course.

- ▶ 5. Add a task that ensures specific content exists in the `/var/www/html/index.html` file.

```
- name: Web content is in place
  ansible.builtin.copy:
    content: "Example web content"
    dest: /var/www/html/index.html
```

- ▶ 6. Add a task that uses the `ansible.posix.firewalld` module to ensure that the firewall ports are open for the `firewalld` service named in the `rule` variable.

```
- name: The firewall port for {{ rule }} is open
  ansible.posix.firewalld:
    service: "{{ rule }}"
    permanent: true
    immediate: true
    state: enabled
```

- 7. Create a new play that queries the web service to ensure that everything has been correctly configured. It must run on `workstation`. Because of that Ansible fact, Ansible does not have to change identity, so set the `become` module to `false`.

You can use the `ansible.builtin.uri` module to inspect a URL. For this task, verify that a status code of 200 is returned to confirm that the web server on `servera.lab.example.com` is running and correctly configured.

```
- name: Verify the Apache service
hosts: workstation
become: false
tasks:
  - name: Ensure the webserver is reachable
    ansible.builtin.uri:
      url: http://servera.lab.example.com
      status_code: 200
```

- 8. When completed, the playbook contains the following content: Review the playbook and confirm that both plays are correct.

```
---
- name: Deploy and start Apache HTTPD service
  hosts: webserver
  vars:
    web_pkg: httpd
    firewall_pkg: firewalld
    web_service: httpd
    firewall_service: firewalld
    python_pkg: python3-PyMySQL
    rule: http

  tasks:
    - name: Required packages are installed and up to date
      ansible.builtin.dnf:
        name:
          - "{{ web_pkg }}"
          - "{{ firewall_pkg }}"
          - "{{ python_pkg }}"
        state: latest

    - name: The {{ firewall_service }} service is started and enabled
      ansible.builtin.service:
        name: "{{ firewall_service }}"
        enabled: true
        state: started

    - name: The {{ web_service }} service is started and enabled
      ansible.builtin.service:
        name: "{{ web_service }}"
        enabled: true
        state: started

    - name: Web content is in place
      ansible.builtin.copy:
```

```
content: "Example web content"
dest: /var/www/html/index.html

- name: The firewall port for {{ rule }} is open
  ansible.posix.firewalld:
    service: "{{ rule }}"
    permanent: true
    immediate: true
    state: enabled

- name: Verify the Apache service
  hosts: workstation
  become: false
  tasks:
    - name: Ensure the webserver is reachable
      ansible.builtin.uri:
        url: http://servera.lab.example.com
        status_code: 200
```

- ▶ 9. Before you run the playbook, use the `ansible-navigator run --syntax-check` command to verify its syntax. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation data-variables]$ ansible-navigator run \
> -m stdout playbook.yml --syntax-check
playbook: /home/student/data-variables/playbook.yml
```

- ▶ 10. Use the `ansible-navigator run` command to run the playbook. Watch the output as Ansible installs the packages, starts and enables the services, and ensures the web server is reachable.

```
[student@workstation data-variables]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Deploy and start Apache HTTPD service] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Required packages are installed and up to date] ****
changed: [servera.lab.example.com]

TASK [The firewalld service is started and enabled] ****
ok: [servera.lab.example.com]

TASK [The httpd service is started and enabled] ****
changed: [servera.lab.example.com]

TASK [Web content is in place] ****
changed: [servera.lab.example.com]

TASK [The firewall port for http is open] ****
changed: [servera.lab.example.com]
```

```
PLAY [Verify the Apache service] ****
TASK [Gathering Facts] ****
ok: [workstation]

TASK [Ensure the webserver is reachable] ****
ok: [workstation]

PLAY RECAP ****
servera.lab.example.com    : ok=6    changed=4    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
workstation                 : ok=2    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
```

Finish

On the workstation machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish data-variables
```

This concludes the section.

Managing Secrets

Objectives

- Encrypt sensitive variables by using Ansible Vault, and run playbooks that reference Vault-encrypted variable files.

Introducing Ansible Vault

Ansible might need access to sensitive data, such as passwords or API keys, to configure managed hosts. Normally, this information is stored as plain text in inventory variables or other Ansible files. In that case, however, any user with access to the Ansible files, or a version control system that stores the Ansible files, would have access to this sensitive data. This poses an obvious security risk.

Ansible Vault, which is included with Ansible, can be used to encrypt and decrypt any data file used by Ansible. To use Ansible Vault, use the command-line tool named `ansible-vault` to create, edit, encrypt, decrypt, and view files.

Ansible Vault can encrypt any data file used by Ansible. This might include inventory variables, included variable files in a playbook, variable files passed as arguments when executing the playbook, or variables defined in Ansible roles.



Important

Ansible Vault does not implement its own cryptographic functions but rather uses an external Python toolkit. Files are protected with symmetric encryption using AES256 with a password as the secret key. Note that the way this is done has not been formally audited by a third party.

Creating an Encrypted File

To create a new encrypted file, use the `ansible-vault create` *filename* command. This command prompts for the new Vault password and then opens a file using the default editor, `vi`. You can export the `EDITOR` environment variable to specify a different default editor. For example, to set the default editor to `nano`, run the `export EDITOR=nano` command.

```
[student@demo ~]$ ansible-vault create secret.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

Instead of entering the Vault password through standard input, you can use a Vault password file to store the Vault password. You need to carefully protect this file using file permissions and other means.

```
[student@demo ~]$ ansible-vault create --vault-password-file=vault-pass secret.yml
```

The cipher used to protect files is AES256 in recent versions of Ansible, but files encrypted with earlier versions might still use 128-bit AES.

Viewing an Encrypted File

You can use the `ansible-vault view filename` command to view an Ansible Vault-encrypted file without opening it for editing.

```
[student@demo ~]$ ansible-vault view secret1.yml
Vault password: secret
my_secret: "yJJvPqhsiusmmPPZdnjndkdNjYNDjdj782meUZcw"
```

Editing an Existing Encrypted File

To edit an existing encrypted file, Ansible Vault provides the `ansible-vault edit filename` command. This command decrypts the file to a temporary file and allows you to edit it. When saved, it copies the content and removes the temporary file.

```
[student@demo ~]$ ansible-vault edit secret.yml
Vault password: redhat
```



Note

The `edit` subcommand always rewrites the file, so you should only use it when making changes. This can have implications when the file is kept under version control. You should always use the `view` subcommand to view the file's contents without making changes.

Encrypting an Existing File

To encrypt a file that already exists, use the `ansible-vault encrypt filename` command. This command can take the names of multiple files to be encrypted as arguments.

```
[student@demo ~]$ ansible-vault encrypt secret1.yml secret2.yml
New Vault password: redhat
Confirm New Vault password: redhat
Encryption successful
```

Use the `--output=OUTPUT_FILE` option to save the encrypted file with a new name. You can only use one input file with the `--output` option.

Decrypting an Existing File

An existing encrypted file can be permanently decrypted by using the `ansible-vault decrypt filename` command. When decrypting a single file, you can use the `--output` option to save the decrypted file under a different name.

```
[student@demo ~]$ ansible-vault decrypt secret1.yml --output=secret1-decrypted.yml
Vault password: redhat
Decryption successful
```

Changing the Password of an Encrypted File

You can use the `ansible-vault rekey filename` command to change the password of an encrypted file. This command can rekey multiple data files at the same time. It prompts for the original password and then the new password.

```
[student@demo ~]$ ansible-vault rekey secret.yml
Vault password: redhat
New Vault password: RedHat
Confirm New Vault password: RedHat
Rekey successful
```

When using a Vault password file, use the `--new-vault-password-file` option:

```
[student@demo ~]$ ansible-vault rekey \
> --new-vault-password-file=NEW_VAULT_PASSWORD_FILE secret.yml
```

Playbooks and Ansible Vault

To run a playbook that accesses files encrypted with Ansible Vault, you need to provide the encryption password to the `ansible-navigator` command. If you do not provide the password, the playbook returns an error:

```
[student@demo ~]$ ansible-navigator run -m stdout test-secret.yml
ERROR! Attempting to decrypt but no vault secrets found
```

You can provide the Vault password using one of the following options:

- Prompt interactively
- Specify the Vault password file
- Use the `ANSIBLE_VAULT_PASSWORD_FILE` environment variable

To provide the Vault password interactively, use `--playbook-artifact-enable false` and `--vault-id @prompt` as illustrated in the following example:

```
[student@demo ~]$ ansible-navigator run -m stdout \
> --playbook-artifact-enable false \
> site.yml --vault-id @prompt
Vault password (default): redhat
```



Important

You must disable playbook artifacts to enter the Vault password interactively. The `ansible-navigator` command hangs if it needs to prompt you for an interactive Vault password and playbook artifacts are not disabled. Playbook artifacts are enabled by default.

You can use the `ansible-navigator --playbook-artifact-enable false` command to disable playbook artifacts.

You can also disable playbook artifacts by modifying your project `ansible-navigator.yml` file or the `.ansible-navigator.yml` file in your home directory. Set the `playbook-artifact` setting in that file to `enable: false`.

The following minimal `ansible-navigator.yml` file disables playbook artifacts:

```
ansible-navigator:  
  playbook-artifact:  
    enable: false
```

Instead of providing the Vault encryption password interactively, you can specify a file that stores the encryption password in plain text by using the `--vault-password-file` option.

The password must be a string stored as a single line in the file. Because that file contains the sensitive plain text password, it is vital that it be protected through file permissions and other security measures.

```
[student@demo ~]$ ansible-navigator run -m stdout site.yml \  
> --vault-password-file=vault-pw-file
```

You can also use the `ANSIBLE_VAULT_PASSWORD_FILE` environment variable to specify the default location of the password file.



Important

You can use multiple Ansible Vault passwords with `ansible-navigator`.

To use multiple passwords, pass multiple `--vault-id` or `--vault-password-file` options to the `ansible-navigator` command.

```
[student@demo ~]$ ansible-navigator run -m stdout \  
> --playbook-artifact-enable false site.yml \  
> --vault-id one@prompt --vault-id two@prompt  
Vault password (one):  
Vault password (two):  
...output omitted...
```

The Vault IDs `one` and `two` preceding `@prompt` can be anything, and you can even omit them entirely. If you use the `--vault-id id` option when you encrypt a file with the `ansible-vault` command, then the password for the matching ID is the first password tried when running the `ansible-navigator` command. If it does not match, then `ansible-navigator` tries the other passwords that you provided. The Vault ID `@prompt` with no ID is actually shorthand for `default@prompt`, which means to prompt for the password for Vault ID `default`.

Recommended Practices for Variable File Management

To simplify management, it makes sense to set up your Ansible project so that sensitive variables and all other variables are kept in separate files. The files containing sensitive variables can then be protected with the `ansible-vault` command.

Remember that the preferred way to manage group variables and host variables is to create directories at the playbook level. The `group_vars` directory normally contains variable files

with names that match the host groups to which they apply. The `host_vars` directory normally contains variable files with names that match the hostnames of managed hosts to which they apply.

You can use subdirectories within the `group_vars` or `host_vars` directories for each host group or managed host. Those directories can contain multiple variable files, and all of those files are used by the host group or managed host.

In the following example project directory for the `playbook.yml` playbook, members of the `webservers` host group use variables in the `group_vars/webservers/vars` file. The `demo.example.com` host uses the variables in both the `host_vars/demo.example.com/vars` and `host_vars/demo.example.com/vault` files.:

```
.  
└── ansible.cfg  
└── group_vars  
    └── webservers  
        └── vars  
└── host_vars  
    └── demo.example.com  
        ├── vars  
        └── vault  
└── inventory  
└── playbook.yml
```

If you do create subdirectories for each host group or managed host, most variables for `demo.example.com` can be placed in the `vars` file, but sensitive variables can be kept secret by placing them in the `vault` file. You can use `ansible-vault` to encrypt the `vault` file and leave the `vars` file as plain text.

You can name files in the `host_vars/demo.example.com` any valid file name you choose. The file names used in the `host_vars/demo.example.com` directory are examples only; they have no special significance. That directory could contain more files, some that are encrypted by Ansible Vault, and some that are not.

Playbook variables (as opposed to inventory variables) can also be protected with Ansible Vault. You can place sensitive playbook variables in a separate file that is encrypted with Ansible Vault, then include that encrypted variables file in a playbook by using a `vars_files` directive. This can be useful, because playbook variables take precedence over inventory variables.

If you are using multiple Vault passwords with your playbook, make sure that each encrypted file is assigned a Vault ID, and that you enter the matching password with that Vault ID when running the playbook. This ensures that the correct password is selected first when decrypting the vault-encrypted file, which is faster than forcing Ansible to try each of the Vault passwords that you provided until it finds the right one.



References

Encrypting content with Ansible Vault – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/vault.html

Keep vaulted variables safely visible – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html#keep-vaulted-variables-safely-visible

► Guided Exercise

Managing Secrets

In this exercise, you encrypt sensitive variables with Ansible Vault to protect them, and then run a playbook that uses those variables.

Outcomes

- Execute a playbook using variables defined in an encrypted file.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start data-secret
```

Instructions

- 1. Change into the `/home/student/data-secret` directory.

```
[student@workstation ~]$ cd ~/data-secret  
[student@workstation data-secret]$
```

- 2. Edit the contents of the encrypted `secret.yml` file. The file can be decrypted using `redhat` as the password. Uncomment the `username` and `pwhash` variable entries.

- 2.1. Edit the encrypted `/home/student/data-secret/secret.yml` file. Enter `redhat` as the Vault password when prompted. The encrypted file opens in the default editor, `vim`.

```
[student@workstation data-secret]$ ansible-vault edit secret.yml  
Vault password: redhat
```

- 2.2. Uncomment the two variable entries (`username` and `pwhash`) by removing the pound sign (#) at the start of each line, and then save and close the file.

- 3. Create a playbook named `/home/student/data-secret/create_users.yml`. The playbook should contain one play (`create user accounts for all our servers` in the following example), which uses the variables defined in the `/home/student/data-secret/secret.yml` encrypted file.

Configure the play to use the `devservers` host group. Run this play as the `devops` user on the remote managed host. Configure the play to create the `ansibleuser1` user defined by the `username` variable. Set the user's password using the password hash stored in the `pwhash` variable.

```

---
- name: create user accounts for all our servers
  hosts: devservers
  become: True
  remote_user: devops
  vars_files:
    - secret.yml
  tasks:
    - name: Creating user from secret.yml
      ansible.builtin.user:
        name: "{{ username }}"
        password: "{{ pwhash }}"

```

- ▶ 4. Verify the syntax of your `create_users.yml` playbook by running the `ansible-navigator run -m stdout --syntax-check` command.

Use the `--vault-id @prompt` option so that it interactively prompts you for the Vault password that decrypts the `secret.yml` file. Resolve any syntax errors in your playbook before you continue.

```

[student@workstation data-secret]$ ansible-navigator run -m stdout \
> --playbook-artifact-enable false create_users.yml \
> --syntax-check --vault-id @prompt
Vault password (default): redhat

playbook: /home/student/data-secret/create_users.yml

```

- ▶ 5. Create a password file named `vault-pass` that contains the password for `ansible-navigator` to use instead of prompting you for a password when it runs the `create_users.yml` playbook. The file must contain the plain text `redhat` as the Vault password. Change the permissions of the file to `0600`.

```

[student@workstation data-secret]$ echo 'redhat' > vault-pass
[student@workstation data-secret]$ chmod 0600 vault-pass

```

- ▶ 6. Run the Ansible Playbook to create the `ansibleuser1` user on a remote system, using the Vault password in the `vault-pass` file to decrypt the hashed password for that user. That password is stored as a variable in the `secret.yml` Ansible Vault encrypted file.

```

[student@workstation data-secret]$ ansible-navigator run \
> -m stdout create_users.yml --vault-password-file=vault-pass

PLAY [create user accounts for all our servers] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Creating users from secret.yml] ****

```

```
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=2    changed=1    unreachable=0    failed=0
skipped=0      rescued=0    ignored=0
```

- ▶ 7. Verify that the playbook ran correctly. The `ansibleuser1` user should exist and have the correct password on the `servera.lab.example.com` machine.

Test this by using `ssh` to log in to the `servera.lab.example.com` machine as the `ansibleuser1` user with `redhat` as the password.

To make sure that SSH only tries to authenticate by password and not by using an SSH key, use the `-o PreferredAuthentications=password` option when you log in.

Log off from `servera` when you have successfully logged in.

```
[student@workstation data-secret]$ ssh -o PreferredAuthentications=password \
> ansibleuser1@servera.lab.example.com
ansibleuser1@servera.lab.example.com's password: redhat
...output omitted...
[ansibleuser1@servera ~]$ exit
logout
Connection to servera.lab.example.com closed.
```

Finish

On the workstation machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish data-secret
```

This concludes the section.

Managing Facts

Objectives

- Reference data about managed hosts using Ansible facts, and configure custom facts on managed hosts.

Describing Ansible Facts

Ansible *facts* are variables that are automatically discovered by Ansible on a managed host. Facts contain host-specific information that can be used just like regular variables in plays, conditionals, loops, or any other statement that depends on a value collected from a managed host.

Some facts gathered for a managed host might include:

- The host name
- The kernel version
- Network interface names
- Network interface IP addresses
- Operating system version
- Number of CPUs
- Available or free memory
- Size and free space of storage devices

You can even create *custom facts*, which are stored on the managed host and are unique to that system.

Facts are a convenient way to retrieve the state of a managed host and to determine what action to take based on that state. For example:

- Your play might restart a server by using a conditional task based on the value of a fact that was gathered, such as the status of a particular service.
- The play might customize a MySQL configuration file depending on the available memory that is reported by a fact.
- The IPv4 address used in a configuration file might be set based on the value of a fact.

Normally, every play runs the `ansible.builtin.setup` module automatically to gather facts, before it performs its first task.

This is reported as the `Gathering Facts` task in Ansible 2.3 and later, or simply as `setup` in earlier versions of Ansible. By default, you do not need to have a task to run `ansible.builtin.setup` in your play. It is normally run automatically for you.

One way to see what facts are gathered for your managed hosts is to run a short playbook that gathers facts and uses the `ansible.builtin.debug` module to print the value of the `ansible_facts` variable.

```
- name: Fact dump
hosts: all
tasks:
  - name: Print all facts
    ansible.builtin.debug:
      var: ansible_facts
```

When you run the playbook, the facts are displayed in the job output:

```
[user@demo ~]$ ansible-navigator run -m stdout facts.yml

PLAY [Fact dump] ****
TASK [Gathering Facts] ****
ok: [demo1.example.com]

TASK [Print all facts] ****
ok: [demo1.example.com] => {
  "ansible_facts": {
    "all_ipv4_addresses": [
      "10.30.0.178",
      "172.25.250.10"
    ],
    "all_ipv6_addresses": [
      "fe80::8389:96fd:e53e:979",
      "fe80::cb51:6814:6342:7bbc"
    ],
    "ansible_local": {}
  },
  "apparmor": {
    "status": "disabled"
  },
  "architecture": "x86_64",
  "bios_date": "04/01/2014",
  "bios_vendor": "SeaBIOS",
  "bios_version": "1.13.0-2.module+el8.2.1+7284+aa32a2c4",
  "board_asset_tag": "NA",
  "board_name": "NA",
  "board_serial": "NA",
  "board_vendor": "NA",
  "board_version": "NA",
  "chassis_asset_tag": "NA",
  "chassis_serial": "NA",
  "chassis_vendor": "Red Hat",
  "chassis_version": "RHEL 7.6.0 PC (i440FX + PIIX, 1996)",
  "cmdline": {
    "BOOT_IMAGE": "(hd0,gpt3)/vmlinuz-5.14.0-70.13.1.el9_0.x86_64",
    "console": "ttyS0,115200n8",
    "crashkernel": "1G-4G:192M,4G-64G:256M,64G-:512M",
    "net.ifnames": "0",
    "no_timer_check": true,
```

```

    "root": "UUID=fb535add-9799-4a27-b8bc-e8259f39a767"
},
...output omitted...

```

The playbook displays the content of the `ansible_facts` variable in JSON format as a dictionary of variables. You can browse the output to see what facts are gathered, and to find facts that you might want to use in your plays.

The following table shows some facts that might be gathered from a managed node and which might be useful in a playbook:

Examples of Ansible Facts

Fact	Variable
Short hostname	<code>ansible_facts['hostname']</code>
Fully qualified domain name	<code>ansible_facts['fqdn']</code>
Main IPv4 address (based on routing)	<code>ansible_facts['default_ipv4']['address']</code>
List of the names of all network interfaces	<code>ansible_facts['interfaces']</code>
Size of the /dev/vda1 disk partition	<code>ansible_facts['devices']['vda']['partitions']['vda1']['size']</code>
List of DNS servers	<code>ansible_facts['dns']['nameservers']</code>
Version of the currently running kernel	<code>ansible_facts['kernel']</code>



Note

Remember that when a variable's value is a dictionary, one of two syntaxes can be used to retrieve the value. To take two examples from the preceding table:

- `ansible_facts['default_ipv4']['address']` can also be written `ansible_facts.default_ipv4.address`
- `ansible_facts['dns']['nameservers']` can also be written `ansible_facts.dns.nameservers`

When a fact is used in a playbook, Ansible dynamically substitutes the variable name for the fact with the corresponding value:

```

---
- hosts: all
  tasks:
    - name: Prints various Ansible facts
      ansible.builtin.debug:
        msg: >
          The default IPv4 address of {{ ansible_facts.fqdn }}
          is {{ ansible_facts.default_ipv4.address }}

```

The following output shows how Ansible was able to query the managed node and dynamically use the system information to update the variable. You can also use facts to create dynamic groups of hosts that match particular criteria.

```

[user@demo ~]$ ansible-navigator run -m stdout playbook.yml
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [demo1.example.com]

TASK [Prints various Ansible facts] ****
ok: [demo1.example.com] => {
    "msg": "The default IPv4 address of demo1.example.com is 172.25.250.10\\n"
}

PLAY RECAP ****
demo1.example.com : ok=2    changed=0    unreachable=0    failed=0    skipped=0
                    rescued=0   ignored=0

```

Ansible Facts Injected as Variables

Before Ansible 2.5, facts were always injected as individual variables prefixed with the string `ansible_` instead of being part of the `ansible_facts` variable. For example, the `ansible_facts['distribution']` fact was called `ansible_distribution`.

Many playbooks still use facts injected as variables instead of the new syntax, which uses the `ansible_facts.*` namespace.

One reason why the Ansible community discourages injecting facts as variables is because it risks unexpected collisions between facts and variables. A fact has a very high precedence that overrides playbook and inventory host and group variables, so this can lead to unexpected side effects.

The following table shows some examples of facts with both the `ansible_*` and `ansible_facts.*` names.

Comparison of Selected Ansible Fact Names

<code>ansible_facts.* name</code>	<code>ansible_* name</code>
<code>ansible_facts['hostname']</code>	<code>ansible_hostname</code>
<code>ansible_facts['fqdn']</code>	<code>ansible_fqdn</code>

<code>ansible_facts.* name</code>	<code>ansible_* name</code>
<code>ansible_facts['default_ipv4']['address']</code>	<code>ansible_default_ipv4['address']</code>
<code>ansible_facts['interfaces']</code>	<code>ansible_interfaces</code>
<code>ansible_facts['devices']['vda']['partitions']['vda1']['size']</code>	<code>ansible_devices['vda']['partitions']['vda1']['size']</code>
<code>ansible_facts['dns']['nameservers']</code>	<code>ansible_dns['nameservers']</code>
<code>ansible_facts['kernel']</code>	<code>ansible_kernel</code>



Important

Currently, Ansible recognizes both the new fact-naming system (using `ansible_facts`) and the earlier, pre-2.5 "facts injected as separate variables" naming system.

You can disable the `ansible_` naming system by setting the `inject_facts_as_vars` parameter in the `[defaults]` section of the Ansible configuration file to `false`. The default setting is currently `true`.

If it is set to `false`, you can only reference Ansible facts using the new `ansible_facts.*` naming system. In that case, attempts to reference facts through the `ansible_*` namespace results in an error.

Turning off Fact Gathering

Sometimes, you do not want to gather facts for your play. This might be for several reasons:

- You might not be using any facts and want to speed up the play, or reduce load caused by the play on the managed hosts.
- The managed hosts perhaps cannot run the `ansible.builtin.setup` module for some reason, or you need to install some prerequisite software before gathering facts.

To disable fact gathering for a play, set the `gather_facts` keyword to `no`:

```
---
- name: This play does not automatically gather any facts
  hosts: large_datacenter
  gather_facts: no
```

Even if `gather_facts: no` is set for a play, you can manually gather facts at any time by running a task that uses the `ansible.builtin.setup` module:

```
tasks:
  - name: Manually gather facts
    ansible.builtin.setup:
```

Gathering a Subset of Facts

All facts are gathered by default. You can configure the `ansible.builtin.setup` module to only gather a subset of facts, instead of all facts. For example, to only gather hardware facts, set `gather_subset` to `hardware`:

```
- name: Collect only hardware facts
  ansible.builtin.setup:
    gather_subset:
      - hardware
```

If you want to gather all facts except a certain subset, add an exclamation point (!) in front of the subset name:

```
- name: Collect all facts except for hardware facts
  ansible.builtin.setup:
    gather_subset:
      - !hardware
```

Visit https://docs.ansible.com/ansible/latest/collections/ansible/builtin/setup_module.html#parameter-gather_subset to view possible values for the `gather_subset` parameter.

Creating Custom Facts

You can use *custom facts* to define certain values for managed hosts. Plays can use custom facts to populate configuration files or conditionally run tasks.

Custom facts are stored locally on each managed host. These facts are integrated into the list of standard facts gathered by the `ansible.builtin.setup` module when it runs on the managed host.

You can statically define custom facts in an INI or JSON file, or you can generate them dynamically when you run a play. Dynamic custom facts are gathered via executable scripts, which generate JSON output.

By default, the `ansible.builtin.setup` module loads custom facts from files and scripts in the `etc/ansible/facts.d` directory of each managed host. The name of each file or script must end in `.fact` for it to be used. Dynamic custom fact scripts must output JSON-formatted facts and must be executable.

The following example static custom facts file is written in INI format. An INI-formatted custom facts file contains a top level defined by a section, followed by the key-value pairs of the facts to define:

```
[packages]
web_package = httpd
db_package = mariadb-server

[users]
user1 = joe
user2 = jane
```

You can provide the same facts in JSON format. The following JSON facts are equivalent to the facts specified by the INI format in the preceding example. The JSON data could be stored in a static text file or printed to standard output by an executable script:

```
{
  "packages": {
    "web_package": "httpd",
    "db_package": "mariadb-server"
  },
  "users": {
    "user1": "joe",
    "user2": "jane"
  }
}
```

**Note**

Custom fact files cannot be in YAML format like a playbook. JSON format is the closest equivalent.

The `ansible.builtin.setup` module stores custom facts in the `ansible_facts['ansible_local']` variable. Facts are organized based on the name of the file that defined them. For example, assume that the `/etc/ansible/facts.d/custom.fact` file on the managed host produces the preceding custom facts. In that case, the value of `ansible_facts['ansible_local']['custom']['users']['user1']` is `joe`.

You can inspect the structure of your custom facts by gathering facts and using the `ansible.builtin.debug` module to display the contents of the `ansible_local` variable with a play similar to the following example:

```
- name: Custom fact testing
  hosts: demo1.example.com
  gather_facts: yes

  tasks:
    - name: Display all facts in ansible_local
      ansible.builtin.debug:
        var: ansible_local
```

When you run the play, you might see output similar to the following example:

```
...output omitted...
TASK [Display all facts in ansible_local] ****
ok: [demo1.example.com] => {
  "ansible_local": {
    "custom": {
      "packages": {
        "db_package": "mariadb-server",
        "web_package": "httpd"
      },
      "users": {
        "user1": "joe",
        "user2": "jane"
      }
    }
  }
}
```

```

        "user2": "jane"
    }
}
}
...
...output omitted...

```

You can use custom facts the same way as default facts in playbooks:

```

[user@demo ~]$ cat playbook.yml
---
- hosts: all
  tasks:
    - name: Prints various Ansible facts
      ansible.builtin.debug:
        msg: >
          The package to install on {{ ansible_facts['fqdn'] }}
          is {{ ansible_facts['ansible_local']['custom']['packages']
          ['web_package'] }}

[user@demo ~]$ ansible-navigator run -m stdout playbook.yml
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [demo1.example.com]

TASK [Prints various Ansible facts] ****
ok: [demo1.example.com] => {
    "msg": "The package to install on demo1.example.com  is httpd"
}

PLAY RECAP ****
demo1.example.com    : ok=2      changed=0      unreachable=0      failed=0      skipped=0
                      rescued=0     ignored=0

```

Using Magic Variables

Ansible sets some special variables automatically.

These *magic variables* can also be useful to get information specific to a particular managed host.

Magic variable names are reserved, so you should not define variables with these names.

Four of the most useful magic variables are:

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 yet been gathered for that host.

group_names

Lists all groups that the current managed host is in.

groups

Lists all groups and hosts in the inventory.

inventory_hostname

Contains the hostname for the current managed host as configured in the inventory. This might be different from the hostname reported by facts for various reasons.

One way to get insight into their values is to use the `ansible.builtin.debug` module to display the contents of these variables.

For example, the following task causes every host that runs the play to print out a list of all network interfaces on the `demo2.example.com` host. This task works as long as facts were gathered for `demo2` earlier in the play or by a preceding play in the playbook. It uses the `hostvars` magic variable to access the `ansible_facts['interfaces']` fact for that host.

```
- name: Print list of network interfaces for demo2
  ansible.builtin.debug:
    var: hostvars['demo2.example.com']['ansible_facts']['interfaces']
```

You can use the same approach with regular variables, not only facts. Keep in mind that the preceding task is run by every host in the play, so it would be more efficient to use a different module to apply information gathered from one host to the configuration of each of those other managed hosts.

Remember that you can use the `ansible.builtin.setup` module in a task to refresh gathered facts at any time. However, fact gathering does cause your playbook to take longer to run.

Several other magic variables are also available. For more information, see https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html.



References

Ansible facts – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_vars_facts.html#ansible-facts

ansible.builtin.setup module – Gathers facts about remote hosts – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/setup_module.html

Special Variables – Ansible Documentation

https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html

► Guided Exercise

Managing Facts

In this exercise, you gather Ansible facts from a managed host and use them in plays.

Outcomes

You should be able to:

- Gather facts from a host.
- Create tasks that use the gathered facts.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start data-facts
```

Instructions

- 1. Change into the `/home/student/data-facts` directory.

```
[student@workstation ~]$ cd ~/data-facts
[student@workstation data-facts]$
```

- 2. Use the Ansible `debug` module to view facts. Create a playbook called `display_facts.yml` that contains a play that displays facts for the `webserver` host.

```
---
- name: Display ansible_facts
hosts: webserver
tasks:
  - name: Display facts
    debug:
      var: ansible_facts
```

Use the `ansible-navigator` command to run the `display_facts.yml` playbook. Review the output and observe the values of some variables it displays.

```
[student@workstation data-facts]$ ansible-navigator run \
> -m stdout display_facts.yml
...output omitted...
"system": "Linux",
"system_capabilities": [],
```

```

"system_capabilities_enforced": "False",
"system_vendor": "Red Hat",
"uptime_seconds": 6775,
"user_dir": "/root",
"user_gecos": "root",
"user_gid": 0,
"user_id": "root",
"user_shell": "/bin/bash",
"user_uid": 0,
"userspace_architecture": "x86_64",
"userspace_bits": "64",
"virtualization_role": "guest",
"virtualization_tech_guest": [
    "openstack"
],
"virtualization_tech_host": [
    "kvm"
],
"virtualization_type": "openstack"
}
}

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0

```

- 3. Use the Ansible `debug` module to view specific facts. Create and run a playbook called `display_specific_facts.yml` that contains a play that displays specific facts for the `webserver` host.
- Create a play in the playbook `display_specific_facts.yml` that contains a play to show specific facts for the `webserver` host.

```

---
- name: Display specific ansible_facts
hosts: webserver
tasks:
- name: Display specific facts
  debug:
    msg: >
        Host "{{ ansible_facts['fqdn'] }}" with Python
version "{{ ansible_facts['python_version'] }}" has
"{{ ansible_facts['processor_count'] }}" processors and
"{{ ansible_facts['memtotal_mb'] }}" MiB of total system memory.

```

- 3.2. Use the `ansible-navigator` command to run the `display_specific_facts.yml` playbook and review the output.

```

[student@workstation data-facts]$ ansible-navigator run \
> -m stdout display_specific_facts.yml

PLAY [Display specific ansible_facts] ****
TASK [Gathering Facts] ****

```

```
ok: [servera.lab.example.com]

TASK [Display specific facts] ****
ok: [servera.lab.example.com] => {
    "msg": "Host \"servera.lab.example.com\" with Python version \"3.9.10\" has
    \"1\" processors and \"960\" MiB of total system memory.\n"
}

PLAY RECAP ****
servera.lab.example.com      : ok=2      changed=0      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=0
```

- ▶ 4. Add a play to the `display_specific_facts.yml` playbook that displays the current value of the `ansible_local` variable. Run the playbook. The resulting output shows that the variable is empty or undefined, because no custom facts are set for your managed hosts at this point.
- 4.1. Add a task to the `display_specific_facts.yml` playbook that displays the current value of the `ansible_local` variable. The full playbook should look like the following example:

```
---
- name: Display specific ansible_facts
hosts: webserver
tasks:
  - name: Display specific facts
    debug:
      msg: >
        Host "{{ ansible_facts['fqdn'] }}" with Python
version "{{ ansible_facts['python_version'] }}"
"{{ ansible_facts['processor_count'] }}" processors and
"{{ ansible_facts['memtotal_mb'] }}" MiB of total system memory.

  - name: Display ansible_local variable
    debug:
      msg: The ansible_local variable is set to
"{{ ansible_facts['ansible_local'] }}"
```

- 4.2. Run the `display_specific_facts.yml` playbook again.

```
[student@workstation data-facts]$ ansible-navigator run \
> -m stdout display_specific_facts.yml

...output omitted...

TASK [Display ansible_local variable] ****
ok: [servera.lab.example.com] => {
    "msg": "The ansible_local variable is set to \"{}\""
}

PLAY RECAP ****
servera.lab.example.com      : ok=3      changed=0      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=0
```

- 5. Create the `/etc/ansible/facts.d` directory on `servera`, then create the `custom.fact` file in that directory. The fact file defines the package to install and the service to start on `servera`.

- 5.1. Create the `/etc/ansible/facts.d` directory on `servera`.

```
[student@workstation data-facts]$ ssh servera
[student@servera ~]$ sudo mkdir -p /etc/ansible/facts.d
[sudo] password for student: student
```

- 5.2. Create the `custom.fact` file in the `/etc/ansible/facts.d` directory on `servera`. You need to be `root` in order to create a file in that directory, but make sure that you return to the `student` account after creating and editing the file but before continuing this activity.

The contents of the file should read as follows:

```
[general]
package = httpd
service = httpd
state = started
enabled = true
```

- 6. Create a playbook named `playbook.yml` that contains a single play. That play runs on the managed hosts in the `webserver` host group. It uses custom facts to install a package and ensure a network service is in a particular state on each host in the group.

- 6.1. Create a play in the `playbook.yml` playbook with the following name and `hosts` directive:

```
---
- name: Install Apache and starts the service
  hosts: webserver
```

- 6.2. Create the first task for that play. It should make sure that the latest version of the package referenced by the `ansible_facts['ansible_local']['custom']['general']['package']` custom fact for the managed host is installed.

```
tasks:
- name: Install the required package
  dnf:
    name: "{{ ansible_facts['ansible_local']['custom']['general']
['package'] }}"
    state: latest
```

- 6.3. Create another task that uses the `ansible_facts['ansible_local']['custom']['general']['service']` custom fact to control the specified service.

That task must also use the `ansible_facts['ansible_local']['custom']['general']['state']` custom fact to determine whether or not to start or stop the service, and the `ansible_facts['ansible_local']['custom']['general']['enabled']` custom fact to control whether or not it is started when the system boots.

```

- name: Start the service
  service:
    name: "{{ ansible_facts['ansible_local']['custom']['general']
['service'] }}"
    state: "{{ ansible_facts['ansible_local']['custom']['general']
['state'] }}"
    enabled: "{{ ansible_facts['ansible_local']['custom']['general']
['enabled'] }}"

```

- 6.4. The complete playbook should consist of the following content. Review the playbook contents and ensure that all the tasks are defined.

```

---
- name: Install Apache and starts the service
  hosts: webserver

  tasks:
    - name: Install the required package
      dnf:
        name: "{{ ansible_facts['ansible_local']['custom']['general']
['package'] }}"
        state: latest

    - name: Start the service
      service:
        name: "{{ ansible_facts['ansible_local']['custom']['general']
['service'] }}"
        state: "{{ ansible_facts['ansible_local']['custom']['general']
['state'] }}"
        enabled: "{{ ansible_facts['ansible_local']['custom']['general']
['enabled'] }}"

```

- 7. Verify the syntax of the `playbook.yml` playbook by running the `ansible-navigator run --syntax-check` command. Correct any reported errors before moving to the next step. You should see output similar to the following example:

```

[student@workstation data-facts]$ ansible-navigator run \
> -m stdout playbook.yml --syntax-check

playbook: /home/student/data-facts/playbook.yml

```

- 8. Create and run a playbook called `check_httpd.yml` to verify that the `httpd` service is *not* currently running on servera.

- 8.1. Create and run a playbook called `check_httpd.yml` with the following contents:

```

---
- name: Check httpd status
  hosts: webserver
  tasks:
    - name: Check httpd status

```

```
command: systemctl status httpd
register: result

- name: Display http status
  debug:
    var: result
```

- 8.2. Run the `check_httpd.yml` playbook and verify that the `httpd` service is not currently running on the `servera` machine.

```
[student@workstation data-facts]$ ansible-navigator run \
> -m stdout check_httpd.yml

...output omitted...

TASK [Check httpd status] *****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "msg": "Could not
find the requested service httpd: host"}

PLAY RECAP *****
servera.lab.example.com      : ok=1    changed=0    unreachable=0    failed=1
  skipped=0    rescued=0    ignored=0
Please review the log for errors.
```

- ▶ 9. Run the `playbook.yml` playbook using the `ansible-navigator run` command. Watch the output as Ansible installs the package and then enables the service.

```
[student@workstation data-facts]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Install Apache and start the service] *****

TASK [Gathering Facts] *****
ok: [servera.lab.example.com]

TASK [Install the required package] *****
changed: [servera.lab.example.com]

TASK [Start the service] *****
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com      : ok=3    changed=2    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

- ▶ 10. Run the `check_httpd.yml` playbook again to determine whether the `httpd` service is now running on the `servera` machine.

```
[student@workstation data-facts]$ ansible-navigator run \
> -m stdout check_httpd.yml
...output omitted...
    "stdout_lines": [
        "● httpd.service - The Apache HTTP Server",
        "    Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled;
        vendor preset: disabled)",
        "    Active: active (running) since Tue 2022-06-21 19:09:06 EDT; 22s
        ago",
    ...output omitted...
```

Finish

On the workstation machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish data-facts
```

This concludes the section.

▶ Lab

Managing Variables and Facts

In this lab, you write and run an Ansible Playbook that uses variables, secrets, and facts.

Outcomes

- You should be able to define variables and use facts in a playbook, as well as use variables defined in an encrypted file.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

The `serverb.lab.example.com` managed host is defined in this inventory as a member of the `webserver` host group. A developer has asked you to write an Ansible Playbook to automate the setup of a web server environment on `serverb.lab.example.com`, which controls user access to its website using basic authentication.

The `files` subdirectory contains the following files:

- An `httpd.conf` configuration file for the Apache web service for basic authentication
- A `.htaccess` file, used to control access to the web server's document root directory
- An `htpasswd` file containing credentials for permitted users

```
[student@workstation ~]$ lab start data-review
```

Instructions

1. In the working directory, create the `playbook.yml` playbook. In the playbook, start creating a play to install and configure the web server hosts with an Apache HTTP Server that has basic authentication enabled. Configure the `webserver` host group to contain the managed hosts for the play.

Define the following play variables:

Variable	Values
firewall_pkg	<code>firewalld</code>
firewall_svc	<code>firewalld</code>
web_pkg	<code>httpd</code>
web_svc	<code>httpd</code>
ssl_pkg	<code>mod_ssl</code>
httpdconf_src	<code>files/httpd.conf</code>
httpdconf_dest	<code>/etc/httpd/conf/httpd.conf</code>
htaccess_src	<code>files/.htaccess</code>
secrets_dir	<code>/etc/httpd/secrets</code>
secrets_src	<code>files/htpasswd</code>
secrets_dest	<code>"{{ secrets_dir }}/htpasswd"</code>
web_root	<code>/var/www/html</code>

2. Add a tasks section to the play. Write a task that ensures the latest version of the necessary packages are installed. These packages are defined by the `firewall_pkg`, `web_pkg`, and `ssl_pkg` variables.
3. Add a second task to the play that ensures that the file specified by the `httpdconf_src` variable has been copied (with the `ansible.builtin.copy` module) to the location specified by the `httpdconf_dest` variable on the managed host. The file must be owned by the `root` user and the `root` group. Set `0644` as the file permissions.
4. Add a third task that uses the `ansible.builtin.file` module to create the directory specified by the `secrets_dir` variable on the managed host. This directory holds the password files used for the basic authentication of web services. The file must be owned by the `apache` user and the `apache` group. Set `0500` as the file permissions.
5. Add a fourth task that uses the `ansible.builtin.copy` module to add an `htpasswd` file, used for basic authentication of web users. The source should be defined by the `secrets_src` variable. The destination should be defined by the `secrets_dest` variable. The file must be owned by the `apache` user and group. Set `0400` as the file permissions.
6. Add a fifth task that uses the `ansible.builtin.copy` module to create a `.htaccess` file in the document root directory of the web server. Copy the file specified by the `htaccess_src` variable to `{{ web_root }}/.htaccess`. The file must be owned by the `apache` user and the `apache` group. Set `0400` as the file permissions.
7. Add a sixth task that uses the `ansible.builtin.copy` module to create the web content file, `index.html`, in the directory specified by the `web_root` variable. The file should contain the message `HOSTNAME (IPADDRESS) has been customized by Ansible.`, where `HOSTNAME` is the fully qualified host name of the managed host and `IPADDRESS` is its IPv4 IP address. Use the `content` option with the `ansible.builtin.copy` module to specify the content of the file, and Ansible facts to specify the host name and IP address.

8. Add a seventh task that uses the `ansible.builtin.service` module to enable and start the firewall service on the managed host.
9. Add an eighth task that uses the `ansible.posix.firewalld` module to enable access to the `https` service that is needed for users to access web services on the managed host. This firewall change should be permanent and should take place immediately.
10. Add a final task that uses the `ansible.builtin.service` module to enable and start the web service on the managed host for all configuration changes to take effect. The name of the web service is defined by the `web_svc` variable.
11. Define a second play in the `playbook.yml` file that uses the `workstation` machine as the managed host to test authentication to the web server. It does not need privilege escalation. Define a variable named `web_user` with the value `guest`.
12. Add a directive to the play that adds additional variables from a variable file named `vars/secret.yml`. This file contains a variable named `web_pass` that specifies the password for the web user. You create this file later in the lab.
Define the start of the task list.
13. Add two tasks to the second play.
The first task uses the `ansible.builtin.uri` module to request content from `https://serverb.lab.example.com` using basic authentication. Use the `web_user` and `web_pass` variables to authenticate to the web server. The task should verify a return HTTP status code of 200. Register the task result in a variable named `auth_test`.
Note that the certificate presented by `serverb` is not trusted, so you need to avoid certificate validation.
The second task uses the `ansible.builtin.debug` module to print the content returned from the web server, which is contained in the `auth_test` variable.
14. Create a `vars/secret.yml` file, encrypted with Ansible Vault. Use the password `redhat` to encrypt it. It should set the `web_pass` variable to `redhat`, which is the web user's password.
15. Run the `playbook.yml` playbook. Verify that content is successfully returned from the web server, and that it matches what was configured in an earlier task.

Evaluation

Run the `lab grade data-review` command on `workstation` to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab grade data-review
```

Finish

On the workstation machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish data-review
```

This concludes the section.

► Solution

Managing Variables and Facts

In this lab, you write and run an Ansible Playbook that uses variables, secrets, and facts.

Outcomes

- You should be able to define variables and use facts in a playbook, as well as use variables defined in an encrypted file.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

The `serverb.lab.example.com` managed host is defined in this inventory as a member of the `webserver` host group. A developer has asked you to write an Ansible Playbook to automate the setup of a web server environment on `serverb.lab.example.com`, which controls user access to its website using basic authentication.

The `files` subdirectory contains the following files:

- An `httpd.conf` configuration file for the Apache web service for basic authentication
- A `.htaccess` file, used to control access to the web server's document root directory
- An `htpasswd` file containing credentials for permitted users

```
[student@workstation ~]$ lab start data-review
```

Instructions

1. In the working directory, create the `playbook.yml` playbook. In the playbook, start creating a play to install and configure the web server hosts with an Apache HTTP Server that has basic authentication enabled. Configure the `webserver` host group to contain the managed hosts for the play.

Define the following play variables:

Variable	Values
firewall_pkg	firewalld
firewall_svc	firewalld
web_pkg	httpd
web_svc	httpd
ssl_pkg	mod_ssl
httpdconf_src	files/httpd.conf
httpdconf_dest	/etc/httpd/conf/httpd.conf
htaccess_src	files/.htaccess
secrets_dir	/etc/httpd/secrets
secrets_src	files/htpasswd
secrets_dest	"{{ secrets_dir }}/htpasswd"
web_root	/var/www/html

- 1.1. Change into the /home/student/data-review directory.

```
[student@workstation ~]$ cd ~/data-review
[student@workstation data-review]$
```

- 1.2. Create the playbook.yml playbook file and edit it in a text editor. The beginning of the file should appear as follows:

```
---
- name: install and configure webserver with basic auth
  hosts: webserver
  vars:
    firewall_pkg: firewalld
    firewall_svc: firewalld
    web_pkg: httpd
    web_svc: httpd
    ssl_pkg: mod_ssl
    httpdconf_src: files/httpd.conf
    httpdconf_dest: /etc/httpd/conf/httpd.conf
    htaccess_src: files/.htaccess
    secrets_dir: /etc/httpd/secrets
    secrets_src: files/htpasswd
    secrets_dest: "{{ secrets_dir }}/htpasswd"
    web_root: /var/www/html
```

2. Add a tasks section to the play. Write a task that ensures the latest version of the necessary packages are installed. These packages are defined by the firewall_pkg, web_pkg, and ssl_pkg variables.

- 2.1. Define the beginning of the tasks section by adding the following line to the play:

```
tasks:
```

- 2.2. Add the following lines to the play to define a task that uses the `ansible.builtin.dnf` module to install the required packages:

```
- name: latest version of necessary packages installed
  ansible.builtin.dnf:
    name:
      - "{{ firewall_pkg }}"
      - "{{ web_pkg }}"
      - "{{ ssl_pkg }}"
    state: latest
```

3. Add a second task to the play that ensures that the file specified by the `httpdconf_src` variable has been copied (with the `ansible.builtin.copy` module) to the location specified by the `httpdconf_dest` variable on the managed host. The file must be owned by the `root` user and the `root` group. Set `0644` as the file permissions.

Add the following lines to the play to define a task that uses the `ansible.builtin.copy` module to copy the contents of the file defined by the `httpdconf_src` variable to the location specified by the `httpdconf_dest` variable.

```
- name: configure web service
  ansible.builtin.copy:
    src: "{{ httpdconf_src }}"
    dest: "{{ httpdconf_dest }}"
    owner: root
    group: root
    mode: 0644
```

4. Add a third task that uses the `ansible.builtin.file` module to create the directory specified by the `secrets_dir` variable on the managed host. This directory holds the password files used for the basic authentication of web services. The file must be owned by the `apache` user and the `apache` group. Set `0500` as the file permissions.

Add the following lines to the play to define a task that uses the `ansible.builtin.file` module to create the directory defined by the `secrets_dir` variable.

```
- name: secrets directory exists
  ansible.builtin.file:
    path: "{{ secrets_dir }}"
    state: directory
    owner: apache
    group: apache
    mode: 0500
```

5. Add a fourth task that uses the `ansible.builtin.copy` module to add an `htpasswd` file, used for basic authentication of web users. The source should be defined by the `secrets_src` variable. The destination should be defined by the `secrets_dest` variable. The file must be owned by the `apache` user and group. Set `0400` as the file permissions.

```
- name: htpasswd file exists
ansible.builtin.copy:
  src: "{{ secrets_src }}"
  dest: "{{ secrets_dest }}"
  owner: apache
  group: apache
  mode: 0400
```

6. Add a fifth task that uses the `ansible.builtin.copy` module to create a `.htaccess` file in the document root directory of the web server. Copy the file specified by the `htaccess_src` variable to `{{ web_root }}/.htaccess`. The file must be owned by the `apache` user and the `apache` group. Set `0400` as the file permissions.

Add the following lines to the play to define a task that uses the `ansible.builtin.copy` module to create the `.htaccess` file using the file defined by the `htaccess_src` variable.

```
- name: .htaccess file installed in docroot
ansible.builtin.copy:
  src: "{{ htaccess_src }}"
  dest: "{{ web_root }}/.htaccess"
  owner: apache
  group: apache
  mode: 0400
```

7. Add a sixth task that uses the `ansible.builtin.copy` module to create the web content file, `index.html`, in the directory specified by the `web_root` variable. The file should contain the message `HOSTNAME (IPADDRESS) has been customized by Ansible.`, where `HOSTNAME` is the fully qualified host name of the managed host and `IPADDRESS` is its IPv4 IP address. Use the `content` option with the `ansible.builtin.copy` module to specify the content of the file, and Ansible facts to specify the host name and IP address.

Add the following lines to the play to define a task that uses the `ansible.builtin.copy` module to create the `index.html` file in the directory defined by the `web_root` variable. Populate the file with the content specified using the `ansible_facts['fqdn']` and `ansible_facts['default_ipv4']['address']` Ansible facts retrieved from the managed host.

```
- name: create index.html
ansible.builtin.copy:
  content: "{{ ansible_facts['fqdn'] }} ({{ ansible_facts['default_ipv4'] }}['address']) has been customized by Ansible.\n"
  dest: "{{ web_root }}/index.html"
```

8. Add a seventh task that uses the `ansible.builtin.service` module to enable and start the firewall service on the managed host.

Add the following lines to the play to define a task that uses the `ansible.builtin.service` module to enable and start the firewall service.

```
- name: firewall service enabled and started
ansible.builtin.service:
  name: "{{ firewall_svc }}"
  state: started
  enabled: true
```

9. Add an eighth task that uses the `ansible.posix.firewalld` module to enable access to the `https` service that is needed for users to access web services on the managed host. This firewall change should be permanent and should take place immediately.

Add the following lines to the play to define a task that uses the `ansible.posix.firewalld` module to open the HTTPS port for the web service.

```
- name: open the port for the web server
  ansible.posix.firewalld:
    service: https
    state: enabled
    immediate: true
    permanent: true
```

10. Add a final task that uses the `ansible.builtin.service` module to enable and start the web service on the managed host for all configuration changes to take effect. The name of the web service is defined by the `web_svc` variable.

```
- name: web service enabled and started
  ansible.builtin.service:
    name: "{{ web_svc }}"
    state: started
    enabled: true
```

11. Define a second play in the `playbook.yml` file that uses the `workstation` machine as the managed host to test authentication to the web server. It does not need privilege escalation. Define a variable named `web_user` with the value `guest`.

- 11.1. Add the following line to define the start of a second play. Note that there is no indentation.

```
- name: test web server with basic auth
```

- 11.2. Add the following line to indicate that the play applies to the `workstation` managed host.

```
hosts: workstation
```

- 11.3. Add the following line to disable privilege escalation.

```
become: false
```

- 11.4. Add the following lines to define the `web_user` play variable.

```
vars:
  web_user: guest
```

12. Add a directive to the play that adds additional variables from a variable file named `vars/secret.yml`. This file contains a variable named `web_pass` that specifies the password for the web user. You create this file later in the lab.

Define the start of the task list.

- 12.1. Using the `vars_files` keyword, add the following lines to the play to instruct Ansible to use variables found in the `vars/secret.yml` variable file.

```
vars_files:
  - vars/secret.yml
```

- 12.2. Add the following line to define the beginning of the `tasks` list.

```
tasks:
```

13. Add two tasks to the second play.

The first task uses the `ansible.builtin.uri` module to request content from `https://serverb.lab.example.com` using basic authentication. Use the `web_user` and `web_pass` variables to authenticate to the web server. The task should verify a return HTTP status code of 200. Register the task result in a variable named `auth_test`.

Note that the certificate presented by `serverb` is not trusted, so you need to avoid certificate validation.

The second task uses the `ansible.builtin.debug` module to print the content returned from the web server, which is contained in the `auth_test` variable.

- 13.1. Add the following lines to create the task for verifying the web service from the control node. Be sure to indent the first line with four spaces.

```
- name: connect to web server with basic auth
  ansible.builtin.uri:
    url: https://serverb.lab.example.com
    validate_certs: no
    force_basic_auth: yes
    user: "{{ web_user }}"
    password: "{{ web_pass }}"
    return_content: yes
    status_code: 200
    register: auth_test
```

- 13.2. Create the second task using the `ansible.builtin.debug` module. The content returned from the web server is added to the registered variable as the key `content`.

```
- ansible.builtin.debug:
  var: auth_test.content
```

- 13.3. The completed playbook should consist of the following content:

```
---
- name: install and configure webserver with basic auth
  hosts: webserver
  vars:
    firewall_pkg: firewalld
    firewall_svc: firewalld
    web_pkg: httpd
    web_svc: httpd
    ssl_pkg: mod_ssl
```

```
httpdconf_src: files/httpd.conf
httpdconf_dest: /etc/httpd/conf/httpd.conf
htaccess_src: files/.htaccess
secrets_dir: /etc/httpd/secrets
secrets_src: files/htpasswd
secrets_dest: "{{ secrets_dir }}/htpasswd"
web_root: /var/www/html
tasks:
  - name: latest version of necessary packages installed
    ansible.builtin.dnf:
      name:
        - "{{ firewall_pkg }}"
        - "{{ web_pkg }}"
        - "{{ ssl_pkg }}"
      state: latest

  - name: configure web service
    ansible.builtin.copy:
      src: "{{ httpdconf_src }}"
      dest: "{{ httpdconf_dest }}"
      owner: root
      group: root
      mode: 0644

  - name: secrets directory exists
    ansible.builtin.file:
      path: "{{ secrets_dir }}"
      state: directory
      owner: apache
      group: apache
      mode: 0500

  - name: htpasswd file exists
    ansible.builtin.copy:
      src: "{{ secrets_src }}"
      dest: "{{ secrets_dest }}"
      owner: apache
      group: apache
      mode: 0400

  - name: .htaccess file installed in docroot
    ansible.builtin.copy:
      src: "{{ htaccess_src }}"
      dest: "{{ web_root }}/.htaccess"
      owner: apache
      group: apache
      mode: 0400

  - name: create index.html
    ansible.builtin.copy:
      content: "{{ ansible_facts['fqdn'] }} {{ ansible_facts['default_ipv4']['address'] }} has been customized by Ansible.\n"
      dest: "{{ web_root }}/index.html"

  - name: firewall service enable and started
```

```

ansible.builtin.service:
  name: "{{ firewall_svc }}"
  state: started
  enabled: true

  - name: open the port for the web server
  ansible.posix.firewalld:
    service: https
    state: enabled
    immediate: true
    permanent: true

  - name: web service enabled and started
  ansible.builtin.service:
    name: "{{ web_svc }}"
    state: started
    enabled: true

  - name: test web server with basic auth
  hosts: workstation
  become: false
  vars:
    - web_user: guest
  vars_files:
    - vars/secret.yml
  tasks:
    - name: connect to web server with basic auth
    ansible.builtin.uri:
      url: https://serverb.lab.example.com
      validate_certs: no
      force_basic_auth: yes
      user: "{{ web_user }}"
      password: "{{ web_pass }}"
      return_content: yes
      status_code: 200
      register: auth_test

    - ansible.builtin.debug:
        var: auth_test.content

```

13.4. Save and close the `playbook.yml` file.

- 14.** Create a `vars/secret.yml` file, encrypted with Ansible Vault. Use the password `redhat` to encrypt it. It should set the `web_pass` variable to `redhat`, which is the web user's password.

14.1. Create a subdirectory named `vars` in the working directory.

```
[student@workstation data-review]$ mkdir vars
```

14.2. Create the encrypted variable file, `vars/secret.yml`, using Ansible Vault. Set the password for the encrypted file to `redhat`.

```
[student@workstation data-review]$ ansible-vault create vars/secret.yml  
New Vault password: redhat  
Confirm New Vault password: redhat
```

14.3. Add the following variable definition to the file:

```
web_pass: redhat
```

14.4. Save and close the file.

15. Run the `playbook.yml` playbook. Verify that content is successfully returned from the web server, and that it matches what was configured in an earlier task.

- 15.1. Before running the playbook, verify that its syntax is correct by running `ansible-navigator` with the `--syntax-check` option.

Use `--vault-id @prompt` to be prompted for the Vault password. Enter `redhat` when prompted for the password.

If it reports any errors, correct them before moving to the next step.

You should see output similar to the following:

```
[student@workstation data-review]$ ansible-navigator run -m stdout \  
> --playbook-artifact-enable false \  
> playbook.yml --syntax-check --vault-id @prompt  
Vault password (default): redhat  
  
playbook: /home/student/data-review/playbook.yml
```

- 15.2. Using the `ansible-navigator` command, run the playbook with the `--vault-id @prompt` option. Enter `redhat` when prompted for the password.

```
[student@workstation data-review]$ ansible-navigator run -m stdout \  
> --playbook-artifact-enable false \  
> playbook.yml --vault-id @prompt  
Vault password: redhat  
PLAY [Install and configure webserver with basic auth] *****  
  
...output omitted...  
  
TASK [connect to web server with basic auth] *****  
ok: [workstation]  
  
TASK [debug] *****  
ok: [workstation] => {  
    "auth_test.content": "serverb.lab.example.com (172.25.250.11) has been  
    customized by Ansible.\n"  
}
```

```
PLAY RECAP ****
workstation : ok=3    changed=0    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0
serverb.lab.example.com : ok=10    changed=8    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0
```

Evaluation

Run the `lab grade data-review` command on `workstation` to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab grade data-review
```

Finish

On the workstation machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish data-review
```

This concludes the section.

Summary

- Ansible variables help you reuse values across files in an entire Ansible project.
- You can define variables for hosts and host groups in the inventory file.
- You can define variables for plays and tasks in the playbook or in external files.
- Extra variables are defined on the command line and take precedence over all other variables.
- You can use the `register` keyword to capture the output of a command in a variable.
- Ansible Vault provides one way to protect sensitive data, such as password hashes and private keys that are used by your Ansible Playbooks.
- Ansible facts are variables that Ansible automatically discovers from a managed host.

Chapter 4

Implementing Task Control

Goal

Manage task control, handlers, and task errors in Ansible Playbooks.

Objectives

- Use loops to write efficient tasks and use conditions to control when to run tasks.
- Implement a task that runs only when another task changes the managed host.
- Control what happens when a task fails, and what conditions cause a task to fail.

Sections

- Writing Loops and Conditional Tasks (and Guided Exercise)
- Implementing Handlers (and Guided Exercise)
- Handling Task Failure (and Guided Exercise)

Lab

- Implementing Task Control

Writing Loops and Conditional Tasks

Objectives

- Use loops to write efficient tasks and use conditions to control when to run tasks.

Task Iteration with Loops

Using loops makes it possible to avoid writing multiple tasks that use the same module. For example, instead of writing five tasks to ensure that five users exist, you can write one task that iterates over a list of five users to ensure that they all exist.

To iterate a task over a set of items, you can use the `loop` keyword. You can configure loops to repeat a task using each item in a list, the contents of each of the files in a list, a generated sequence of numbers, or using more complicated structures.

This section covers simple loops that iterate over a list of items. Consult the documentation for more advanced looping scenarios.

Simple Loops

A simple loop iterates a task over a list of items. The `loop` keyword is added to the task, and takes as a value the list of items over which the task should be iterated. The loop variable `item` holds the value used during each iteration.

Consider the following snippet that uses the `ansible.builtin.service` module twice to ensure that two network services are running:

```
- name: Postfix is running
  ansible.builtin.service:
    name: postfix
    state: started

- name: Dovecot is running
  ansible.builtin.service:
    name: dovecot
    state: started
```

These two tasks can be rewritten to use a simple loop so that only one task is needed to ensure that both services are running:

```
- name: Postfix and Dovecot are running
  ansible.builtin.service:
    name: "{{ item }}"
    state: started
  loop:
    - postfix
    - dovecot
```

The loop can use a list provided by a variable.

In the following example, the `mail_services` variable contains the list of services that need to be running.

```
vars:
  mail_services:
    - postfix
    - dovecot

tasks:
  - name: Postfix and Dovecot are running
    ansible.builtin.service:
      name: "{{ item }}"
      state: started
    loop: "{{ mail_services }}"
```

Loops over a List of Dictionaries

The `loop` list does not need to be a list of simple values.

In the following example, each item in the list is actually a dictionary. Each dictionary in the example has two keys, `name` and `groups`, and the value of each key in the current `item` loop variable can be retrieved with the `item['name']` and `item['groups']` variables, respectively.

```
- name: Users exist and are in the correct groups
  user:
    name: "{{ item['name'] }}"
    state: present
    groups: "{{ item['groups'] }}"
  loop:
    - name: jane
      groups: wheel
    - name: joe
      groups: root
```

The outcome of the preceding task is that the user `jane` is present and a member of the group `wheel`, and that the user `joe` is present and a member of the group `root`.

Earlier-style Loop Keywords

Before Ansible 2.5, most playbooks used a different syntax for loops. Multiple loop keywords were provided, which used the `with_` prefix, followed by the name of an Ansible look-up plug-in (an advanced feature not covered in detail in this course). This syntax for looping is very common in existing playbooks, but will probably be deprecated at some point in the future.

Some examples are listed in the following table:

Earlier-style Ansible Loops

Loop keyword	Description
with_items	Behaves the same as the <code>loop</code> keyword for simple lists, such as a list of strings or a list of dictionaries. Unlike <code>loop</code> , if lists of lists are provided to <code>with_items</code> , they are flattened into a single-level list. The <code>item</code> loop variable holds the list item used during each iteration.
with_file	Requires a list of control node file names. The <code>item</code> loop variable holds the content of a corresponding file from the file list during each iteration.
with_sequence	Requires parameters to generate a list of values based on a numeric sequence. The <code>item</code> loop variable holds the value of one of the generated items in the generated sequence during each iteration.

The following playbook shows an example of the `with_items` keyword:

```
vars:
  data:
    - user0
    - user1
    - user2
tasks:
  - name: "with_items"
    ansible.builtin.debug:
      msg: "{{ item }}"
    with_items: "{{ data }}"
```



Important

Since Ansible 2.5, the recommended way to write loops is to use the `loop` keyword.

However, you should still understand the earlier syntax, especially `with_items`, because it is widely used in existing playbooks. You are likely to encounter playbooks and roles that continue to use `with_*` keywords for looping.

Any task using the earlier syntax can be converted to use `loop` in conjunction with Ansible filters. You do not need to know how to use Ansible filters to do this. The Ansible documentation contains a good reference on how to convert the earlier loops to the new syntax, as well as examples of how to loop over items that are not simple lists. See the "Migrating from with_X to loop" [https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html#migrating-from-with-x-to-loop] section of the *Ansible User Guide*.

You might encounter tasks from earlier playbooks that contain `with_*` keywords.

Advanced looping techniques are beyond the scope of this course. All iteration tasks in this course can be implemented with either the `with_items` or the `loop` keyword.

Using Register Variables with Loops

The `register` keyword can also capture the output of a task that loops. The following snippet shows the structure of the `register` variable from a task that loops:

```
[student@workstation loopdemo]$ cat loop_register.yml
---
- name: Loop Register Test
  gather_facts: no
  hosts: localhost
  tasks:
    - name: Looping Echo Task
      ansible.builtin.shell: "echo This is my item: {{ item }}"
      loop:
        - one
        - two
      register: echo_results ①

    - name: Show echo_results variable
      ansible.builtin.debug:
        var: echo_results ②
```

- ①** The `echo_results` variable is registered.
- ②** The contents of the `echo_results` variable are displayed to the screen.

Running the preceding playbook yields the following output:

```
[student@workstation loopdemo]$ ansible-navigator run -m stdout loop_register.yml

PLAY [Loop Register Test] ****
TASK [Looping Echo Task] ****
changed: [localhost] => (item=one)
changed: [localhost] => (item=two)

TASK [Show echo_results variable] ****
ok: [localhost] => {
  "echo_results": {①
    "changed": true,
    "msg": "All items completed",
    "results": [②
      {③
        "ansible_loop_var": "item",
        "changed": true,
        "cmd": "echo This is my item: one",
        "delta": "0:00:00.004519",
        "end": "2022-06-29 17:32:54.065165",
        "failed": false,
        "...output omitted...
        "item": "one",
        "msg": "",
        "rc": 0,
        "start": "2022-06-29 17:32:54.060646",
```

```

        "stderr": "",
        "stderr_lines": [],
        "stdout": "This is my item: one",
        "stdout_lines": [
            "This is my item: one"
        ]
    },
    {❸
        "ansible_loop_var": "item",
        "changed": true,
        "cmd": "echo This is my item: two",
        "delta": "0:00:00.004175",
        "end": "2022-06-29 17:32:54.296940",
        "failed": false,
        ...output omitted...
        "item": "two",
        "msg": "",
        "rc": 0,
        "start": "2022-06-29 17:32:54.292765",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "This is my item: two",
        "stdout_lines": [
            "This is my item: two"
        ]
    }
],❹
"skipped": false
}
}
...output omitted...

```

- ❶ The { character indicates that the start of the echo_results variable is composed of key-value pairs.
- ❷ The results key contains the results from the previous task. The [character indicates the start of a list.
- ❸ The start of task metadata for the first item (indicated by the item key). The output of the echo command is found in the stdout key.
- ❹ The start of task result metadata for the second item.
- ❺ The] character indicates the end of the results list.

In the preceding example, the results key contains a list. In the next example, the playbook is modified so that the second task iterates over this list:

```

[student@workstation loopdemo]$ cat new_loop_register.yml
---
- name: Loop Register Test
  gather_facts: no
  hosts: localhost
  tasks:
    - name: Looping Echo Task

```

```

    ansible.builtin.shell: "echo This is my item: {{ item }}"
    loop:
      - one
      - two
    register: echo_results

    - name: Show stdout from the previous task.
      ansible.builtin.debug:
        msg: "STDOUT from previous task: {{ item['stdout'] }}"
        loop: "{{ echo_results['results'] }}"

```

After running the preceding playbook, you see the following output:

```

PLAY [Loop Register Test] ****
TASK [Looping Echo Task] ****
changed: [localhost] => (item=one)
changed: [localhost] => (item=two)

TASK [Show stdout from the previous task.] ****
ok: [localhost] => (item={'changed': True, 'stdout': 'This is my item: one',
'stderr': '', 'rc': 0, 'cmd': 'echo This is my item: one', 'start': '2022-06-29
17:41:15.558529', 'end': '2022-06-29 17:41:15.563615', 'delta': '0:00:00.005086',
'msg': '', 'invocation': {'module_args': {'_raw_params': 'echo This is my
item: one', '_uses_shell': True, 'warn': False, 'stdin_add_newline': True,
'strip_empty_ends': True, 'argv': None, 'chdir': None, 'executable': None,
'creates': None, 'removes': None, 'stdin': None}}, 'stdout_lines': ['This
is my item: one'], 'stderr_lines': [], 'failed': False, 'item': 'one',
'ansible_loop_var': 'item'}) => {
    "msg": "STDOUT from previous task: This is my item: one"
}
ok: [localhost] => (item={'changed': True, 'stdout': 'This is my item: two',
'stderr': '', 'rc': 0, 'cmd': 'echo This is my item: two', 'start': '2022-06-29
17:41:15.810566', 'end': '2022-06-29 17:41:15.814932', 'delta': '0:00:00.004366',
'msg': '', 'invocation': {'module_args': {'_raw_params': 'echo This is my
item: two', '_uses_shell': True, 'warn': False, 'stdin_add_newline': True,
'strip_empty_ends': True, 'argv': None, 'chdir': None, 'executable': None,
'creates': None, 'removes': None, 'stdin': None}}, 'stdout_lines': ['This
is my item: two'], 'stderr_lines': [], 'failed': False, 'item': 'two',
'ansible_loop_var': 'item'}) => {
    "msg": "STDOUT from previous task: This is my item: two"
}
...output omitted...

```

Running Tasks Conditionally

Ansible can use *conditionals* to run tasks or plays when certain conditions are met. For example, you can use a conditional to determine available memory on a managed host before Ansible installs or configures a service.

Conditionals help you to differentiate between managed hosts and assign them functional roles based on the conditions that they meet. Playbook variables, registered variables, and Ansible facts can all be tested with conditionals. Operators to compare strings, numeric data, and Boolean values are available.

The following scenarios illustrate the use of conditionals in Ansible.

- Define a hard limit in a variable (for example, `min_memory`) and compare it against the available memory on a managed host.
- Capture the output of a command and evaluate it to determine whether a task completed before taking further action. For example, if a program fails, then a batch is skipped.
- Use Ansible facts to determine the managed host network configuration and decide which template file to send (for example, network bonding or trunking).
- Evaluate the number of CPUs to determine how to properly tune a web server.
- Compare a registered variable with a predefined variable to determine if a service changed. For example, test the MD5 checksum of a service configuration file to see if the service is changed.

Conditional Task Syntax

The `when` statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.

One of the simplest conditions that can be tested is whether a Boolean variable is true or false. The `when` statement in the following example causes the task to run only if `run_my_task` is true.

```
---
- name: Simple Boolean Task Demo
  hosts: all
  vars:
    run_my_task: true

  tasks:
    - name: httpd package is installed
      ansible.builtin.dnf:
        name: httpd
      when: run_my_task
```



Note

Boolean variables can have the value `true` or `false`.

In Ansible content, you can express those values in other ways: `True`, `yes`, or `1` are also accepted for `true`; and `False`, `no`, or `0` are also accepted for `false`. You might see `true` and `yes`, or `false` and `no` used interchangeably to express Boolean values in existing Ansible content.

Ansible YAML files are based on the YAML 1.1 standard, but the YAML 1.2 standard specifies that you can only use `true` or `false` to set Boolean values. For this reason, you might see gradual standardization toward using only `true` or `false` for Boolean values in playbooks and other Ansible files, even though the equivalent ways to express those values are still valid. Whether Ansible should eventually use only those ways of expressing Boolean values is an open question and an ongoing discussion in the Ansible community.

**Important**

When using true/false conditions such as in the preceding example, you must be very careful to make sure that your variable is treated by Ansible as a Boolean and not a string.

Starting with Ansible Core 2.12, strings are always treated by `when` conditionals as true Booleans if they contain any content. (The default automation execution environment in Ansible Automation Platform 2.2 uses Ansible Core 2.13.)

Therefore, if the `run_my_task` variable in the preceding example were written as shown in the following example then it would be treated as a string with content and have the Boolean value `true`, and the task would run. This is probably not the behavior that you want.

```
run_my_task: "false"
```

If it had been written as shown in the next example, however, it would be treated as the Boolean value `false` and the task would *not* run:

```
run_my_task: false
```

To ensure that this is the case, you could rewrite the previous `when` condition to convert an accidental string value to a Boolean and to pass Boolean values unchanged:

```
when: run_my_task | bool
```

The next example is a bit more sophisticated, and tests whether the `my_service` variable has a value. If it does, the value of `my_service` is used as the name of the package to install. If the `my_service` variable is not defined, then the task is skipped without an error.

```
---
- name: Test Variable is Defined Demo
  hosts: all
  vars:
    my_service: httpd

  tasks:
    - name: "{{ my_service }} package is installed"
      ansible.builtin.dnf:
        name: "{{ my_service }}"
      when: my_service is defined
```

The following table shows some operations that you can use when working with conditionals:

Example Conditionals

Operation	Example
Equal (value is a string)	<code>ansible_facts['machine'] == "x86_64"</code>

Operation	Example
Equal (value is numeric)	<code>max_memory == 512</code>
Less than	<code>min_memory < 128</code>
Greater than	<code>min_memory > 256</code>
Less than or equal to	<code>min_memory <= 256</code>
Greater than or equal to	<code>min_memory >= 512</code>
Not equal to	<code>min_memory != 512</code>
Variable exists	<code>min_memory is defined</code>
Variable does not exist	<code>min_memory is not defined</code>
Boolean variable is <code>true</code> . The values of <code>1</code> , <code>True</code> , or <code>yes</code> evaluate to <code>true</code> .	<code>memory_available</code>
Boolean variable is <code>false</code> . The values of <code>0</code> , <code>False</code> , or <code>no</code> evaluate to <code>false</code> .	<code>not memory_available</code>
First variable's value is present as a value in second variable's list	<code>ansible_facts['distribution'] in supported_distros</code>

The last entry in the preceding table might be confusing at first. The following example illustrates how it works.

In the example, the `ansible_facts['distribution']` variable is a fact determined during the `Gathering Facts` task, and identifies the managed host's operating system distribution. The `supported_distros` variable was created by the playbook author, and contains a list of operating system distributions that the playbook supports. If the value of `ansible_facts['distribution']` is in the `supported_distros` list, the conditional passes and the task runs.

```
---
- name: Demonstrate the "in" keyword
  hosts: all
  gather_facts: yes
  vars:
    supported_distros:
      - RedHat
      - Fedora
  tasks:
    - name: Install httpd using dnf, where supported
      ansible.builtin.dnf:
        name: http
        state: present
      when: ansible_facts['distribution'] in supported_distros
```

**Important**

Observe the indentation of the `when` statement. Because the `when` statement is not a module variable, it must be placed outside the module by being indented at the top level of the task.

A task is a YAML dictionary, and the `when` statement is one more key in the task, just like the task's name and the module it uses. A common convention places any `when` keyword that might be present after the task's name and the module (and module arguments).

Testing Multiple Conditions

One `when` statement can be used to evaluate multiple conditionals. To do so, conditionals can be combined with either the `and` or `or` keywords, and grouped with parentheses.

The following snippets show some examples of how to express multiple conditions.

- If a conditional statement should be met when either condition is true, then use the `or` statement. For example, the following condition is met if the machine is running either Red Hat Enterprise Linux or Fedora:

```
when: ansible_facts['distribution'] == "RedHat" or ansible_facts['distribution']
      == "Fedora"
```

- With the `and` operation, both conditions have to be true for the entire conditional statement to be met. For example, the following condition is met if the remote host is a Red Hat Enterprise Linux 9.0 host, and the installed kernel is the specified version:

```
when: ansible_facts['distribution_version'] == "9.0" and ansible_facts['kernel']
      == "5.14.0-70.13.1.el9_0.x86_64"
```

The `when` keyword also supports using a list to describe a list of conditions. When a list is provided to the `when` keyword, all the conditionals are combined using the `and` operation. The example below demonstrates another way to combine multiple conditional statements using the `and` operator:

```
when:
  - ansible_facts['distribution_version'] == "9.0"
  - ansible_facts['kernel'] == "5.14.0-70.13.1.el9_0.x86_64"
```

This format improves readability, a key goal of well-written Ansible Playbooks.

- You can express more complex conditional statements by grouping conditions with parentheses. This ensures that they are correctly interpreted.

For example, the following conditional statement is met if the machine is running either Red Hat Enterprise Linux 9 or Fedora 34. This example uses the greater-than character (`>`) so that the long conditional can be split over multiple lines in the playbook, to make it easier to read.

```
when: >
  ( ansible_facts['distribution'] == "RedHat" and
    ansible_facts['distribution_major_version'] == "9" )
  or
  ( ansible_facts['distribution'] == "Fedora" and
    ansible_facts['distribution_major_version'] == "34" )
```

Combining Loops and Conditional Tasks

You can combine loops and conditionals.

In the following example, the `ansible.builtin.dnf` module installs the `mariadb-server` package if there is a file system mounted on `/` with more than 300 MiB free. The `ansible_facts['mounts']` fact is a list of dictionaries, each one representing facts about one mounted file system. The loop iterates over each dictionary in the list, and the conditional statement is not met unless a dictionary is found that represents a mounted file system where both conditions are true.

```
- name: install mariadb-server if enough space on root
  ansible.builtin.dnf:
    name: mariadb-server
    state: latest
  loop: "{{ ansible_facts['mounts'] }}"
  when: item['mount'] == "/" and item['size_available'] > 3000000000
```



Important

When you use `when` with `loop` for a task, the `when` statement is checked for each item.

The following example also combines conditionals and `register` variables. This playbook restarts the `httpd` service only if the `postfix` service is running:

```
---
- name: Restart HTTPD if Postfix is Running
  hosts: all
  tasks:
    - name: Get Postfix server status
      ansible.builtin.command: /usr/bin/systemctl is-active postfix ①
      register: result ②

    - name: Restart Apache HTTPD based on Postfix status
      ansible.builtin.service:
        name: httpd
        state: restarted
      when: result.rc == 0 ③
```

① Is Postfix running?

② Save information on the module's result in a variable named `result`.

- ③ Evaluate the output of the Postfix task. If the exit code of the `systemctl` command is 0, then Postfix is active and this task restarts the `httpd` service.



References

Loops – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html

Tests – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_tests.html

Conditionals – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html

What Makes A Valid Variable Name – Variables – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#what-makes-a-valid-variable-name

For more information on the change to Boolean handling in conditionals in community Ansible 5 (and Ansible Core 2.12) and later, see

https://docs.ansible.com/ansible/latest/porting_guides/porting_guide_5.html#deprecated

► Guided Exercise

Writing Loops and Conditional Tasks

In this exercise, you write a playbook containing tasks that have conditionals and loops.

Outcomes

- Implement Ansible conditionals using the `when` keyword.
- Implement task iteration using the `loop` keyword in conjunction with conditionals.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start control-flow
```

Instructions

- 1. On the `workstation` machine, change to the `/home/student/control-flow` directory.

```
[student@workstation ~]$ cd ~/control-flow  
[student@workstation control-flow]$
```

- 2. The `lab` command created an Ansible configuration file as well as an inventory file. The inventory file contains the `servera.lab.example.com` server in the `database_dev` host group, and the `serverb.lab.example.com` server in the `database_prod` host group. Review the contents of the file before proceeding.

```
[student@workstation control-flow]$ cat inventory  
[database_dev]  
servera.lab.example.com  
  
[database_prod]  
serverb.lab.example.com
```

- 3. Create the `playbook.yml` playbook, which contains a play with two tasks. Use the `database_dev` host group. The first task installs the MariaDB required packages, and the second task ensures that the MariaDB service is running.
- 3.1. Create the `playbook.yml` playbook and define the `mariadb_packages` variable with two values: `mariadb-server` and `python3-PyMySQL`.

```
---
- name: MariaDB server is running
hosts: database_dev
vars:
  mariadb_packages:
    - mariadb-server
    - python3-PyMySQL
```

- 3.2. Define a task that uses the `ansible.builtin.dnf` module and the `mariadb_packages` variable. The task uses the `mariadb_packages` variable to install the required packages.

```
tasks:
- name: MariaDB packages are installed
  ansible.builtin.dnf:
    name: "{{ item }}"
    state: present
  loop: "{{ mariadb_packages }}"
```



Important

Using `loop` is not the most efficient way to install packages. In the preceding code, the `ansible.builtin.dnf` module runs once for each package in the `mariadb_packages` list.

Normally, you should install all the packages as one transaction, by passing the entire list of packages to the module, rather than passing it one module at a time:

```
- name: MariaDB packages are installed
  ansible.builtin.dnf:
    name: "{{ mariadb_packages }}"
    state: present
```

However, other modules like `ansible.builtin.user` do not allow you to do this; you have to pass that module one user to operate upon at a time. In those cases, `loop` is an invaluable tool.

This example is simply meant as a way for you to see how `loop` works.

- 3.3. Define a second task to start the `mariadb` service. The full playbook should consist of the following content:

```
- name: MariaDB server is running
hosts: database_dev
vars:
  mariadb_packages:
    - mariadb-server
    - python3-PyMySQL

tasks:
- name: MariaDB packages are installed
  ansible.builtin.dnf:
```

```

name: "{{ item }}"
state: present
loop: "{{ mariadb_packages }}"

- name: Start MariaDB service
  ansible.builtin.service:
    name: mariadb
    state: started
    enabled: true
  
```

- ▶ 4. Run the playbook and watch the output of the play.

```

[student@workstation control-flow]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [MariaDB server is running] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [MariaDB packages are installed] ****
changed: [servera.lab.example.com] => (item=mariadb-server)
changed: [servera.lab.example.com] => (item=python3-PyMySQL)

TASK [Start MariaDB service] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=3    changed=2    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
  
```

- ▶ 5. Update the first task to only run if the managed host uses Red Hat Enterprise Linux as its operating system. Update the play to use the database_prod host group.

```

- name: MariaDB server is running
  hosts: database_prod
  vars:
...output omitted...
  tasks:
    - name: MariaDB packages are installed
      ansible.builtin.dnf:
        name: "{{ item }}"
        state: present
      loop: "{{ mariadb_packages }}"
      when: ansible_facts['distribution'] == "RedHat"
...output omitted...
  
```

- 6. Run the playbook again and watch the output of the play.

```
[student@workstation control-flow]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [MariaDB server is running] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [MariaDB packages are installed] ****
ok: [serverb.lab.example.com] => (item=mariadb-server)
ok: [serverb.lab.example.com] => (item=python3-PyMySQL)

TASK [Start MariaDB service] ****
ok: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com      : ok=3    changed=0    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

Ansible executes the task because `serverb.lab.example.com` uses Red Hat Enterprise Linux.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish control-flow
```

This concludes the section.

Implementing Handlers

Objectives

- Implement a task that runs only when another task changes the managed host.

Ansible Handlers

Ansible modules are designed to be *idempotent*. This means that if you run a playbook multiple times, the result is always the same. You can run plays and their tasks multiple times, but managed hosts are only changed if those changes are required to get the managed hosts to the desired state.

However, sometimes when a task does make a change to the system, a further task might need to be run. For example, a change to a service's configuration file might then require that the service be reloaded so that the changed configuration takes effect.

Handlers are tasks that respond to a notification triggered by other tasks. Tasks only notify their handlers when the task changes something on a managed host. Each handler is triggered by its name after the play's block of tasks.

If no task notifies the handler by name then the handler does not run. If one or more tasks notify the handler, the handler runs once after all other tasks in the play have completed. Because handlers are tasks, administrators can use the same modules in handlers that they would use for any other task.

Normally, handlers are used to reboot hosts and restart services.



Note

Use unique names for your handlers. When multiple handlers are defined with the same name, only the last handler defined with the shared name runs.

Handlers can be considered as *inactive* tasks that only get triggered when explicitly invoked using a `notify` statement. The following snippet shows how the Apache server is only restarted by the `restart apache` handler when a configuration file is updated and notifies it:

```
tasks:
  - name: copy demo.example.conf configuration template①
    ansible.builtin.template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify: ②
      - restart apache③

handlers: ④
  - name: restart apache⑤
```

```
ansible.builtin.service:⑥
  name: httpd
  state: restarted
```

- ① The task that notifies the handler.
- ② The `notify` statement indicates the task needs to trigger a handler.
- ③ The name of the handler to run.
- ④ The `handlers` keyword indicates the start of the list of handler tasks.
- ⑤ The name of the handler invoked by tasks.
- ⑥ The module to use for the handler.

In the previous example, the `restart apache` handler is triggered when notified by the `template` task that a change happened. A task might call more than one handler in its `notify` section. Ansible treats the `notify` statement as an array and iterates over the handler names:

```
tasks:
  - name: copy demo.example.conf configuration template
    ansible.builtin.template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:
      - restart mysql
      - restart apache

handlers:
  - name: restart mysql
    ansible.builtin.service:
      name: mariadb
      state: restarted

  - name: restart apache
    ansible.builtin.service:
      name: httpd
      state: restarted
```

Describing the Benefits of Using Handlers

As discussed in the Ansible documentation, there are some important things to remember about using handlers:

- Handlers always run in the order specified by the `handlers` section of the play. They do not run in the order in which they are listed by `notify` statements in a task, or in the order in which tasks notify them.
- Handlers normally run after all other tasks in the play complete. A handler called by a task in the `tasks` part of the playbook does not run until *all* tasks under `tasks` have been processed. (Some minor exceptions to this exist.)
- Handler names exist in a per-play namespace. If two handlers are incorrectly given the same name, only one of them runs.

- Even if more than one task notifies a handler, the handler runs one time. If no tasks notify it, the handler does not run.
- If a task that includes a `notify` statement does not report a `changed` result (for example, a package is already installed and the task reports `ok`), the handler is not notified. Ansible notifies handlers only if the task reports the `changed` status.



Important

Handlers are meant to perform an extra action when a task makes a change to a managed host. They should not be used as a replacement for normal tasks.



References

Handlers: running operations on change – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_handlers.html

► Guided Exercise

Implementing Handlers

In this exercise, you implement handlers in playbooks.

Outcomes

- You should be able to define handlers in playbooks and notify them to apply configuration changes.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start control-handlers
```

Instructions

- 1. On the workstation machine, open a new terminal and change to the `/home/student/control-handlers` directory.

```
[student@workstation ~]$ cd ~/control-handlers
[student@workstation control-handlers]$
```

- 2. Edit the `configure_webapp.yml` playbook file. This playbook installs and configures a web application server. When the web application server configuration changes, the playbook triggers a restart of the appropriate service.

- 2.1. Review the `configure_webapp.yml` playbook. It begins with the initialization of some variables:

```
---
- name: Web application server is deployed
  hosts: webapp
  vars:
    packages: ①
      - nginx
      - php-fpm
    web_service: nginx ②
    app_service: php-fpm ③
    resources_dir: /home/student/control-handlers/files ④
    web_config_src: "{{ resources_dir }}/nginx.conf.standard" ⑤
    web_config_dst: /etc/nginx/nginx.conf ⑥
    app_config_src: "{{ resources_dir }}/php-fpm.conf.standard" ⑦
```

```
app_config_dst: /etc/php-fpm.conf ⑧
```

tasks:

- ➊ packages specifies the name of the packages to install for the web application services.
 - ➋ web_service specifies the name of the web server service.
 - ➌ app_service specifies the name of the application server service.
 - ➍ resources_dir specifies the directory where configuration files are located.
 - ➎ web_config_src specifies the location of the web server configuration file to install.
 - ➏ web_config_dst: specifies the location of the installed web server configuration file on the managed hosts.
 - ➐ app_config_src specifies the location of the application server configuration file to install.
 - ➑ web_config_dst: specifies the location of the installed application server configuration file on the managed hosts.
- 2.2. In the `configure_webapp.yml` file, define a task that uses the `ansible.builtin.dnf` module to install the required packages as defined by the `packages` variable.

The task should read as follows:

```
tasks:
- name: "{{ packages }} packages are installed"
  ansible.builtin.dnf:
    name: "{{ packages }}"
    state: present
```

- 2.3. Add two tasks to start and enable the web and application server services. The tasks should read as follows:

```
- name: Make sure the web service is running
  ansible.builtin.service:
    name: "{{ web_service }}"
    state: started
    enabled: true

- name: Make sure the application service is running
  ansible.builtin.service:
    name: "{{ app_service }}"
    state: started
    enabled: true
```

- 2.4. Add a task to download `nginx.conf.standard` to `/etc/nginx/nginx.conf` on the managed host, using the `ansible.builtin.copy` module. Add a condition that notifies the `restart web service` handler to restart the web server service after a configuration file change. The task should read as follows:

```

- name: The {{ web_config_dst }} file has been deployed
  ansible.builtin.copy:
    src: "{{ web_config_src }}"
    dest: "{{ web_config_dst }}"
    force: true
  notify:
    - restart web service

```

- 2.5. Add a task to download `php-fpm.conf.standard` to `/etc/php-fpm.conf` on the managed host, using the `ansible.builtin.copy` module. Add a condition that notifies the `restart app service` handler to restart the application server service after a configuration file change. The task should read as follows:

```

- name: The {{ app_config_dst }} file has been deployed
  ansible.builtin.copy:
    src: "{{ app_config_src }}"
    dest: "{{ app_config_dst }}"
    force: true
  notify:
    - restart app service

```

- 2.6. Add the `handlers` keyword to define the start of the handler tasks. Define the first handler, `restart web service`, which restarts the `nginx` service. The handler should read as follows:

```

handlers:
  - name: restart web service
    ansible.builtin.service:
      name: "{{ web_service }}"
      state: restarted

```

- 2.7. Define the second handler, `restart app service`, which restarts the `php-fpm` service. The handler should read as follows:

```

- name: restart app service
  ansible.builtin.service:
    name: "{{ app_service }}"
    state: restarted

```

The completed playbook should consist of the following content:

```

---
- name: Web application server is deployed
  hosts: webapp
  vars:
    packages:
      - nginx
      - php-fpm
    web_service: nginx
    app_service: php-fpm
    resources_dir: /home/student/control-handlers/files
    web_config_src: "{{ resources_dir }}/nginx.conf.standard"

```

```
web_config_dst: /etc/nginx/nginx.conf
app_config_src: "{{ resources_dir }}/php-fpm.conf.standard"
app_config_dst: /etc/php-fpm.conf

tasks:
  - name: "{{ packages }}" packages are installed
    ansible.builtin.dnf:
      name: "{{ packages }}"
      state: present

  - name: Make sure the web service is running
    ansible.builtin.service:
      name: "{{ web_service }}"
      state: started
      enabled: true

  - name: Make sure the application service is running
    ansible.builtin.service:
      name: "{{ app_service }}"
      state: started
      enabled: true

  - name: The {{ web_config_dst }} file has been deployed
    ansible.builtin.copy:
      src: "{{ web_config_src }}"
      dest: "{{ web_config_dst }}"
      force: true
    notify:
      - restart web service

  - name: The {{ app_config_dst }} file has been deployed
    ansible.builtin.copy:
      src: "{{ app_config_src }}"
      dest: "{{ app_config_dst }}"
      force: true
    notify:
      - restart app service

handlers:
  - name: restart web service
    ansible.builtin.service:
      name: "{{ web_service }}"
      state: restarted

  - name: restart app service
    ansible.builtin.service:
      name: "{{ app_service }}"
      state: restarted
```

- 3. Before running the playbook, verify that its syntax is correct by running `ansible-navigator` with the `--syntax-check` option. Correct any reported errors before moving to the next step. You should see output similar to the following:

```
[student@workstation control-handlers]$ ansible-navigator run \
> -m stdout configure_webapp.yml --syntax-check

playbook: /home/student/control-handlers/configure_webapp.yml
```

- 4. Run the `configure_webapp.yml` playbook. The output shows that the handlers are being executed.

```
[student@workstation control-handlers]$ ansible-navigator run \
> -m stdout configure_webapp.yml

PLAY [Web application server is deployed] *****

TASK [Gathering Facts] *****
ok: [servera.lab.example.com]

TASK [['nginx', 'php-fpm'] packages are installed] *****
changed: [servera.lab.example.com]

TASK [Make sure the web service is running] *****
changed: [servera.lab.example.com]

TASK [Make sure the application service is running] *****
changed: [servera.lab.example.com]

TASK [The /etc/nginx/nginx.conf file has been deployed] *****
changed: [servera.lab.example.com]

TASK [The /etc/php-fpm.conf file has been deployed] *****
changed: [servera.lab.example.com]

RUNNING HANDLER [restart web service] *****
changed: [servera.lab.example.com]

RUNNING HANDLER [restart app service] *****
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com      : ok=8    changed=7    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

- 5. Run the playbook again.

```
[student@workstation control-handlers]$ ansible-navigator run \
> -m stdout configure_webapp.yml

PLAY [Web application server is deployed] *****

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

```
ok: [servera.lab.example.com]

TASK [[nginx', 'php-fpm'] packages are installed] ****
changed: [servera.lab.example.com]

TASK [Make sure the web service is running] ****
changed: [servera.lab.example.com]

TASK [Make sure the application service is running] ****
changed: [servera.lab.example.com]

TASK [The /etc/nginx/nginx.conf file has been deployed] ****
changed: [servera.lab.example.com]

TASK [The /etc/php-fpm.conf file has been deployed] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=6      changed=0      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=0
```

This time the handlers are skipped. If the remote /etc/nginx/nginx.conf configuration file is changed in the future, executing the playbook would trigger the `restart web` service handler but not the `restart app` service handler.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish control-handlers
```

This concludes the section.

Handling Task Failure

Objectives

- Control what happens when a task fails, and what conditions cause a task to fail.

Managing Task Errors in Plays

Ansible evaluates the return code of each task to determine whether the task succeeded or failed. Normally, when a task fails Ansible immediately skips all subsequent tasks.

However, sometimes you might want to have play execution continue even if a task fails. For example, you might expect that a particular task could fail, and you might want to recover by conditionally running some other task. A number of Ansible features can be used to manage task errors.

Ignoring Task Failure

By default, if a task fails, the play is aborted. However, this behavior can be overridden by ignoring failed tasks. You can use the `ignore_errors` keyword in a task to accomplish this.

The following snippet shows how to use `ignore_errors` in a task to continue playbook execution on the host even if the task fails. For example, if the `notapkg` package does not exist then the `ansible.builtin.dnf` module fails, but having `ignore_errors` set to `yes` allows execution to continue.

```
- name: Latest version of notapkg is installed
  ansible.builtin.dnf:
    name: notapkg
    state: latest
    ignore_errors: yes
```

Forcing Execution of Handlers After Task Failure

Normally when a task fails and the play aborts on that host, any handlers that had been notified by earlier tasks in the play do not run. If you set the `force_handlers: yes` keyword on the play, then notified handlers are called even if the play aborted because a later task failed.



Important

If you have `ignore_errors: yes` set on a task or for the task's play, if that task fails the failure is ignored. In that case, the play keeps running and handlers still run, even if you have `force_handlers: no` set, unless some other error causes the play to fail.

The following snippet shows how to use the `force_handlers` keyword in a play to force execution of the notified handler even if a subsequent task fails:

```

---
- hosts: all
force_handlers: yes
tasks:
  - name: a task which always notifies its handler
    ansible.builtin.command: /bin/true
    notify: restart the database

  - name: a task which fails because the package doesn't exist
    ansible.builtin.dnf:
      name: notapkg
      state: latest

handlers:
  - name: restart the database
    ansible.builtin.service:
      name: mariadb
      state: restarted

```



Important

Remember that handlers are notified when a task reports a `changed` result but are not notified when it reports an `ok` or `failed` result.

If you set `force_handlers: yes` on the play, then any handlers that have been notified are run even if a later task failure causes the play to fail. Otherwise, handlers are not run at all when a play fails.

Setting `force_handlers: yes` on a play does not cause handlers to be notified for tasks that report `ok` or `failed`; it only causes the handlers to run that have already been notified before the point at which the play failed.

Specifying Task Failure Conditions

You can use the `failed_when` keyword on a task to specify which conditions indicate that the task has failed. This is often used with command modules that might successfully execute a command, but where the command's output indicates a failure.

For example, you can run a script that outputs an error message and then use that message to define the failed state for the task. The following example shows one way that you can use the `failed_when` keyword in a task:

```

tasks:
  - name: Run user creation script
    ansible.builtin.shell: /usr/local/bin/create_users.sh
    register: command_result
    failed_when: "'Password missing' in command_result.stdout"

```

The `ansible.builtin.fail` module can also be used to force a task failure. You could instead write that example as two tasks:

```

tasks:
  - name: Run user creation script
    ansible.builtin.shell: /usr/local/bin/create_users.sh
    register: command_result
    ignore_errors: yes

  - name: Report script failure
    ansible.builtin.fail:
      msg: "The password is missing in the output"
      when: "'Password missing' in command_result.stdout"

```

You can use the `ansible.builtin.fail` module to provide a clear failure message for the task. This approach also enables delayed failure, which means that you can run intermediate tasks to complete or roll back other changes.

Specifying When a Task Reports "Changed" Results

When a task makes a change to a managed host, it reports the `changed` state and notifies handlers. When a task does not need to make a change, it reports `ok` and does not notify handlers.

Use the `changed_when` keyword to control how a task reports that it has changed something on the managed host. For example, the `ansible.builtin.command` module in the next example validates the `httpd` configuration on a managed host.

This task validates the configuration syntax, but nothing is actually changed on the managed host. Subsequent tasks can use the value of the `httpd_config_status` variable.

It normally would always report `changed` when it runs. To suppress that change report, `changed_when: false` is set so that it only reports `ok` or `failed`.

```

  - name: Validate httpd configuration
    ansible.builtin.command: httpd -t
    changed_when: false
    register: httpd_config_status

```

The following example uses the `ansible.builtin.shell` module and only reports `changed` if the string "Success" is found in the output of the registered variable. If it does report `changed`, then it notifies the handler.

```

tasks:
  - ansible.builtin.shell:
      cmd: /usr/local/bin/upgrade-database
    register: command_result
    changed_when: "'Success' in command_result.stdout"
    notify:
      - restart_database

handlers:
  - name: restart_database
    ansible.builtin.service:
      name: mariadb
      state: restarted

```

Ansible Blocks and Error Handling

In playbooks, *blocks* are clauses that logically group tasks, and can be used to control how tasks are executed. For example, a task block can have a `when` keyword to apply a conditional to multiple tasks:

```
- name: block example
hosts: all
tasks:
  - name: installing and configuring DNF versionlock plugin
    block:
      - name: package needed by dnf
        ansible.builtin.dnf:
          name: python3-dnf-plugin-versionlock
          state: present
      - name: lock version of tzdata
        ansible.builtin.lineinfile:
          dest: /etc/yum/pluginconf.d/versionlock.list
          line: tzdata-2016j-1
          state: present
    when: ansible_distribution == "RedHat"
```

Blocks also allow for error handling in combination with the `rescue` and `always` statements. If any task in a `block` fails, then `rescue` tasks are executed to recover.

After the tasks in the `block` clause run, as well as the tasks in the `rescue` clause if there was a failure, then tasks in the `always` clause run.

To summarize:

- `block`: Defines the main tasks to run.
- `rescue`: Defines the tasks to run if the tasks defined in the `block` clause fail.
- `always`: Defines the tasks that always run independently of the success or failure of tasks defined in the `block` and `rescue` clauses.

The following example shows how to implement a block in a playbook.

```
tasks:
  - name: Upgrade DB
    block:
      - name: upgrade the database
        ansible.builtin.shell:
          cmd: /usr/local/lib/upgrade-database
    rescue:
      - name: revert the database upgrade
        ansible.builtin.shell:
          cmd: /usr/local/lib/revert-database
    always:
      - name: always restart the database
        ansible.builtin.service:
          name: mariadb
          state: restarted
```

The `when` condition on a `block` clause also applies to its `rescue` and `always` clauses if present.



References

Error Handling in Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_error_handling.html

Error Handling – Blocks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_blocks.html#blocks-error-handling

► Guided Exercise

Handling Task Failure

In this exercise, you explore different ways to handle task failure in an Ansible Playbook.

Outcomes

- Ignore failed commands during the execution of playbooks.
- Force execution of handlers.
- Override what constitutes a failure in tasks.
- Override the changed state for tasks.
- Implement block, rescue, and always in playbooks.

Before You Begin

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start control-errors
```

Instructions

- 1. On the workstation machine, change to the /home/student/control-errors directory.

```
[student@workstation ~]$ cd ~/control-errors  
[student@workstation control-errors]$
```

- 2. The lab command created an Ansible configuration file as well as an inventory file, which contains the servera.lab.example.com server in the databases group. Review the file before proceeding.

```
[student@workstation control-errors]$ cat inventory  
[databases]  
servera.lab.example.com
```

- 3. Create a playbook named `playbook.yml` that contains a play with two tasks. Write the first task with a deliberate error to cause failure.
- 3.1. Open the playbook in a text editor. Define three variables: `web_package` with a value of `http`, `db_package` with a value of `mariadb-server`, and `db_service` with a value of `mariadb`. These variables are used to install the required packages and start the server.

The `http` value is an intentional error in the package name. The (intentionally incorrect) should consist of the following content:

```
---  
- name: Task Failure Exercise  
  hosts: databases  
  vars:  
    web_package: http  
    db_package: mariadb-server  
    db_service: mariadb
```

- 3.2. Define two tasks that use the `ansible.builtin.dnf` module and the two variables, `web_package` and `db_package`. The tasks should install the required packages and read as follows:

```
tasks:  
  - name: Install {{ web_package }} package  
    ansible.builtin.dnf:  
      name: "{{ web_package }}"  
      state: present  
  
  - name: Install {{ db_package }} package  
    ansible.builtin.dnf:  
      name: "{{ db_package }}"  
      state: present
```

- 4. Run the playbook and watch the output of the play.

```
[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml  
  
PLAY [Task Failure Exercise] *****  
  
TASK [Gathering Facts] *****  
ok: [servera.lab.example.com]  
  
TASK [Install http package] *****  
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failures":  
  ["No package http available."], "msg": "Failed to install some of the specified  
  packages", "rc": 1, "results": []}  
  
PLAY RECAP *****  
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=1  
  skipped=0    rescued=0    ignored=0  
Please review the log for errors.
```

The task failed because there is no existing package called `http`. Because the first task failed, the second task did not run.

- 5. Update the first task to ignore any errors by adding the `ignore_errors` keyword. The tasks should consist of the following content:

```

tasks:
  - name: Install {{ web_package }} package
    ansible.builtin.dnf:
      name: "{{ web_package }}"
      state: present
      ignore_errors: yes

  - name: Install {{ db_package }} package
    ansible.builtin.dnf:
      name: "{{ db_package }}"
      state: present

```

- 6. Run the playbook again and watch the output of the play.

```

[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml

PLAY [Task Failure Exercise] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install http package] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failures": [
  {"No package http available."}, "msg": "Failed to install some of the specified
  packages", "rc": 1, "results": []}
...ignoring

TASK [Install mariadb-server package] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=3      changed=1      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=1

```

Although the first task failed, Ansible executed the second one.

- 7. In this step, you set up a `block` keyword, so that you can experiment with how they work.

- 7.1. Update the playbook by nesting the first task in a `block` clause. Remove the line that sets `ignore_errors: yes`. The block should consist of the following content:

```

- name: Attempt to set up a webserver
  block:
    - name: Install {{ web_package }} package
      ansible.builtin.dnf:
        name: "{{ web_package }}"
        state: present

```

- 7.2. Nest the task that installs the `mariadb-server` package in a `rescue` clause. If the task listed in the `block` clause fails, then this task runs. The `block` clause should consist of the following content:

```

rescue:
  - name: Install {{ db_package }} package
    ansible.builtin.dnf:
      name: "{{ db_package }}"
      state: present

```

- 7.3. Finally, add an `always` clause to start the database server upon installation using the `ansible.builtin.service` module. The `always` clause should consist of the following content:

```

always:
  - name: Start {{ db_service }} service
    ansible.builtin.service:
      name: "{{ db_service }}"
      state: started

```

- 7.4. The completed task should consist of the following content:

```

tasks:
  - name: Attempt to set up a webserver
    block:
      - name: Install {{ web_package }} package
        ansible.builtin.dnf:
          name: "{{ web_package }}"
          state: present
    rescue:
      - name: Install {{ db_package }} package
        ansible.builtin.dnf:
          name: "{{ db_package }}"
          state: present
    always:
      - name: Start {{ db_service }} service
        ansible.builtin.service:
          name: "{{ db_service }}"
          state: started

```

- 8. Run the playbook again and observe the output.

- 8.1. Run the playbook. The task in the block that makes sure `web_package` is installed fails, which causes the task in the `rescue` block to run. The task in the `always` block then runs.

```

[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml
PLAY [Task Failure Exercise] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install http package] ****

```

Chapter 4 | Implementing Task Control

```
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failures":  
  ["No package http available."], "msg": "Failed to install some of the specified  
  packages", "rc": 1, "results": []}  
  
TASK [Install mariadb-server package] ****  
ok: [servera.lab.example.com]  
  
TASK [Start mariadb service] ****  
changed: [servera.lab.example.com]  
  
PLAY RECAP ****  
servera.lab.example.com : ok=3    changed=1    unreachable=0    failed=0  
  skipped=0   rescued=1   ignored=0
```

- 8.2. Edit the playbook, correcting the value of the `web_package` variable to read `httpd`. This causes the task in the block to succeed the next time you run the playbook.

```
vars:  
  web_package: httpd  
  db_package: mariadb-server  
  db_service: mariadb
```

- 8.3. Run the playbook again. This time, the task in the block does not fail. This causes the task in the `rescue` section to be ignored. The task in the `always` section still runs.

```
[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml  
  
PLAY [Task Failure Exercise] ****  
  
TASK [Gathering Facts] ****  
ok: [servera.lab.example.com]  
  
TASK [Install httpd package] ****  
changed: [servera.lab.example.com]  
  
TASK [Start mariadb service] ****  
ok: [servera.lab.example.com]  
  
PLAY RECAP ****  
servera.lab.example.com : ok=3    changed=1    unreachable=0    failed=0  
  skipped=0   rescued=0   ignored=0
```

- 9. This step explores how to control the condition that causes a task to be reported as "changed" for a managed host.

- 9.1. Edit the playbook to add two tasks to the start of the play, preceding the `block` clause. The first task uses the `ansible.builtin.command` module to run the `date` command and register the result in the `command_result` variable. The second task uses the `ansible.builtin.debug` module to print the standard output of the first task's command.

```
tasks:
  - name: Check local time
    ansible.builtin.command: date
    register: command_result

  - name: Print local time
    ansible.builtin.debug:
      var: command_result.stdout
```

- 9.2. Run the playbook. You should see that the first task, which runs the `ansible.builtin.command` module, reports `changed`, even though it did not change the remote system; it only collected information about the time. That is because the `ansible.builtin.command` module cannot tell the difference between a command that collects data and a command that changes state.

```
[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml

PLAY [Task Failure Exercise] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Check local time] ****
changed: [servera.lab.example.com]

TASK [Print local time] ****
ok: [servera.lab.example.com] => {
    "command_result.stdout": "Tue Jul  5 03:04:51 PM EDT 2022"
}

TASK [Install httpd package] ****
ok: [servera.lab.example.com]

TASK [Start mariadb service] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=5    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

If you run the playbook again, the `Check local time` task returns `changed` again.

- 9.3. That `ansible.builtin.command` task should not report `changed` every time it runs because it is not changing the managed host. Because you know that the task never changes a managed host, add the line `changed_when: false` to the task to suppress the change.

```
tasks:
  - name: Check local time
    ansible.builtin.command: date
    register: command_result
    changed_when: false
```

```
- name: Print local time
  ansible.builtin.debug:
    var: command_result.stdout
```

- 9.4. Run the playbook again and notice that the task now reports ok, but the task is still being run and is still saving the time in the variable.

```
[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml

PLAY [Task Failure Exercise] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Check local time] ****
ok: [servera.lab.example.com]

TASK [Print local time] ****
ok: [servera.lab.example.com] => {
    "command_result.stdout": "Tue Jul  5 03:06:43 PM EDT 2022"
}

TASK [Install httpd package] ****
ok: [servera.lab.example.com]

TASK [Start mariadb service] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=5    changed=0    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

- 10. As a final exercise, edit the playbook to explore how the `failed_when` keyword interacts with tasks.

- 10.1. Edit the `Install {{ web_package }}` package task so that it reports as having failed when `web_package` has the value `httpd`. Because this is the case, the task reports a failure when you run the play.

Be careful with your indentation to ensure that the keyword is correctly set on the task.

```
block:
  - name: Install {{ web_package }} package
    ansible.builtin.dnf:
      name: "{{ web_package }}"
      state: present
    failed_when: web_package == "httpd"
```

- 10.2. Run the playbook.

```
[student@workstation control-errors]$ ansible-navigator run -m stdout playbook.yml

PLAY [Task Failure Exercise] ****
```

```
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Check local time] ****
ok: [servera.lab.example.com]

TASK [Print local time] ****
ok: [servera.lab.example.com] => {
    "command_result.stdout": "Tue Jul  5 03:08:41 PM EDT 2022"
}

TASK [Install httpd package] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false,
"failed_when_result": true, "msg": "Nothing to do", "rc": 0, "results": []}

TASK [Install mariadb-server package] ****
ok: [servera.lab.example.com]

TASK [Start mariadb service] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=5      changed=0      unreachable=0      failed=0
                               skipped=0     rescued=1     ignored=0
```

Look carefully at the output. The `failed_when` keyword changes the status that the task reports *after* the task runs; it does not change the behavior of the task itself.

However, the reported failure might change the behavior of the rest of the play. Because that task was in a block and reported that it failed, the `Install mariadb-server` package task in the block's `rescue` section was run.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish control-errors
```

This concludes the section.

► Lab

Implementing Task Control

In this lab, you install the Apache web server and secure it using `mod_ssl`. You use conditions, handlers, and task failure handling in your playbook to deploy the environment.

Outcomes

- Define conditionals in Ansible Playbooks
- Set up loops that iterate over elements
- Define handlers in playbooks
- Handle task errors.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start control-review
```

Instructions

1. On the `workstation` machine, change to the `/home/student/control-review` directory.
2. The project directory contains a partially completed play in the `playbook.yml` playbook. Under the `#Fail Fast Message` comment, add a task that uses the `ansible.builtin.fail` module. Provide an appropriate name for the task.

This task should only be executed when the remote system does not meet the following minimum requirements:

- Has at least the amount of RAM specified by the `min_ram_mb` variable. The `min_ram_mb` variable is defined in the `vars.yml` file and has a value of 256.
 - Is running Red Hat Enterprise Linux.
3. Under the `#Install all Packages` comment, add a task named `Ensure required packages are present` to install the latest version of any missing packages. Required packages are specified by the `packages` variable, which is defined in the `vars.yml` file.
 4. Under the `#Enable and start services` comment, add a task to start services. All services specified by the `services` variable, which is defined in the `vars.yml` file, should be started and enabled. Provide an appropriate name for the task.
 5. Under the `#Block of config tasks` comment, add a task block to the play. This block contains two tasks:

- A task to ensure that the directory specified by the `ssl_cert_dir` variable exists on the remote host. This directory stores the web server's certificates.
- A task to copy all files specified by the `web_config_files` variable to the remote host. Examine the structure of the `web_config_files` variable in the `vars.yml` file. Configure the task to copy each file to the correct destination on the remote host.

This task should trigger the `restart web service` handler if any of these files are changed on the remote server.

Additionally, a debug task is executed if either of the two tasks above fail. In this case, the task prints the following message: `One or more of the configuration changes failed, but the web service is still active.`

Provide an appropriate name for all tasks.

6. The play configures the remote host to listen for standard HTTPS requests. Under the `#Configure the firewall` comment, add a task to configure `firewalld`. Ensure that the task configures the remote host to accept standard HTTP and HTTPS connections. The configuration changes must be effective immediately and persist after a reboot. Provide an appropriate name for the task.
7. Define the `restart web service` handler.
When triggered, this task should restart the web service defined by the `web_service` variable, defined in the `vars.yml` file.
8. From the `~/control-review` directory, run the `playbook.yml` playbook. The playbook should execute without errors, and trigger the execution of the handler task.
9. Verify that the web server now responds to HTTPS requests, using the self-signed custom certificate to encrypt the connection. The web server should return `Configured` for both HTTP and HTTPS.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade control-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish control-review
```

This concludes the section.

► Solution

Implementing Task Control

In this lab, you install the Apache web server and secure it using `mod_ssl`. You use conditions, handlers, and task failure handling in your playbook to deploy the environment.

Outcomes

- Define conditionals in Ansible Playbooks
- Set up loops that iterate over elements
- Define handlers in playbooks
- Handle task errors.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start control-review
```

Instructions

1. On the `workstation` machine, change to the `/home/student/control-review` directory.

```
[student@workstation ~]$ cd ~/control-review  
[student@workstation control-review]$
```

2. The project directory contains a partially completed play in the `playbook.yml` playbook. Under the `#Fail Fast Message` comment, add a task that uses the `ansible.builtin.fail` module. Provide an appropriate name for the task.

This task should only be executed when the remote system does not meet the following minimum requirements:

- Has at least the amount of RAM specified by the `min_ram_mb` variable. The `min_ram_mb` variable is defined in the `vars.yml` file and has a value of 256.
- Is running Red Hat Enterprise Linux.

The completed task should consist of the following content:

```
tasks:  
  #Fail Fast Message  
  - name: Show Failed System Requirements Message  
    ansible.builtin.fail:  
      msg: "The {{ inventory_hostname }} did not meet minimum reqs."  
      when: >  
        ansible_facts['memtotal_mb'] < min_ram_mb or  
        ansible_facts['distribution'] != "RedHat"
```

3. Under the `#Install all Packages` comment, add a task named `Ensure required packages are present` to install the latest version of any missing packages. Required packages are specified by the `packages` variable, which is defined in the `vars.yml` file.

The completed task should consist of the following content:

```
#Install all Packages  
- name: Ensure required packages are present  
  ansible.builtin.dnf:  
    name: "{{ packages }}"  
    state: latest
```

4. Under the `#Enable and start services` comment, add a task to start services. All services specified by the `services` variable, which is defined in the `vars.yml` file, should be started and enabled. Provide an appropriate name for the task.

The completed task should consist of the following content:

```
#Enable and start services  
- name: Ensure services are started and enabled  
  ansible.builtin.service:  
    name: "{{ item }}"  
    state: started  
    enabled: yes  
  loop: "{{ services }}"
```

5. Under the `#Block of config tasks` comment, add a task block to the play. This block contains two tasks:

- A task to ensure that the directory specified by the `ssl_cert_dir` variable exists on the remote host. This directory stores the web server's certificates.
- A task to copy all files specified by the `web_config_files` variable to the remote host. Examine the structure of the `web_config_files` variable in the `vars.yml` file. Configure the task to copy each file to the correct destination on the remote host.

This task should trigger the `restart web` service handler if any of these files are changed on the remote server.

Additionally, a debug task is executed if either of the two tasks above fail. In this case, the task prints the following message: `One or more of the configuration changes failed, but the web service is still active.`

Provide an appropriate name for all tasks.

The completed task block should consist of the following content:

```
#Block of config tasks
- name: Setting up the SSL cert directory and config files
  block:
    - name: Create SSL cert directory
      ansible.builtin.file:
        path: "{{ ssl_cert_dir }}"
        state: directory

    - name: Copy Config Files
      ansible.builtin.copy:
        src: "{{ item['src'] }}"
        dest: "{{ item['dest'] }}"
        loop: "{{ web_config_files }}"
        notify: restart web service

  rescue:
    - name: Configuration Error Message
      ansible.builtin.debug:
        msg: >
          One or more of the configuration
          changes failed, but the web service
          is still active.
```

6. The play configures the remote host to listen for standard HTTPS requests. Under the `#Configure the firewall` comment, add a task to configure `firewalld`.

Ensure that the task configures the remote host to accept standard HTTP and HTTPS connections. The configuration changes must be effective immediately and persist after a reboot. Provide an appropriate name for the task.

The completed task should consist of the following content:

```
#Configure the firewall
- name: ensure web server ports are open
  ansible.builtin.firewalld:
    service: "{{ item }}"
    immediate: true
```

```

permanent: true
state: enabled
loop:
  - http
  - https

```

7. Define the `restart web service` handler.

When triggered, this task should restart the web service defined by the `web_service` variable, defined in the `vars.yml` file.

Add a `handlers` section to the end of the play:

```

handlers:
  - name: restart web service
    ansible.builtin.service:
      name: "{{ web_service }}"
      state: restarted

```

The completed playbook should consist of the following content:

```

---
- name: Playbook Control Lab
  hosts: webservers
  vars_files: vars.yml
  tasks:
    #Fail Fast Message
    - name: Show Failed System Requirements Message
      ansible.builtin.fail:
        msg: "The {{ inventory_hostname }} did not meet minimum reqs."
        when: >
          ansible_facts['memtotal_mb'] < min_ram_mb or
          ansible_facts['distribution'] != "RedHat"

    #Install all Packages
    - name: Ensure required packages are present
      ansible.builtin.dnf:
        name: "{{ packages }}"
        state: latest

    #Enable and start services
    - name: Ensure services are started and enabled
      ansible.builtin.service:
        name: "{{ item }}"
        state: started
        enabled: yes
      loop: "{{ services }}"

    #Block of config tasks
    - name: Setting up the SSL cert directory and config files
      block:
        - name: Create SSL cert directory
          ansible.builtin.file:
            path: "{{ ssl_cert_dir }}"
            state: directory

```

```

- name: Copy Config Files
  ansible.builtin.copy:
    src: "{{ item['src'] }}"
    dest: "{{ item['dest'] }}"
  loop: "{{ web_config_files }}"
  notify: restart web service

rescue:
  - name: Configuration Error Message
    ansible.builtin.debug:
      msg: >
        One or more of the configuration
        changes failed, but the web service
        is still active.

#Configure the firewall
- name: ensure web server ports are open
  ansible.builtin.firewalld:
    service: "{{ item }}"
    immediate: true
    permanent: true
    state: enabled
  loop:
    - http
    - https

#Add handlers
handlers:
  - name: restart web service
    ansible.builtin.service:
      name: "{{ web_service }}"
      state: restarted

```

8. From the ~/control-review directory, run the `playbook.yml` playbook. The playbook should execute without errors, and trigger the execution of the handler task.

```

[student@workstation control-review]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Playbook Control Lab] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Show Failed System Requirements Message] ****
skipping: [serverb.lab.example.com]

TASK [Ensure required packages are present] ****
changed: [serverb.lab.example.com]

TASK [Ensure services are started and enabled] ****
changed: [serverb.lab.example.com] => (item=httpd)
ok: [serverb.lab.example.com] => (item=firewalld)

```

```

TASK [Create SSL cert directory] *****
changed: [serverb.lab.example.com]

TASK [Copy Config Files] *****
changed: [serverb.lab.example.com] => (item={'src': 'server.key', 'dest': '/etc/httpd/conf.d/ssl'})
changed: [serverb.lab.example.com] => (item={'src': 'server.crt', 'dest': '/etc/httpd/conf.d/ssl'})
changed: [serverb.lab.example.com] => (item={'src': 'ssl.conf', 'dest': '/etc/httpd/conf.d'})
changed: [serverb.lab.example.com] => (item={'src': 'index.html', 'dest': '/var/www/html'})

TASK [ensure web server ports are open] *****
changed: [serverb.lab.example.com] => (item=http)
changed: [serverb.lab.example.com] => (item=https)

RUNNING HANDLER [restart web service] *****
changed: [serverb.lab.example.com]

PLAY RECAP *****
serverb.lab.example.com      : ok=7      changed=6      unreachable=0      failed=0
                               skipped=1    rescued=0    ignored=0

```

- Verify that the web server now responds to HTTPS requests, using the self-signed custom certificate to encrypt the connection. The web server should return **Configured** for both HTTP and HTTPS.

```

[student@workstation control-review]$ curl -k -vvv https://serverb.lab.example.com
*   Trying 172.25.250.11:443...
* Connected to serverb.lab.example.com (172.25.250.11) port 443 (#0)
...output omitted...
< HTTP/1.1 200 OK
< Date: Tue, 05 Jul 2022 19:36:48 GMT
< Server: Apache/2.4.51 (Red Hat Enterprise Linux) OpenSSL/3.0.1
< Last-Modified: Tue, 05 Jul 2022 19:35:30 GMT
< ETag: "24-5e313f48fbb2c"
< Accept-Ranges: bytes
< Content-Length: 36
< Content-Type: text/html; charset=UTF-8
<
Configured for both HTTP and HTTPS.
* Connection #0 to host serverb.lab.example.com left intact

```

Evaluation

As the student user on the workstation machine, use the lab command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade control-review
```

Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish control-review
```

This concludes the section.

Summary

- Loops are used to iterate over a set of values, such as a simple list of strings, or a list of dictionaries.
- Conditionals are used to execute tasks or plays only when certain conditions have been met.
- Handlers are special tasks that execute at the end of the play if notified by other tasks.
- Handlers are only notified when a task reports that it changed something on a managed host.
- Tasks can be configured to handle error conditions by ignoring task failure, forcing handlers to be called even if the task failed, marking a task as failed when it succeeded, or overriding the behavior that causes a task to be marked as changed.
- Blocks are used to group tasks as a unit and to execute other tasks depending upon whether all the tasks in the block succeed.

Chapter 5

Deploying Files to Managed Hosts

Goal

Deploy, manage, and adjust files on hosts managed by Ansible.

Objectives

- Create, install, edit, and remove files on managed hosts, and manage the permissions, ownership, SELinux context, and other characteristics of those files.
- Deploy files to managed hosts that are customized by using Jinja2 templates.

Sections

- Modifying and Copying Files to Hosts (and Guided Exercise)
- Deploying Custom Files with Jinja2 Templates (and Guided Exercise)

Lab

- Deploying Files to Managed Hosts

Modifying and Copying Files to Hosts

Objectives

- Create, install, edit, and remove files on managed hosts, and manage the permissions, ownership, SELinux context, and other characteristics of those files.

Describing File Modules

Most of the commonly used modules related to Linux file management are provided with `ansible-core` in the `ansible.builtin` collection. They perform tasks such as creating, copying, editing, and modifying permissions and other attributes of files. The following table provides a list of frequently used file management modules:

Commonly Used File Modules in `ansible.builtin`

Module name	Module description
<code>blockinfile</code>	Insert, update, or remove a block of multiline text surrounded by customizable marker lines.
<code>copy</code>	Copy a file from the local or remote machine to a location on a managed host. Similar to the <code>file</code> module, the <code>copy</code> module can also set file attributes, including SELinux context.
<code>fetch</code>	This module works like the <code>copy</code> module, but in reverse. This module is used for fetching files from remote machines to the control node and storing them in a file tree, organized by host name.
<code>file</code>	Set attributes such as permissions, ownership, SELinux contexts, and time stamps of regular files, symlinks, hard links, and directories. This module can also create or remove regular files, symlinks, hard links, and directories. A number of other file-related modules support the same options to set attributes as the <code>file</code> module, including the <code>copy</code> module.
<code>lineinfile</code>	Ensure that a particular line is in a file, or replace an existing line using a back-reference regular expression. This module is primarily useful when you want to change a single line in a file.
<code>stat</code>	Retrieve status information for a file, similar to the Linux <code>stat</code> command.

In addition, the `ansible.posix` collection, which is included in the default automation execution environment, provides some additional modules that are useful for file management:

Commonly Used File Modules in ansible.posix

Module name	Module description
patch	Apply patches to files by using GNU patch.
synchronize	A wrapper around the rsync command to simplify common tasks. The synchronize module is not intended to provide access to the full power of the rsync command, but does make the most common invocations easier to implement. You might still need to call the rsync command directly via the run command module depending on your use case.

Automation Examples with Files Modules

The following examples show ways that you can use these modules to automate common file management tasks.

Ensuring a File Exists on Managed Hosts

Use the ansible.builtin.file module to touch a file on managed hosts. This works like the touch command, creating an empty file if it does not exist, and updating its modification time if it does exist. In this example, in addition to touching the file, Ansible ensures that the owning user, group, and permissions of the file are set to specific values.

```
- name: Touch a file and set permissions
  ansible.builtin.file:
    path: /path/to/file
    owner: user1
    group: group1
    mode: 0640
    state: touch
```

Example outcome:

```
[user@host ~]$ ls -l file
-rw-r-----  user1 group1 0 Nov 25 08:00 file
```

Modifying File Attributes

You can use the ansible.builtin.file module to ensure that a new or existing file has the correct permissions or SELinux type as well.

For example, the following file has retained the default SELinux context relative to a user's home directory, which is not the desired context.

```
[user@host ~]$ ls -Z samba_file
-rw-r--r--  owner group unconfined_u:object_r:user_home_t:s0 samba_file
```

The following task ensures that the SELinux context type attribute of the samba_file file is the desired samba_share_t type. This behavior is similar to the Linux chcon command.

```
- name: SELinux type is set to samba_share_t
ansible.builtin.file:
  path: /path/to/samba_file
  setype: samba_share_t
```

Example outcome:

```
[user@host ~]$ ls -Z samba_file
-rw-r--r--. owner group unconfined_u:object_r:samba_share_t:s0 samba_file
```

File attribute parameters are available in multiple file management modules. Use the `ansible-navigator doc` command for additional information, providing the `ansible.builtin.file` or `ansible.builtin.copy` module as an argument.



Note

To set SELinux file contexts persistently in the policy, some options include:

- If you know how to use Ansible roles, you can use the supported `redhat.rhel_system_roles.selinux` role. That is covered in Chapter 7 of the *Red Hat Enterprise Linux Automation with Ansible* (RH294) training course.
- You can use the module `community.general.setcontext` in the community-supported `community.general` Ansible Content Collection.

Copying and Editing Files on Managed Hosts

In this example, the `ansible.builtin.copy` module is used to copy a file located in the Ansible working directory on the control node to selected managed hosts.

By default, this module assumes that `force: yes` is set. That forces the module to overwrite the remote file if it exists but has different contents to the file being copied. If `force: no` is set, then it only copies the file to the managed host if it does not already exist.

```
- name: Copy a file to managed hosts
ansible.builtin.copy:
  src: file
  dest: /path/to/file
```

To retrieve files from managed hosts use the `ansible.builtin.fetch` module. This could be used to retrieve a file such as an SSH public key from a reference system before distributing it to other managed hosts.

```
- name: Retrieve SSH key from reference host
ansible.builtin.fetch:
  src: "/home/{{ user }}/.ssh/id_rsa.pub"
  dest: "files/keys/{{ user }}.pub"
```

To ensure a specific single line of text exists in an existing file, use the `lineinfile` module:

```
- name: Add a line of text to a file
ansible.builtin.lineinfile:
  path: /path/to/file
  line: 'Add this line to the file'
  state: present
```

To add a block of text to an existing file, use the `ansible.builtin.blockinfile` module:

```
- name: Add additional lines to a file
ansible.builtin.blockinfile:
  path: /path/to/file
  block: |
    First line in the additional block of text
    Second line in the additional block of text
  state: present
```



Note

When using the `ansible.builtin.blockinfile` module, commented block markers are inserted at the beginning and end of the block to ensure idempotency.

```
# BEGIN ANSIBLE MANAGED BLOCK
First line in the additional block of text
Second line in the additional block of text
# END ANSIBLE MANAGED BLOCK
```

You can use the `marker` parameter to the module to help ensure that the right comment character or text is being used for the file in question.

Removing a File from Managed Hosts

A basic example to remove a file from managed hosts is to use the `ansible.builtin.file` module with the `state: absent` parameter. The `state` parameter is optional to many modules. You should always make your intentions clear whether you want `state: present` or `state: absent` for several reasons. Some modules support other options as well. It is possible that the default could change at some point, but perhaps most importantly, it makes it easier to understand the state the system should be in based on your task.

```
- name: Make sure a file does not exist on managed hosts
ansible.builtin.file:
  dest: /path/to/file
  state: absent
```

Retrieving the Status of a File on Managed Hosts

The `ansible.builtin.stat` module retrieves facts for a file, similar to the Linux `stat` command. Parameters provide the functionality to retrieve file attributes, determine the checksum of a file, and more.

The `ansible.builtin.stat` module returns a dictionary of values containing the file status data, which allows you to refer to individual pieces of information using separate variables.

The following example registers the results of a `ansible.builtin.stat` module task and then prints the MD5 checksum of the file that it checked. (The more modern SHA256 algorithm is also available; MD5 is being used here for legibility.)

```
- name: Verify the checksum of a file
  ansible.builtin.stat:
    path: /path/to/file
    checksum_algorithm: md5
  register: result

- ansible.builtin.debug
  msg: "The checksum of the file is {{ result.stat.checksum }}"
```

The outcome should be similar to the following:

```
TASK [Get md5 checksum of a file] ****
ok: [hostname]

TASK [debug] ****
ok: [hostname] => {
    "msg": "The checksum of the file is 5f76590425303022e933c43a7f2092a3"
}
```

Information about the values returned by the `ansible.builtin.stat` module are documented in `ansible-navigator doc ansible.builtin.stat`, or you can register a variable and display its contents to see what is available:

```
- name: Examine all stat output of /etc/passwd
  hosts: workstation

  tasks:
    - name: stat /etc/passwd
      ansible.builtin.stat:
        path: /etc/passwd
      register: results

    - name: Display stat results
      debug:
        var: results
```

Synchronizing Files Between the Control Node and Managed Hosts

The `ansible.posix.synchronize` module is a wrapper around the `rsync` tool, which simplifies common file management tasks in your playbooks. The `rsync` tool must be installed on both the local and remote host. By default, when using the `ansible.posix.synchronize` module, the "local host" is the host that the `ansible.posix.synchronize` task originates on (usually the control node), and the "destination host" is the host that `ansible.posix.synchronize` connects to.

The following example synchronizes a file located in the Ansible working directory to the managed hosts:

```
- name: synchronize local file to remote files
ansible.posix.synchronize:
  src: file
  dest: /path/to/file
```

You can use the `ansible.posix.synchronize` module and its many parameters in many different ways, including synchronizing directories. Run the `ansible-navigator doc ansible.posix.synchronize` command for additional parameters and playbook examples.



References

`chmod(1)`, `chown(1)`, `rsync(1)`, `stat(1)` and `touch(1)` man pages

`ansible-navigator doc` command

Ansible documentation – Index of all Modules – `ansible.builtin`

https://docs.ansible.com/ansible/latest/collections/index_module.html#ansible-builtin

► Guided Exercise

Modifying and Copying Files to Hosts

In this exercise, you use standard Ansible modules to create, install, edit, and remove files on managed hosts and manage the permissions, ownership, and SELinux contexts of those files.

Outcomes

- Retrieve files from managed hosts, by host name, and store them locally.
- Create playbooks that use common file management modules from the `ansible.builtin` Ansible Content Collection such as `copy`, `file`, `lineinfile`, and `blockinfile`.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start file-manage
```

Instructions

- 1. As the student user on `workstation`, change to the `/home/student/file-manage` working directory. Create a playbook called `secure_log_backups.yml` in the current working directory. Configure the playbook to use the `ansible.builtin.fetch` module to retrieve the `/var/log/secure` log file from each of the managed hosts and store them on the control node. The playbook should create the `secure-backups` directory with subdirectories named after the host name of each managed host. Store the backup files in their respective subdirectories.

- 1.1. Navigate to the `/home/student/file-manage` working directory.

```
[student@workstation ~]$ cd ~/file-manage  
[student@workstation file-manage]$
```

- 1.2. Create the `secure_log_backups.yml` playbook. It should contain a play to fetch the log files from all managed hosts in the inventory, and the play should connect as the remote `root` user:

```
---  
- name: Use the fetch module to retrieve secure log files  
  hosts: all  
  remote_user: root
```

13. Add a task to the play in the `secure_log_backups.yml` playbook. That task must retrieve the `/var/log/secure` log file from the managed hosts and store it in the `~/file-manage/secure-backups` directory. The `ansible.builtin.fetch` module creates the `~/file-manage/secure-backups` directory if it does not exist. Use the `flat: no` parameter to ensure the default behavior of appending the host name, path, and file name to the destination.

```
tasks:  
  - name: Fetch the /var/log/secure log file from managed hosts  
    ansible.builtin.fetch:  
      src: /var/log/secure  
      dest: secure-backups  
      flat: no
```

14. Before running the playbook, use the `ansible-navigator run --syntax-check` command to verify its syntax. Correct any errors before moving to the next step.

```
[student@workstation file-manage]$ ansible-navigator run \  
> -m stdout secure_log_backups.yml --syntax-check  
playbook: /home/student/file-manage/secure_log_backups.yml
```

15. Use the `ansible-navigator run` command to execute the playbook:

```
[student@workstation file-manage]$ ansible-navigator run \  
> -m stdout secure_log_backups.yml  
  
PLAY [Use the fetch module to retrieve secure log files] *****  
  
TASK [Gathering Facts] *****  
ok: [servera.lab.example.com]  
ok: [serverb.lab.example.com]  
  
TASK [Fetch the /var/log/secure file from managed hosts] *****  
changed: [serverb.lab.example.com]  
changed: [servera.lab.example.com]  
  
PLAY RECAP *****  
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0  
  skipped=0    rescued=0    ignored=0  
serverb.lab.example.com : ok=2    changed=1    unreachable=0    failed=0  
  skipped=0    rescued=0    ignored=0
```

16. Confirm that the playbook retrieved the `/var/log/secure` files from the managed hosts.

```
[student@workstation file-manage]$ tree -F secure-backups  
secure-backups  
├── servera.lab.example.com/  
│   └── var/  
│       └── log/  
│           └── secure  
└── serverb.lab.example.com/
```

```

└── var/
    └── log/
        └── secure

```

- 2. Create the `copy_file.yml` playbook in the current working directory. Configure a play in the playbook that connects as the `root` user and copies the `/home/student/file-manage/files/users.txt` file to all managed hosts.

- 2.1. Edit the `copy_file.yml` playbook so that it contains one play that starts with the following initial content:

```

---
- name: Using the copy module
  hosts: all
  remote_user: root

```

- 2.2. Add a task to the play that uses the `ansible.builtin.copy` module to copy the `/home/student/file-manage/files/users.txt` file to all managed hosts. It must set the following parameters for the `users.txt` file:

Parameter	Values
src	files/users.txt
dest	/home/devops/users.txt
owner	devops
group	devops
mode	u+rwx, g-wx, o-rwx
setype	samba_share_t

```

tasks:
  - name: Copy a file to managed hosts and set attributes
    ansible.builtin.copy:
      src: files/users.txt
      dest: /home/devops/users.txt
      owner: devops
      group: devops
      mode: u+rwx, g-wx, o-rwx
      setype: samba_share_t

```

- 2.3. Use the `ansible-navigator run --syntax-check` command to verify the syntax of the `copy_file.yml` playbook.

```

[student@workstation file-manage]$ ansible-navigator run \
> -m stdout copy_file.yml --syntax-check
playbook: /home/student/file-manage/copy_file.yml

```

- 2.4. Run the playbook:

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout copy_file.yml

PLAY [Using the copy module] ****

TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [Copy a file to managed hosts and set attributes] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=2      changed=1      unreachable=0      failed=0
    skipped=0      rescued=0      ignored=0
serverb.lab.example.com    : ok=2      changed=1      unreachable=0      failed=0
    skipped=0      rescued=0      ignored=0
```

- 2.5. Use the `ls -Z` command as user devops on the `servera` machine to verify the attributes of the `users.txt` file on the managed hosts.

```
[student@workstation file-manage]$ ssh devops@servera 'ls -Z'
unconfined_u:object_r:samba_share_t:s0 users.txt
```

- ▶ 3. In a preceding step, you set the `samba_share_t` SELinux type field for the `users.txt` file. However, you have decided that you want to change the SELinux context on that file to the default context from the SELinux policy of each system.

Create a playbook called `selinux_defaults.yml` in the current working directory. Configure a play in the playbook that uses the `ansible.builtin.file` module to set the default SELinux context on the file `/home/devops/users.txt` for `user`, `role`, `type`, and `level` fields.



Note

To avoid unnecessary changes to the file context by your plays, normally you would also edit the play in `copy_file.yml` to remove the `setype` keyword from the `ansible.builtin.copy` task in its play.

For simplicity, this exercise skips this step.

- 3.1. Create the `selinux_defaults.yml` playbook. It should contain the following play that uses the `ansible.builtin.file` module to set the default context on the file.

```
---
- name: Using the file module to ensure SELinux file context
hosts: all
remote_user: root
tasks:
  - name: SELinux file context is set to defaults
    ansible.builtin.file:
```

```
path: /home/devops/users.txt
seuser: _default
serole: _default
setype: _default
selevel: _default
```

- 3.2. Use the `ansible-navigator run --syntax-check` command to verify the syntax of the `selinux_defaults.yml` playbook.

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout selinux_defaults.yml --syntax-check
playbook: /home/student/file-manage/selinux_defaults.yml
```

- 3.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout selinux_defaults.yml

PLAY [Using the file module to ensure SELinux file context] *****

TASK [Gathering Facts] *****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [SELinux file context is set to defaults] *****
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com    : ok=2      changed=1      unreachable=0      failed=0
skipped=0    rescued=0    ignored=0
serverb.lab.example.com    : ok=2      changed=1      unreachable=0      failed=0
skipped=0    rescued=0    ignored=0
```

- 3.4. Use the `ls -Z` command as user devops on the `servera` machine to verify the default file attributes of `unconfined_u:object_r:user_home_t:s0`.

```
[student@workstation file-manage]$ ssh devops@servera 'ls -Z'
unconfined_u:object_r:user_home_t:s0 users.txt
```

- ▶ 4. Create a playbook called `add_line.yml` in the current working directory. Configure a play in the playbook to use the `ansible.builtin.lineinfile` module to append the `This line was added by the lineinfile module.` to the `/home/devops/users.txt` file on all managed hosts.

- 4.1. Create the `add_line.yml` playbook. It should contain the following play:

```
---
- name: Add text to an existing file
  hosts: all
  remote_user: devops
  tasks:
```

Chapter 5 | Deploying Files to Managed Hosts

```
- name: Add a single line of text to a file
  ansible.builtin.lineinfile:
    path: /home/devops/users.txt
    line: This line was added by the lineinfile module.
    state: present
```

- 4.2. Use `ansible-navigator run --syntax-check` command to verify the syntax of the `add_line.yml` playbook.

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout add_line.yml --syntax-check
playbook: /home/student/file-manage/add_line.yml
```

- 4.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout add_line.yml

PLAY [Add text to an existing file] ****

TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [Add a single line of text to a file] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=2      changed=1      unreachable=0      failed=0
skipped=0      rescued=0      ignored=0
serverb.lab.example.com      : ok=2      changed=1      unreachable=0      failed=0
skipped=0      rescued=0      ignored=0
```

- 4.4. Use the `cat` command as the `devops` user on the `servera` machine to verify the content of the `users.txt` file on the managed hosts.

```
[student@workstation file-manage]$ ssh devops@servera 'cat users.txt'
This line was added by the lineinfile module.
```

- 5. Create a playbook called `add_block.yml` in the current working directory. Configure a play in the playbook to use the `ansible.builtin.blockinfile` module to append the following block of text to the `/home/devops/users.txt` file on all managed hosts.

This block of text consists of two lines.
They have been added by the `blockinfile` module.

- 5.1. Create the `add_block.yml` playbook. Edit it to contain the following play:

```
---
- name: Add block of text to a file
  hosts: all
```

```

remote_user: devops
tasks:
  - name: Add a block of text to an existing file
    ansible.builtin.blockinfile:
      path: /home/devops/users.txt
      block: |
        This block of text consists of two lines.
        They have been added by the blockinfile module.
      state: present

```

- 5.2. Use the `ansible-navigator run --syntax-check` command to verify the syntax of the `add_block.yml` playbook.

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout add_block.yml --syntax-check
playbook: /home/student/file-manage/add_block.yml
```

- 5.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout add_block.yml

PLAY [Add block of text to a file] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [Add a block of text to an existing file] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
serverb.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

- 5.4. Use the `cat` command as the `devops` user on the `servera` machine to verify the correct content of the `/home/devops/users.txt` file on the managed host.

```
[student@workstation file-manage]$ ssh devops@servera 'cat users.txt'
This line was added by the lineinfile module.
# BEGIN ANSIBLE MANAGED BLOCK
This block of text consists of two lines.
They have been added by the blockinfile module.
# END ANSIBLE MANAGED BLOCK
```

- ▶ 6. Create a playbook called `remove_file.yml` in the current working directory. Configure a play in the playbook to use the `ansible.builtin.file` module to remove the `/home/devops/users.txt` file from all managed hosts.

- 6.1. Create the `remove_file.yml` playbook. Edit it to contain the following play:

```
---
- name: Use the file module to remove a file
  hosts: all
  remote_user: devops
  tasks:
    - name: Remove a file from managed hosts
      ansible.builtin.file:
        path: /home/devops/users.txt
        state: absent
```

- 6.2. Use the `ansible-navigator run --syntax-check` command to verify the syntax of the `remove_file.yml` playbook.

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout remove_file.yml --syntax-check
playbook: /home/student/file-manage/remove_file.yml
```

- 6.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-navigator run \
> -m stdout remove_file.yml

PLAY [Use the file module to remove a file] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [Remove a file from managed hosts] ****
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=2    changed=1    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
serverb.lab.example.com    : ok=2    changed=1    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

- 6.4. Use the `ls -l` command as the `devops` user on the `servera` machine to confirm that the `users.txt` file no longer exists on the managed hosts.

```
[student@workstation file-manage]$ ssh devops@servera 'ls -l'
total 0
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish file-manage
```

This concludes the section.

Deploying Custom Files with Jinja2 Templates

Objectives

- Deploy files to managed hosts that are customized by using Jinja2 templates.

Templating Files

The `ansible.builtin` Ansible Content Collection provides a number of modules that can be used to modify existing files. These include `lineinfile` and `blockinfile`, among others. However, they are not always easy to use effectively and correctly.

A much more powerful way to manage files is to *template* them. With this method, you can write a template configuration file that is automatically customized for the managed host when the file is deployed, using Ansible variables and facts. This can be easier to control and is less error-prone.

Introduction to Jinja2

Ansible uses the Jinja2 templating system for template files. Ansible also uses Jinja2 syntax to reference variables in playbooks, so you already know a little about how to use it.

Using Delimiters

Variables and logic expressions are placed between tags, or *delimiters*. When a Jinja2 template is evaluated, the expression `{{ EXPR }}` is replaced with the results of that expression or variable. Jinja2 templates can also use `{% EXPR %}` for special control structures or logic that loops over Jinja2 code or perform tests. You can use the `{# COMMENT #}` syntax to enclose comments that should not appear in the final file.

In the following example of a Jinja2 template file, the first line includes a comment that is not included in the final file. The variable references in the second line are replaced with the values of the system facts being referenced.

```
{# /etc/hosts line #
{{ ansible_facts['default_ipv4']['address'] }}    {{ ansible_facts['hostname'] }}
```

Building a Jinja2 Template

A Jinja2 template is composed of multiple elements: data, variables, and expressions. Those variables and expressions are replaced with their values when the Jinja2 template is rendered. The variables used in the template can be specified in the `vars` section of the playbook. It is possible to use the managed hosts' facts as variables in a template.

Template files are most commonly kept in the `templates` directory of the project for your playbook, and typically are assigned a `.j2` file extension to make it clear that they are Jinja2 template files.

**Note**

A file containing a Jinja2 template does not need to have any specific file extension (for example, .j2). However, providing such a file extension might make it easier for you to remember that it is a template file.

The following example shows how to create a template for /etc/ssh/sshd_config with variables and facts retrieved by Ansible from managed hosts. When the template is deployed by a play, any facts are replaced by their values for the managed host being configured.

```
# {{ ansible_managed }}
# DO NOT MAKE LOCAL MODIFICATIONS TO THIS FILE BECAUSE THEY WILL BE LOST

Port {{ ssh_port }}
ListenAddress {{ ansible_facts['default_ipv4']['address'] }}

HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_ecdsa_key
HostKey /etc/ssh/ssh_host_ed25519_key

SyslogFacility AUTHPRIV

PermitRootLogin {{ root_allowed }}
AllowGroups {{ groups_allowed }}

AuthorizedKeysFile /etc/.rht_authorized_keys .ssh/authorized_keys

PasswordAuthentication {{ passwords_allowed }}

ChallengeResponseAuthentication no

GSSAPIAuthentication yes
GSSAPICleanupCredentials no

UsePAM yes

X11Forwarding yes
UsePrivilegeSeparation sandbox

AcceptEnv LANG LC_CTYPE LC_NUMERIC LC_TIME LC_COLLATE LC_MONETARY LC_MESSAGES
AcceptEnv LC_PAPER LC_NAME LC_ADDRESS LC_TELEPHONE LC_MEASUREMENT
AcceptEnv LC_IDENTIFICATION LC_ALL LANGUAGE
AcceptEnv XMODIFIERS

Subsystem sftp /usr/libexec/openssh/sftp-server
```

Deploying Jinja2 Templates

Jinja2 templates are a powerful tool that you can use to customize configuration files to be deployed on managed hosts. When the Jinja2 template for a configuration file has been created, it can be deployed to managed hosts by using the `ansible.builtin.template` module, which supports the transfer of a local file on the control node to the managed hosts.

To use the `ansible.builtin.template` module, use the following syntax. The value associated with the `src` key specifies the source Jinja2 template, and the value associated with the `dest` key specifies the file to be created on the destination hosts.

```
tasks:
  - name: template render
    ansible.builtin.template:
      src: /tmp/j2-template.j2
      dest: /tmp/dest-config-file.txt
```



Note

The `ansible.builtin.template` module also allows you to specify the owner (the user that owns the file), group, permissions, and SELinux context of the deployed file, just like the `ansible.builtin.file` module. It can also take a `validate` option to run an arbitrary command (such as `visudo -c`) to check the syntax of a file for correctness before templating it into place.

For more details, see `ansible-navigator doc ansible.builtin.template`.

Managing Templated Files

To avoid having other system administrators modify files that are managed by Ansible, it is a good practice to include a comment at the top of the template to indicate that the file should not be manually edited.

One way to do this is to use the "Ansible managed" string set by the `ansible_managed` directive. This is not a normal variable but can be used as one in a template. You can set the value for `ansible_managed` in an `ansible.cfg` file:

```
ansible_managed = Ansible managed
```

To include the `ansible_managed` string inside a Jinja2 template, use the following syntax:

```
{{ ansible_managed }}
```

Control Structures

You can use Jinja2 control structures in template files to reduce repetitive typing, to enter entries for each host in a play dynamically, or conditionally insert text into a file.

Using Loops

Jinja2 uses the `for` statement to provide looping functionality. In the following example, the `users` variable has a list of values. The `user` variable is replaced with all the values in the `users` variable, one value per line.

```
{% for user in users %}
  {{ user }}
{% endfor %}
```

The following example template uses a `for` statement and a conditional to run through all the values in the `users` variable, replacing `myuser` with each value, unless the value is `root`.

```
{# for statement #}
{% for myuser in users if not myuser == "root" %}
User number {{ loop.index }} - {{ myuser }}
{% endfor %}
```

The `loop.index` variable expands to the index number that the loop is currently on. It has a value of 1 the first time the loop executes, and it increments by 1 through each iteration.

As another example, this template also uses a `for` statement. It assumes a `myhosts` variable that contains a list of hosts to be managed has been defined by the inventory being used. If you put the following `for` statement in a Jinja2 template, all hosts in the `myhosts` group from the inventory would be listed in the resulting file.

```
{% for myhost in groups['myhosts'] %}
{{ myhost }}
{% endfor %}
```

For a more practical example, you can use this example to generate an `/etc/hosts` file from host facts dynamically. Assume that you have the following playbook:

```
- name: /etc/hosts is up to date
  hosts: all
  gather_facts: yes
  tasks:
    - name: Deploy /etc/hosts
      ansible.builtin.template:
        src: templates/hosts.j2
        dest: /etc/hosts
```

The following three-line `templates/hosts.j2` template constructs the file from all hosts in the group `all`. (The middle line is extremely long in the template due to the length of the variable names.) It iterates over each host in the group to get three facts for the `/etc/hosts` file.

```
{% for host in groups['all'] %}
{{ hostvars[host]['ansible_facts']['default_ipv4']['address'] }} {{ hostvars[host]
['ansible_facts']['fqdn'] }} {{ hostvars[host]['ansible_facts']['hostname'] }}
{% endfor %}
```

Using Conditionals

Jinja2 uses the `if` statement to provide conditional control. This allows you to put a line in a deployed file if certain conditions are met.

In the following example, the value of the `result` variable is placed in the deployed file only if the value of the `finished` variable is `True`.

```
{% if finished %}
{{ result }}
{% endif %}
```

**Important**

You can use Jinja2 loops and conditionals in Ansible templates, but not in Ansible Playbooks.

Variable Filters

Jinja2 provides filters which change the output format for template expressions, essentially converting the data in a variable to some other format in the file that results from the template.

For example, filters are available for languages such as YAML and JSON. The `to_json` filter formats the expression output using JSON, and the `to_yaml` filter formats the expression output using YAML.

```
{{ output | to_json }}  

{{ output | to_yaml }}
```

Additional filters are available, such as the `to_nice_json` and `to_nice_yaml` filters, which format the expression output in either JSON or YAML human-readable format.

```
{{ output | to_nice_json }}  

{{ output | to_nice_yaml }}
```

Both the `from_json` and `from_yaml` filters expect strings in either JSON or YAML format, respectively.

```
{{ output | from_json }}  

{{ output | from_yaml }}
```

**Note**

Filters are a very powerful concept in Ansible, and are covered in more depth in Chapter 7 of the course *Developing Advanced Automation with Red Hat Ansible Automation Platform* (DO374).

For more information you can also review "Using filters to manipulate data" [https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html] in the *Ansible User Guide*.

**References**

ansible.builtin.template module - Template a file out to a target host – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/template_module.html

Using filters to manipulate data – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html

► Guided Exercise

Deploying Custom Files with Jinja2 Templates

In this exercise, you create a simple template file that your playbook uses to install a customized Message of the Day file on each managed host.

Outcomes

- Build a template file.
- Use the template file in a playbook.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start file-template
```



Note

All the files used during this exercise are available for reference on workstation in the `/home/student/file-template/files` directory.

Instructions

- 1. On workstation, navigate to the `/home/student/file-template` working directory. Review the `inventory` file in the current working directory. This file configures two groups: `webservers` and `workstations`. The `servera.lab.example.com` system is in the `webservers` group, and the `workstation.lab.example.com` system is in the `workstations` group.

- 1.1. Navigate to the `/home/student/file-template` working directory.

```
[student@workstation ~]$ cd ~/file-template  
[student@workstation file-template]$
```

- 1.2. Display the contents of the `inventory` file.

```
[webservers]
servera.lab.example.com

[workstations]
workstation.lab.example.com
```

- ▶ 2. Create a template for the Message of the Day file (`/etc/motd`) and save it as the `motd.j2` file in the current working directory. Include the following variables and facts in the template:

- `ansible_facts['fqdn']`, to insert the FQDN of the managed host.
- `ansible_facts['distribution']` and `ansible_facts['distribution_version']`, to provide Linux distribution information.
- `system_owner`, for the system owner's email. This variable is going to be set by the play in the `motd.yml` playbook, which you edit in the next step.

```
This is the system {{ ansible_facts['fqdn'] }}.
This is a {{ ansible_facts['distribution'] }} version
{{ ansible_facts['distribution_version'] }} system.
Only use this system with permission.
Please report issues to: {{ system_owner }}.
```

- ▶ 3. Create a playbook file named `motd.yml` in the current working directory. Create a play in that file that defines the `system_owner` variable in its `vars` section. The play must have a task that uses the `ansible.builtin.template` module to deploy the `motd.j2` Jinja2 template to the remote file `/etc/motd` on the managed hosts. It must set the owner and group of `/etc/motd` to `root`, and the mode to `0644`.

```
---
- name: Configure SOE
  hosts: all
  remote_user: devops
  become: true
  vars:
    - system_owner: clyde@example.com
  tasks:
    - name: Configure /etc/motd
      ansible.builtin.template:
        src: motd.j2
        dest: /etc/motd
        owner: root
        group: root
        mode: 0644
```

- 4. Before running the playbook, use the `ansible-navigator run --syntax-check` command to verify its syntax. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation file-template]$ ansible-navigator run \
> -m stdout motd.yml --syntax-check
playbook: /home/student/file-template/motd.yml
```

- 5. Run the `motd.yml` playbook.

```
[student@workstation file-template]$ ansible-navigator run \
> -m stdout motd.yml

PLAY [Configure SOE] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
ok: [workstation.lab.example.com]

TASK [Configure /etc/motd] ****
changed: [servera.lab.example.com]
changed: [workstation.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=2    changed=1    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
workstation.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

- 6. From `workstation`, use `ssh` to log in to `servera.lab.example.com` as the `devops` user to verify that the Message of the Day is correctly displayed on your terminal when you log in. Log off when you have finished.

```
[student@workstation file-template]$ ssh devops@servera.lab.example.com
This is the system servera.lab.example.com.
This is a RedHat version 9.0 system.
Only use this system with permission.
Please report issues to: clyde@example.com.
...output omitted...
[devops@servera ~]$ exit
logout
Connection to servera.lab.example.com closed.
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish file-template
```

This concludes the section.

▶ Lab

Deploying Files to Managed Hosts

In this lab, you run a playbook that creates a customized file on your managed hosts by using a Jinja2 template.

Outcomes

- Build a template file.
- Use the template file in a playbook.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start file-review
```



Note

All files used in this exercise are available on `workstation` in the `/home/student/file-review/files` directory.

Instructions

1. Review the `inventory` file in the `/home/student/file-review` directory. This `inventory` file defines the `servers` group, which has the `serverb.lab.example.com` managed host associated with it.
2. Identify the facts on `serverb.lab.example.com` that show its total amount of system memory, and the number of processors it has.
3. In the `/home/student/file-review/templates` directory, create a Jinja2 template file for the Message of the Day (`/etc/motd`), named `motd.j2`.
After this template has been deployed to `serverb.lab.example.com`, when the `devops` user logs in, a message should display on their terminal that shows the system's total memory and processor count. Use the `ansible_facts['memtotal_mb']` and `ansible_facts['processor_count']` facts in the template to provide the system resource information for the message.
4. In the `/home/student/file-review` directory, create a new playbook file called `motd.yml` that contains a new play that runs on all hosts in the inventory. It must log in using the `devops` user on the remote host, and use `become` to enable privilege escalation for the whole play.

The play must have a task that uses the `ansible.builtin.template` module to deploy the `motd.j2` Jinja2 template file to the file `/etc/motd` on the managed hosts. The resulting file must have the `root` user as its owning user and group, and its permissions must be `0644`.

Add an additional task that uses the `ansible.builtin.stat` module to verify that `/etc/motd` exists on the managed hosts and registers its results in a variable. That task must be followed by a task that uses `ansible.builtin.debug` to display the information in that registered variable.

Add a task that uses the `ansible.builtin.copy` module to place `files/issue` into the `/etc/` directory on the managed host, use the same ownership and permissions as `/etc/motd`.

Finally, add a task that uses the `ansible.builtin.file` module to ensure that `/etc/issue.net` is a symbolic link to `/etc/issue` on the managed host.

5. Verify that your playbook contains no syntax errors.
6. Run the `motd.yml` Ansible Playbook.
7. Confirm that the `motd.yml` playbook has run correctly.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade file-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish file-review
```

This concludes the section.

► Solution

Deploying Files to Managed Hosts

In this lab, you run a playbook that creates a customized file on your managed hosts by using a Jinja2 template.

Outcomes

- Build a template file.
- Use the template file in a playbook.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start file-review
```



Note

All files used in this exercise are available on `workstation` in the `/home/student/file-review/files` directory.

Instructions

1. Review the `inventory` file in the `/home/student/file-review` directory. This inventory file defines the `servers` group, which has the `serverb.lab.example.com` managed host associated with it.
 - 1.1. On `workstation`, change to the `/home/student/file-review` directory.

```
[student@workstation ~]$ cd ~/file-review/  
[student@workstation file-review]$
```

- 1.2. Display the content of the `inventory` file.

```
[servers]  
serverb.lab.example.com
```

2. Identify the facts on `serverb.lab.example.com` that show its total amount of system memory, and the number of processors it has.

Create a playbook called `serverb_facts.yml` in the `/home/student/file-review` directory. Edit it to contain a play that uses the `ansible.builtin.debug` module to display a list of all the facts for the `serverb.lab.example.com` managed host. The

`ansible_facts['processor_count']` and `ansible_facts['memtotal_mb']` facts provide information about the resource limits of the managed host.

```
[student@workstation file-review]$ cat serverb_facts.yml
---
- name: Display ansible_facts
  hosts: serverb.lab.example.com
  tasks:
    - name: Display facts
      debug:
        var: ansible_facts

[student@workstation file-review]$ ansible-navigator run \
> -m stdout serverb_facts.yml

PLAY [Display ansible_facts] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Display facts] ****
ok: [serverb.lab.example.com] => {
    "ansible_facts": {
        ...output omitted...
        "memtotal_mb": 960,
        ...output omitted...
        "processor_count": 1,
        ...output omitted...
    }
}

PLAY RECAP ****
serverb.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0
```

- In the `/home/student/file-review/templates` directory, create a Jinja2 template file for the Message of the Day (`/etc/motd`), named `motd.j2`.

After this template has been deployed to `serverb.lab.example.com`, when the `devops` user logs in, a message should display on their terminal that shows the system's total memory and processor count. Use the `ansible_facts['memtotal_mb']` and `ansible_facts['processor_count']` facts in the template to provide the system resource information for the message.

```
System total memory: {{ ansible_facts['memtotal_mb'] }} MiB.
System processor count: {{ ansible_facts['processor_count'] }}
```

- In the `/home/student/file-review` directory, create a new playbook file called `motd.yml` that contains a new play that runs on all hosts in the inventory. It must log in using

the devops user on the remote host, and use become to enable privilege escalation for the whole play.

The play must have a task that uses the `ansible.builtin.template` module to deploy the `motd.j2` Jinja2 template file to the file `/etc/motd` on the managed hosts. The resulting file must have the `root` user as its owning user and group, and its permissions must be `0644`.

Add an additional task that uses the `ansible.builtin.stat` module to verify that `/etc/motd` exists on the managed hosts and registers its results in a variable. That task must be followed by a task that uses `ansible.builtin.debug` to display the information in that registered variable.

Add a task that uses the `ansible.builtin.copy` module to place `files/issue` into the `/etc/` directory on the managed host, use the same ownership and permissions as `/etc/motd`.

Finally, add a task that uses the `ansible.builtin.file` module to ensure that `/etc/issue.net` is a symbolic link to `/etc/issue` on the managed host.

4.1. Create the playbook and configure the `remote_user` and `become` directives.

```
---  
- name: Configure system  
  hosts: all  
  remote_user: devops  
  become: true  
  tasks:
```

4.2. Create a task using the `ansible.builtin.template` module to deploy the `motd.j2` Jinja2 template file to the file `/etc/motd`

```
- name: Configure a custom /etc/motd  
  ansible.builtin.template:  
    src: templates/motd.j2  
    dest: /etc/motd  
    owner: root  
    group: root  
    mode: 0644
```

4.3. Create a task that uses the `ansible.builtin.stat` module to verify that `/etc/motd` exists, and register the results. Add another task using the `ansible.builtin.debug` module to display the registered variable.

```
- name: Check file exists  
  ansible.builtin.stat:  
    path: /etc/motd  
    register: motd  
  
- name: Display stat results  
  ansible.builtin.debug:  
    var: motd
```

4.4. Add a task that uses the `ansible.builtin.copy` module to place `files/issue` into the `/etc/` directory.

```
- name: Copy custom /etc/issue file
ansible.builtin.copy:
  src: files/issue
  dest: /etc/issue
  owner: root
  group: root
  mode: 0644
```

- 4.5. Add a task that uses the `ansible.builtin.file` module to ensure that `/etc/issue.net` is a symbolic link to `/etc/issue`.

```
- name: Ensure /etc/issue.net is a symlink to /etc/issue
ansible.builtin.file:
  src: /etc/issue
  dest: /etc/issue.net
  state: link
  owner: root
  group: root
  force: yes
```

- 4.6. The completed playbook should look as follows.

```
---
- name: Configure system
  hosts: all
  remote_user: devops
  become: true
  tasks:
    - name: Configure a custom /etc/motd
      ansible.builtin.template:
        src: templates/motd.j2
        dest: /etc/motd
        owner: root
        group: root
        mode: 0644

    - name: Check file exists
      ansible.builtin.stat:
        path: /etc/motd
        register: motd

    - name: Display stat results
      ansible.builtin.debug:
        var: motd

    - name: Copy custom /etc/issue file
      ansible.builtin.copy:
        src: files/issue
        dest: /etc/issue
        owner: root
        group: root
        mode: 0644
```

```

- name: Ensure /etc/issue.net is a symlink to /etc/issue
  ansible.builtin.file:
    src: /etc/issue
    dest: /etc/issue.net
    state: link
    owner: root
    group: root
    force: yes
  
```

- Verify that your playbook contains no syntax errors.

Before you run the playbook, use the `ansible-navigator run --syntax-check` command to validate its syntax. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation file-review]$ ansible-navigator run \
> -m stdout motd.yml --syntax-check
playbook: /home/student/file-review/motd.yml
```

- Run the `motd.yml` Ansible Playbook.

```
[student@workstation file-review]$ ansible-navigator run \
> -m stdout motd.yml

PLAY [Configure system] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Configure a custom /etc/motd] ****
changed: [serverb.lab.example.com]

TASK [Check file exists] ****
ok: [serverb.lab.example.com]

TASK [Display stat results] ****
ok: [serverb.lab.example.com] => {
  "motd": {
    "changed": false,
    "failed": false,
    ...output omitted...

TASK [Copy custom /etc/issue file] ****
changed: [serverb.lab.example.com]

TASK [Ensure /etc/issue.net is a symlink to /etc/issue] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com      : ok=6      changed=3      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=0
```

- Confirm that the `motd.yml` playbook has run correctly.

Use ssh to log in to `serverb.lab.example.com` as the `devops` user, and verify that the `/etc/motd` and `/etc/issue` contents are displayed when you log in. Log off when you have finished.

```
[student@workstation file-review]$ ssh devops@serverb.lab.example.com
-----
----- PRIVATE SYSTEM -----
*   Access to this computer system is restricted to authorized users only. *
*
*   Customer information is confidential and must not be disclosed.      *
-----
System total memory: 960 MiB.
System processor count: 1

Register this system with Red Hat Insights: insights-client --register
Create an account or view all your systems at https://red.ht/insights-dashboard
Last login: Thu Jul  7 14:34:04 2022 from 172.25.250.9
[devops@serverb ~]$ logout
```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade file-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish file-review
```

This concludes the section.

Summary

- The file management modules in the `ansible.builtin` and `ansible.posix` Ansible Content Collections enable you to accomplish most tasks related to file management, such as creating, copying, editing, and modifying permissions and other attributes of files.
- Several file management modules can set the permissions mode and SELinux context for files.
- You can use Jinja2 templates to dynamically construct files for deployment.
- A Jinja2 template is usually composed of two elements: variables and expressions. Those variables and expressions are replaced with values when the Jinja2 template is rendered.
- You use the `ansible.builtin.template` module to deploy Jinja2 templates to managed hosts.
- Jinja2 filters transform template expressions from one kind or format of data into another.

Chapter 6

Managing Complex Plays and Playbooks

Goal

Write playbooks for larger, more complex plays and playbooks.

Objectives

- Write sophisticated host patterns to efficiently select hosts for a play.
- Manage large playbooks by importing or including other playbooks or tasks from external files, either unconditionally or based on a conditional test.

Sections

- Selecting Hosts with Host Patterns (and Guided Exercise)
- Including and Importing Files (and Guided Exercise)

Lab

- Managing Complex Plays and Playbooks

Selecting Hosts with Host Patterns

Objectives

- Write sophisticated host patterns to efficiently select hosts for a play.

Referencing Inventory Hosts

Host patterns are used to specify the hosts on which your play runs. In its simplest form, the name of a managed host or a host group in the inventory is a host pattern that specifies that host or host group.

You have already used host patterns in this course. In a play, the `hosts` directive specifies the managed hosts to run the play against.

It is usually easier to control what hosts a play targets by carefully using host patterns and having appropriate inventory groups, instead of setting complex conditionals on the play's tasks. Therefore, it is important to have a robust understanding of host patterns.

The following example inventory is used throughout this section to illustrate host patterns.

```
[student@controlnode ~]$ cat myinventory
web.example.com
data.example.com

[lab]
labhost1.example.com
labhost2.example.com

[test]
test1.example.com
test2.example.com

[datacenter1]
labhost1.example.com
test1.example.com

[datacenter2]
labhost2.example.com
test2.example.com

[datacenter:children]
datacenter1
datacenter2

[new]
192.168.2.1
192.168.2.2
```

To demonstrate how host patterns are resolved, the following examples run `playbook.yml` Ansible Playbook, which contains a play that is edited to have different host patterns to target different subsets of managed hosts from the preceding example inventory.

Managed Hosts

The most basic host pattern is the name of a single managed host listed in the inventory. This specifies that the host is the only one in the inventory that is acted upon by the `ansible-navigator` command.

When the playbook runs, the first `Gathering Facts` task should run on all managed hosts that match the host pattern. A failure during this task causes the managed host to be removed from the play.

You can only use an IP address in a host pattern if it is explicitly listed in the inventory. If the IP address is not listed in the inventory, then you cannot use it to specify the host even if the IP address resolves to that host name in DNS.

The following example shows how a host pattern can be used to reference an IP address contained in an inventory.

```
[student@controlnode ~]$ cat playbook.yml
---
...output omitted...
hosts: 192.168.2.1
...output omitted...

[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****
*****
TASK [Gathering Facts] ****
ok: [192.168.2.1]
...output omitted...
```



Note

One problem with referring to managed hosts by IP address in the inventory is that it can be hard to remember which IP address matches which host for your plays. However, you might have to specify the host by IP address for connection purposes if the host does not have a host name that your execution environment can resolve.

You can point an alias at a particular IP address in your inventory by setting the `ansible_host` host variable. For example, you could have a host in your inventory named `host.example` that you could use for host patterns and inventory groups, and direct connections using that name to the IP address `192.168.2.1` by creating a `host_vars/host.example` file containing the following host variable:

```
ansible_host: 192.168.2.1
```

Specifying Hosts Using a Group

You can use the names of inventory host groups as host patterns. When a group name is used as a host pattern, it specifies that the play acts on the managed hosts that are members of the group.

```
[student@controlnode ~]$ cat playbook.yml
---
...output omitted...
hosts: lab
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [labhost1.example.com]
ok: [labhost2.example.com]
...output omitted...
```

Remember that there is a special group named `all` that matches all managed hosts in the inventory.

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: all
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [labhost2.example.com]
ok: [test2.example.com]
ok: [web.example.com]
ok: [data.example.com]
ok: [labhost1.example.com]
ok: [192.168.2.1]
ok: [test1.example.com]
ok: [192.168.2.2]
```

There is also a special group named `ungrouped`, which includes all managed hosts in the inventory that are not members of any other group:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: ungrouped
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****
```

```
TASK [Gathering Facts] ****
ok: [web.example.com]
ok: [data.example.com]
```

Matching Multiple Hosts with Wildcards

Another method of accomplishing the same thing as the `all` host pattern is to use the asterisk (*) wildcard character, which matches any string. If the host pattern is just a quoted asterisk, then all hosts in the inventory match.

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: '*'
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [labhost2.example.com]
ok: [test2.example.com]
ok: [web.example.com]
ok: [data.example.com]
ok: [labhost1.example.com]
ok: [192.168.2.1]
ok: [test1.example.com]
ok: [192.168.2.2]
```



Important

Some characters that are used in host patterns also have meaning for the shell. If you are using any special wildcards or list characters in an Ansible Playbook, then you must put your host pattern in single quotes to ensure it is parsed correctly.

```
hosts: '!test1.example.com,development'
```

The asterisk character can also be used to match any managed hosts or groups that contain a particular substring.

For example, the following wildcard host pattern matches all inventory names that end in `.example.com`:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: '*.example.com'
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****
```

```
TASK [Gathering Facts] *****
ok: [labhost1.example.com]
ok: [test1.example.com]
ok: [labhost2.example.com]
ok: [test2.example.com]
ok: [web.example.com]
ok: [data.example.com]
```

The following example uses a wildcard host pattern to match the names of hosts or host groups that start with 192.168.2.:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: '192.168.2.*'
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] *****

TASK [Gathering Facts] *****
ok: [192.168.2.1]
ok: [192.168.2.2]
```

The next example uses a wildcard host pattern to match the names of hosts or host groups that begin with datacenter.

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: 'datacenter*'
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] *****

TASK [Gathering Facts] *****
ok: [labhost1.example.com]
ok: [test1.example.com]
ok: [labhost2.example.com]
ok: [test2.example.com]
```

**Important**

The wildcard host patterns match all inventory names, hosts, and host groups. They do not distinguish between names that are DNS names, IP addresses, or groups, which can lead to some unexpected matches.

For example, compare the results of specifying the `datacenter*` host pattern from the preceding example with the results of the `data*` host pattern based on the example inventory:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: 'data*'
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****
TASK [Gathering Facts] ****
ok: [labhost1.example.com]
ok: [test1.example.com]
ok: [labhost2.example.com]
ok: [test2.example.com]
ok: [data.example.com]
```

Lists

Multiple entries in an inventory can be referenced using logical lists. A comma-separated list of host patterns matches all hosts that match any of those host patterns.

If you provide a comma-separated list of managed hosts, then all those managed hosts are targeted:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: labhost1.example.com,test2.example.com,192.168.2.2
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****
TASK [Gathering Facts] ****
ok: [labhost1.example.com]
ok: [test2.example.com]
ok: [192.168.2.2]
```

If you provide a comma-separated list of groups, then all hosts in any of those groups are targeted:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: lab,datacenter1
```

```
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [labhost1.example.com]
ok: [labhost2.example.com]
ok: [test1.example.com]
```

You can also mix managed hosts, host groups, and wildcards, as shown below:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: lab,data*,192.168.2.2
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [labhost1.example.com]
ok: [labhost2.example.com]
ok: [test1.example.com]
ok: [test2.example.com]
ok: [data.example.com]
ok: [192.168.2.2]
```



Note

The colon character (:) can be used instead of a comma. However, the comma is the preferred separator, especially when working with IPv6 addresses as managed host names. You might see the colon syntax in earlier examples.

If an item in a list starts with an ampersand character (&), then hosts must match that item in order to match the host pattern. It operates similarly to a logical AND.

For example, based on our example inventory, the following host pattern matches machines in the `lab` group only if they are also in the `datacenter1` group:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: lab,&datacenter1
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****
```

```
TASK [Gathering Facts] ****
ok: [labhost1.example.com]
```

You could also specify that machines in the `datacenter1` group match only if they are in the `lab` group with the host patterns `&lab`, `datacenter1` or `datacenter1,&lab`.

You can exclude hosts that match a pattern from a list by using the exclamation point or "bang" character (!) in front of the host pattern. This operates like a logical NOT.

This example matches all hosts defined in the `datacenter` group, except `test2.example.com` based on the example inventory:

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: datacenter,!test2.example.com
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [labhost1.example.com]
ok: [test1.example.com]
ok: [labhost2.example.com]
```

The pattern '`!test2.example.com, datacenter`' could have been used in the preceding example to achieve the same result.

The final example shows the use of a host pattern that matches all hosts in the test inventory, except the managed hosts in the `datacenter1` group.

```
[student@controlnode ~]$ cat playbook.yml
...output omitted...
hosts: all,!datacenter1
...output omitted...
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [web.example.com]
ok: [data.example.com]
ok: [labhost2.example.com]
ok: [test2.example.com]
ok: [192.168.2.1]
ok: [192.168.2.2]
```



References

Patterns: targeting hosts and groups – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/intro_patterns.html

How to build your inventory – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html

► Guided Exercise

Selecting Hosts with Host Patterns

In this exercise, you explore how to use host patterns to specify hosts from the inventory for plays. You are provided with several example inventories to explore host patterns.

Outcomes

- Use different host patterns to access various hosts in an inventory.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start projects-host
```

Instructions

- 1. Change into the `/home/student/projects-host` directory, and review the playbook and inventory files in the directory.

```
[student@workstation ~]$ cd ~/projects-host  
[student@workstation projects-host]$
```

- 1.1. List the contents of the directory.

```
[student@workstation projects-host]$ ls  
ansible.cfg inventory1 inventory2 playbook.yml
```

- 1.2. Inspect the first example inventory file, `inventory1`. Observe how the inventory is organized. Identify what hosts and groups are in the inventory, and which domains are used.

```
srv1.example.com  
srv2.example.com  
s1.lab.example.com  
s2.lab.example.com
```

```
[web]  
jupiter.lab.example.com  
saturn.example.com
```

```
[db]  
db1.example.com
```

```
db2.example.com
db3.example.com

[lb]
lb1.lab.example.com
lb2.lab.example.com

[boston]
db1.example.com
jupiter.lab.example.com
lb2.lab.example.com

[london]
db2.example.com
db3.example.com
file1.lab.example.com
lb1.lab.example.com

[dev]
web1.lab.example.com
db3.example.com

[stage]
file2.example.com
db2.example.com

[prod]
lb2.lab.example.com
db1.example.com
jupiter.lab.example.com

[function:children]
web
db
lb
city

[city:children]
boston
london
environments

[environments:children]
dev
stage
prod
new

[new]
172.25.252.23
172.25.252.44
172.25.252.32
```

- 1.3. Inspect the second example inventory file, `inventory2`. Observe how this inventory is organized. Identify what hosts and groups are in the inventory, and which domains are used.

```
workstation.lab.example.com

[london]
servera.lab.example.com

[berlin]
serverb.lab.example.com

[tokyo]
serverc.lab.example.com

[atlanta]
serverd.lab.example.com

[europe:children]
london
berlin
```

- 1.4. Inspect the contents of the playbook, `playbook.yml`. It currently has `db1.example.com` as the host pattern for its play. Observe how that play uses the `ansible.builtin.debug` module to display the name of each managed host.

```
- name: Resolve host patterns
hosts: db1.example.com
gather_facts: no
tasks:
- name: Display managed hosts matching the host pattern
  ansible.builtin.debug:
    msg: "{{ inventory_hostname }}"
```

- ▶ 2. Run the `playbook.yml` playbook using the `inventory1` inventory file and review the output to verify that the `db1.example.com` server is present in the `inventory1` inventory file.

```
[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [db1.example.com] => {
  "msg": "db1.example.com"
}

PLAY RECAP ****
db1.example.com : ok=1    changed=0    unreachable=0    failed=0
                  skipped=0   rescued=0   ignored=0
```

- 3. Modify the host pattern in the playbook to reference an IP address contained in the `inventory1` inventory file. Run the playbook using the `inventory1` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: 172.25.252.44
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [172.25.252.44] => {
    "msg": "172.25.252.44"
}

PLAY RECAP ****
172.25.252.44 : ok=1    changed=0    unreachable=0    failed=0
                  skipped=0   rescued=0   ignored=0
```

- 4. Modify the host pattern to use the `all` group to list all managed hosts in the `inventory1` inventory file. Run the playbook using the `inventory1` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: all
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [srv1.example.com] => {
    "msg": "srv1.example.com"
}
ok: [srv2.example.com] => {
    "msg": "srv2.example.com"
}
ok: [s1.lab.example.com] => {
    "msg": "s1.lab.example.com"
```

```

}

ok: [s2.lab.example.com] => {
    "msg": "s2.lab.example.com"
}

ok: [jupiter.lab.example.com] => {
    "msg": "jupiter.lab.example.com"
}

ok: [saturn.example.com] => {
    "msg": "saturn.example.com"
}

ok: [db1.example.com] => {
    "msg": "db1.example.com"
}

ok: [db2.example.com] => {
    "msg": "db2.example.com"
}

ok: [db3.example.com] => {
    "msg": "db3.example.com"
}

ok: [lb1.lab.example.com] => {
    "msg": "lb1.lab.example.com"
}

ok: [lb2.lab.example.com] => {
    "msg": "lb2.lab.example.com"
}

ok: [file1.lab.example.com] => {
    "msg": "file1.lab.example.com"
}

ok: [web1.lab.example.com] => {
    "msg": "web1.lab.example.com"
}

ok: [file2.example.com] => {
    "msg": "file2.example.com"
}

ok: [172.25.252.23] => {
    "msg": "172.25.252.23"
}

ok: [172.25.252.44] => {
    "msg": "172.25.252.44"
}

ok: [172.25.252.32] => {
    "msg": "172.25.252.32"
}

PLAY RECAP ****
172.25.252.23      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
172.25.252.32      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
172.25.252.44      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
db1.example.com     : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
db2.example.com     : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0

```

```

db3.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
file1.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
file2.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
jupiter.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
lb1.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
lb2.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
s1.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
s2.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
saturn.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
srv1.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
srv2.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0
web1.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0 ignored=0

```

- ▶ 5. Modify the host pattern to use the asterisk (*) character to list all hosts that end in .example.com in the inventory1 inventory file. Run the playbook using the inventory1 inventory file.

```

[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: '*example.com'
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"
[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [srv1.example.com] => {
    "msg": "srv1.example.com"
}
ok: [srv2.example.com] => {
    "msg": "srv2.example.com"
}
ok: [s1.lab.example.com] => {
    "msg": "s1.lab.example.com"
}

```

```

ok: [s2.lab.example.com] => {
    "msg": "s2.lab.example.com"
}
ok: [jupiter.lab.example.com] => {
    "msg": "jupiter.lab.example.com"
}
ok: [saturn.example.com] => {
    "msg": "saturn.example.com"
}
ok: [db1.example.com] => {
    "msg": "db1.example.com"
}
ok: [db2.example.com] => {
    "msg": "db2.example.com"
}
ok: [db3.example.com] => {
    "msg": "db3.example.com"
}
ok: [lb1.lab.example.com] => {
    "msg": "lb1.lab.example.com"
}
ok: [lb2.lab.example.com] => {
    "msg": "lb2.lab.example.com"
}
ok: [file1.lab.example.com] => {
    "msg": "file1.lab.example.com"
}
ok: [web1.lab.example.com] => {
    "msg": "web1.lab.example.com"
}
ok: [file2.example.com] => {
    "msg": "file2.example.com"
}

PLAY RECAP ****
db1.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
db2.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
db3.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
file1.lab.example.com: ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
file2.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
jupiter.lab.example.com: ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
lb1.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
lb2.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
s1.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
s2.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0

```

```

saturn.example.com      : ok=1    changed=0    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0
srv1.example.com       : ok=1    changed=0    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0
srv2.example.com       : ok=1    changed=0    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0
web1.lab.example.com  : ok=1    changed=0    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0

```

- ▶ 6. As you can see in the output of the preceding command, the `*.example.com` domain contains 14 hosts. Modify the host pattern so that hosts in the `*.lab.example.com` domain are ignored. Run the playbook using the `inventory1` inventory file.

```

[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: '*.example.com, !*.lab.example.com'
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"
[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [srv1.example.com] => {
    "msg": "srv1.example.com"
}
ok: [srv2.example.com] => {
    "msg": "srv2.example.com"
}
ok: [saturn.example.com] => {
    "msg": "saturn.example.com"
}
ok: [db1.example.com] => {
    "msg": "db1.example.com"
}
ok: [db2.example.com] => {
    "msg": "db2.example.com"
}
ok: [db3.example.com] => {
    "msg": "db3.example.com"
}
ok: [file2.example.com] => {
    "msg": "file2.example.com"
}

PLAY RECAP ****
db1.example.com      : ok=1    changed=0    unreachable=0    failed=0
  skipped=0   rescued=0   ignored=0

```

```

db2.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
db3.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
file2.example.com    : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
saturn.example.com   : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
srv1.example.com     : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
srv2.example.com     : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0

```

- ▶ 7. Without accessing the groups in the `inventory1` inventory file, modify the host pattern to list these three hosts: `lb1.lab.example.com`, `s1.lab.example.com`, and `db1.example.com`. Run the playbook using the `inventory1` inventory file.

```

[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: lb1.lab.example.com,s1.lab.example.com,db1.example.com
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"
[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [lb1.lab.example.com] => {
    "msg": "lb1.lab.example.com"
}
ok: [s1.lab.example.com] => {
    "msg": "s1.lab.example.com"
}
ok: [db1.example.com] => {
    "msg": "db1.example.com"
}

PLAY RECAP ****
db1.example.com      : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
lb1.lab.example.com   : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0
s1.lab.example.com    : ok=1    changed=0    unreachable=0    failed=0
skipped=0  rescued=0  ignored=0

```

- 8. Use a wildcard host pattern to list hosts that start with a 172.25. IP address in the `inventory1` inventory file. Run the playbook using the `inventory1` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: '172.25.*'
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [172.25.252.23] => {
    "msg": "172.25.252.23"
}
ok: [172.25.252.44] => {
    "msg": "172.25.252.44"
}
ok: [172.25.252.32] => {
    "msg": "172.25.252.32"
}

PLAY RECAP ****
172.25.252.23 : ok=1    changed=0    unreachable=0    failed=0
                  skipped=0   rescued=0   ignored=0
172.25.252.32 : ok=1    changed=0    unreachable=0    failed=0
                  skipped=0   rescued=0   ignored=0
172.25.252.44 : ok=1    changed=0    unreachable=0    failed=0
                  skipped=0   rescued=0   ignored=0
```

- 9. Use a host pattern to list all hosts in the `inventory1` inventory file that start with the letter "s". Run the playbook using the `inventory1` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: 's*'
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
```

```

TASK [Display managed hosts matching the host pattern] ****
ok: [file2.example.com] => {
    "msg": "file2.example.com"
}
ok: [db2.example.com] => {
    "msg": "db2.example.com"
}
ok: [srv1.example.com] => {
    "msg": "srv1.example.com"
}
ok: [srv2.example.com] => {
    "msg": "srv2.example.com"
}
ok: [s1.lab.example.com] => {
    "msg": "s1.lab.example.com"
}
ok: [s2.lab.example.com] => {
    "msg": "s2.lab.example.com"
}
ok: [saturn.example.com] => {
    "msg": "saturn.example.com"
}

PLAY RECAP ****
db2.example.com      : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0
file2.example.com    : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0
s1.lab.example.com   : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0
s2.lab.example.com   : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0
saturn.example.com   : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0
srv1.example.com     : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0
srv2.example.com     : ok=1    changed=0    unreachable=0    failed=0
    skipped=0   rescued=0    ignored=0

```

Notice the `file2.example.com` and `db2.example.com` hosts in the output of the preceding command. They appear in the list because they are both members of a group called `stage`, which also begins with the letter "s."

- 10. Using a list and wildcard host patterns, list all hosts in the `inventory1` inventory in the `prod` group, those hosts with an IP address beginning with 172, and hosts that contain `lab` in their name. Run the playbook using the `inventory1` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: 'prod,172*,*lab*'
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
```

```

ansible.builtin.debug:
  msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****

TASK [Display managed hosts matching the host pattern] ****
ok: [lb2.lab.example.com] => {
    "msg": "lb2.lab.example.com"
}
ok: [db1.example.com] => {
    "msg": "db1.example.com"
}
ok: [jupiter.lab.example.com] => {
    "msg": "jupiter.lab.example.com"
}
ok: [172.25.252.23] => {
    "msg": "172.25.252.23"
}
ok: [172.25.252.44] => {
    "msg": "172.25.252.44"
}
ok: [172.25.252.32] => {
    "msg": "172.25.252.32"
}
ok: [s1.lab.example.com] => {
    "msg": "s1.lab.example.com"
}
ok: [s2.lab.example.com] => {
    "msg": "s2.lab.example.com"
}
ok: [lb1.lab.example.com] => {
    "msg": "lb1.lab.example.com"
}
ok: [file1.lab.example.com] => {
    "msg": "file1.lab.example.com"
}
ok: [web1.lab.example.com] => {
    "msg": "web1.lab.example.com"
}

PLAY RECAP ****
172.25.252.23      : ok=1    changed=0    unreachable=0    failed=0
 skipped=0  rescued=0  ignored=0
172.25.252.32      : ok=1    changed=0    unreachable=0    failed=0
 skipped=0  rescued=0  ignored=0
172.25.252.44      : ok=1    changed=0    unreachable=0    failed=0
 skipped=0  rescued=0  ignored=0
db1.example.com     : ok=1    changed=0    unreachable=0    failed=0
 skipped=0  rescued=0  ignored=0
file1.lab.example.com: ok=1    changed=0    unreachable=0    failed=0
 skipped=0  rescued=0  ignored=0

```

```
jupiter.lab.example.com      : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
lb1.lab.example.com         : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
lb2.lab.example.com         : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
s1.lab.example.com          : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
s2.lab.example.com          : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
web1.lab.example.com        : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

- ▶ 11. List all hosts that belong to both the db and london groups. Run the playbook using the `inventory1` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: db,&london
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory1

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [db2.example.com] => {
    "msg": "db2.example.com"
}
ok: [db3.example.com] => {
    "msg": "db3.example.com"
}

PLAY RECAP ****
db2.example.com      : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
db3.example.com      : ok=1    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

- ▶ 12. Modify the `hosts` value in the `playbook.yml` file so that all servers in the `london` group are targeted. Run the playbook using the `inventory2` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: london
  gather_facts: no
```

```

tasks:
  - name: Display managed hosts matching the host pattern
    ansible.builtin.debug:
      msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory2

PLAY [Resolve host patterns] ****

TASK [Display managed hosts matching the host pattern] ****
ok: [servera.lab.example.com] => {
    "msg": "servera.lab.example.com"
}

PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0

```

- 13. Modify the hosts value in the `playbook.yml` file so that all servers in the `europe` nested group are targeted. Run the playbook using the `inventory2` inventory file.

```

[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: europe
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory2

PLAY [Resolve host patterns] ****

TASK [Display managed hosts matching the host pattern] ****
ok: [servera.lab.example.com] => {
    "msg": "servera.lab.example.com"
}
ok: [serverb.lab.example.com] => {
    "msg": "serverb.lab.example.com"
}

PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0
serverb.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0

```

- ▶ 14. Modify the hosts value in the `playbook.yml` file so that all servers that do not belong to any group are targeted. Run the playbook using the `inventory2` inventory file.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: ungrouped
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory2

PLAY [Resolve host patterns] ****
TASK [Display managed hosts matching the host pattern] ****
ok: [workstation.lab.example.com] => {
    "msg": "workstation.lab.example.com"
}

PLAY RECAP ****
workstation.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
```

- ▶ 15. Modify the hosts value in the `playbook.yml` file to specify a group that does not exist in the `inventory2` inventory file. Run the playbook using the `inventory2` inventory file. Note the message in the output that no match for that host pattern was found.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts: australia
  gather_facts: no
  tasks:
    - name: Display managed hosts matching the host pattern
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }}"

[student@workstation projects-host]$ ansible-navigator run \
> -m stdout playbook.yml -i inventory2

[WARNING]: Could not match supplied host pattern, ignoring: australia

PLAY [Resolve host patterns] ****
skipping: no hosts matched

PLAY RECAP ****
```

Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise.

This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-host
```

This concludes the section.

Including and Importing Files

Objectives

- Manage large playbooks by importing or including other playbooks or tasks from external files, either unconditionally or based on a conditional test.

Managing Large Playbooks

When a playbook gets long or complex, you can divide it up into smaller files to make it easier to manage. You can combine multiple playbooks into a main playbook, or insert lists of tasks from a file into a play. This can make it easier to reuse plays or sequences of tasks in different projects.

Including or Importing Files

Ansible supports two operations for bringing content into a playbook. You can *include* content, or you can *import* content.

When you include content, it is a *dynamic* operation. Ansible processes included content during the run of the playbook, as content is reached.

When you import content, it is a *static* operation. Ansible preprocesses imported content when the playbook is initially parsed, before the run starts.

Importing Playbooks

The `import_playbook` directive allows you to import external files containing lists of plays into a playbook. In other words, you can have a master playbook that imports one or more additional playbooks.

Because the content being imported is a complete playbook, the `import_playbook` feature can only be used at the top level of a playbook and cannot be used inside a play. If you import multiple playbooks, then they are imported and run in order.

The following is a simple example of a master playbook that imports two additional playbooks:

```
- name: Prepare the web server
  import_playbook: web.yml

- name: Prepare the database server
  import_playbook: db.yml
```

You can also interleave plays in your master playbook with imported playbooks.

```

- name: Play 1
  hosts: localhost
  tasks:
    - ansible.builtin.debug:
      msg: Play 1

- name: Import Playbook
  import_playbook: play2.yml

```

In the preceding example, the `Play 1` play runs first, followed by the plays imported from the `play2.yml` playbook.

Importing and Including Tasks

You can import or include a list of tasks from a task file into a play. A task file is a file that contains a flat list of tasks:

```

[admin@node ~]$ cat webserver_tasks.yml
- name: Installs the httpd package
  ansible.builtin.dnf:
    name: httpd
    state: latest

- name: Starts the httpd service
  ansible.builtin.service:
    name: httpd
    state: started

```

Importing Task Files

You can statically import a task file into a play inside a playbook by using the `import_tasks` feature. When you import a task file, the tasks in that file are directly inserted when the playbook is parsed. The location of `import_tasks` in the playbook controls where the tasks are inserted and the order in which multiple imports are run.

```

---
- name: Install web server
  hosts: webservers
  tasks:
    - import_tasks: webserver_tasks.yml

```

When you import a task file, the tasks in that file are directly inserted when the playbook is parsed. Because `import_tasks` statically imports the tasks when the playbook is parsed, the following items must be considered:

- When using the `import_tasks` feature, conditional statements set on the import, such as `when`, are applied to each of the tasks that are imported.
- You cannot use loops with the `import_tasks` feature.
- If you use a variable to specify the name of the file to import, then you cannot use a host or group inventory variable.

Including Task Files

You can also dynamically include a task file into a play inside a playbook by using the `include_tasks` feature.

```
---
- name: Install web server
  hosts: webservers
  tasks:
    - include_tasks: webserver_tasks.yml
```

The `include_tasks` feature does not process content in the playbook until the play is running and that part of the play is reached. The order in which playbook content is processed impacts how the `include_tasks` feature works.

- When using the `include_tasks` feature, conditional statements such as `when` set on the include determine whether the tasks are included in the play at all.
- If you run `ansible-navigator run --list-tasks` to list the tasks in the playbook, then tasks in the included task files are not displayed. The tasks that include the task files are displayed. By comparison, the `import_tasks` feature would not list tasks that import task files, but instead would list the individual tasks from the imported task files.
- You cannot use `ansible-navigator run --start-at-task` to start playbook execution from a task that is in an included task file.
- You cannot use a `notify` statement to trigger a handler name that is in an included task file. You can trigger a handler in the main playbook that includes an entire task file, in which case all tasks in the included file run.

Importing and Including with Conditionals

Conditional statements behave differently depending on whether you are importing or including tasks.

- When you add a conditional to an `import_*` statement, Ansible applies the condition to all tasks within the imported file.
- When you use a conditional on an `include_*` statement, the condition is applied only to the include task itself and not to any other tasks within the included file.

In other words, if you put a conditional on a task that imports content, then each task in the imported content performs that conditional check before it runs.

However, if you put a conditional on a task that includes content, then the conditional determines whether the include happens or not. If the include happens, then all the tasks that are included run normally.



Note

You can find a more detailed discussion of the differences in behavior between `import_tasks` and `include_tasks` when conditionals are used at "Conditionals" [https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html#applying-when-to-roles-imports-and-includes] in the *Ansible User Guide*.

Use Cases for Task Files

Consider the following examples where it might be useful to manage sets of tasks as external files separate from the playbook:

- If new servers require complete configuration, then administrators could create various sets of tasks for creating users, installing packages, configuring services, configuring privileges, setting up access to a shared file system, hardening the servers, installing security updates, and installing a monitoring agent. Each of these sets of tasks could be managed through a separate self-contained task file.
- If servers are managed collectively by the developers, the system administrators, and the database administrators, then every organization can write its own task file which can then be reviewed and integrated by the system manager.
- If a server requires a particular configuration, then it can be integrated as a set of tasks that are executed based on a conditional. In other words, including the tasks only if specific criteria are met.
- If a group of servers needs to run a particular task or set of tasks, then the tasks might only be run on a server if it is part of a specific host group.

Managing Task Files

You can create a dedicated directory for task files, and save all task files in that directory. Then your playbook can simply include or import task files from that directory. This allows construction of a complex playbook and makes it easy to manage its structure and components.

Defining Variables for External Plays and Tasks

The incorporation of plays or tasks from external files into playbooks using the Ansible import and include features greatly enhances the ability to reuse tasks and playbooks across an Ansible environment. To maximize the possibility of reuse, these task and play files should be as generic as possible. Variables can be used to parameterize play and task elements to expand the application of tasks and plays.

For example, the following task file installs the package needed for a web service, and then enables and starts the necessary service.

```
---
- name: Install the httpd package
  ansible.builtin.dnf:
    name: httpd
    state: latest
- name: Start the httpd service
  ansible.builtin.service:
    name: httpd
    enabled: true
    state: started
```

If you parameterize the package and service elements as shown in the following example, then the task file can also be used for the installation and administration of other software and their services, rather than being useful for a web service only.

```
---
- name: Install the {{ package }} package
  ansible.builtin.dnf:
    name: "{{ package }}"
    state: latest
- name: Start the {{ service }} service
  ansible.builtin.service:
    name: "{{ service }}"
    enabled: true
    state: started
```

Subsequently, when incorporating the task file into a playbook, define the variables to use for the task execution as follows:

```
...output omitted...
tasks:
- name: Import task file and set variables
  import_tasks: task.yml
  vars:
    package: httpd
    service: httpd
```

Ansible makes the passed variables available to the tasks imported from the external file.

You can use the same technique to make play files more reusable. When incorporating a play file into a playbook, pass the variables to use for the play execution as follows:

```
...output omitted...
- name: Import play file and set the variable
  import_playbook: play.yml
  vars:
    package: mariadb
```



Important

Earlier versions of Ansible used an `include` feature to include both playbooks and task files, depending on context. This functionality is being deprecated for a number of reasons.

Prior to Ansible 2.0, `include` operated like a static import. In Ansible 2.0 it was changed to operate dynamically, but this created some limitations. In Ansible 2.1 it became possible for `include` to be dynamic or static depending on task settings, which was confusing and error-prone. There were also issues with ensuring that `include` worked correctly in all contexts.

Thus, `include` was replaced in Ansible 2.4 with new directives such as `include_tasks`, `import_tasks`, and `import_playbook`. You might find examples of `include` in earlier playbooks, but you should avoid using it in new ones.



References

Including and Importing – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse/includes.html

Creating Reusable Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html

Conditionals – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html

► Guided Exercise

Including and Importing Files

In this exercise, you include and import playbooks and tasks in a top-level Ansible Playbook.

Outcomes

- Include task and playbook files in playbooks.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start projects-file
```

Instructions

- 1. Change into the `/home/student/projects-file` directory.

```
[student@workstation ~]$ cd ~/projects-file  
[student@workstation projects-file]$
```

- 2. Review the contents of the three files in the `tasks` subdirectory.

- 2.1. Review the contents of the `tasks/environment.yml` file. The file contains tasks for package installation and service administration.

```
---  
- name: Install the {{ package }} package  
  ansible.builtin.dnf:  
    name: "{{ package }}"  
    state: latest  
- name: Start the {{ service }} service  
  ansible.builtin.service:  
    name: "{{ service }}"  
    enabled: true  
    state: started
```

- 2.2. Review the contents of the `tasks/firewall.yml` file. The file contains tasks for installation, administration, and configuration of firewall software.

```
---  
- name: Install the firewall  
  ansible.builtin.dnf:
```

```

name: "{{ firewall_pkg }}"
state: latest

- name: Start the firewall
  ansible.builtin.service:
    name: "{{ firewall_svc }}"
    enabled: true
    state: started

- name: Open the port for {{ rule }}
  ansible.posix.firewalld:
    service: "{{ item }}"
    immediate: true
    permanent: true
    state: enabled
  loop: "{{ rule }}"

```

- 2.3. Review the contents of the `tasks/placeholder.yml` file. This file contains a task for populating a placeholder web content file.

```

---
- name: Create placeholder file
  ansible.builtin.copy:
    content: "{{ ansible_facts['fqdn'] }} has been customized using Ansible.\n"
    dest: "{{ file }}"

```

- ▶ 3. Review the contents of the `test.yml` file in the `plays` subdirectory. This file contains a play that tests connections to a web service.

```

---
- name: Test web service
  hosts: workstation
  become: false
  tasks:
    - name: connect to internet web server
      ansible.builtin.uri:
        url: "{{ url }}"
        status_code: 200

```

- ▶ 4. Create a playbook named `playbook.yml`. Define the first play with the name `Configure web server`. The play should execute against the `servera.lab.example.com` managed host defined in the `inventory` file. The beginning of the file should appear as follows:

```

---
- name: Configure web server
  hosts: servera.lab.example.com

```

- ▶ 5. In the first play of the `playbook.yml` playbook, configure its `tasks` section with three sets of tasks that are included or imported from `tasks` files.

The play's first task must include the first set of tasks from the `tasks/environment.yml` tasks file. Define the necessary task variables to install the `httpd` package and to enable and start the `httpd` service.

The play's second task must import the second set of tasks from the `tasks/firewall.yml` tasks file. Define the necessary task variables to install the `firewalld` package to enable and start the `firewalld` service, and to allow plain text and secure HTTP connections.

The play's third task must import the third set of tasks from the `tasks/placeholder.yml` task file.

- 5.1. Create the tasks section in the first play by adding the following entry to the `playbook.yml` playbook.

```
tasks:
```

- 5.2. Include the first set of tasks from `tasks/environment.yml` using the `include_tasks` feature. Set the package and service variables to `httpd`.

```
- name: Include the environment task file and set the variables
  include_tasks: tasks/environment.yml
  vars:
    package: httpd
    service: httpd
```

- 5.3. Import the second set of tasks from `tasks/firewall.yml` using the `import_tasks` feature. Set the `firewall_pkg` and `firewall_svc` variables to `firewalld`. Set the `rule` variable to `http` and `https`.

```
- name: Import the firewall task file and set the variables
  import_tasks: tasks/firewall.yml
  vars:
    firewall_pkg: firewalld
    firewall_svc: firewalld
    rule:
      - http
      - https
```

- 5.4. Import the last task set from `tasks/placeholder.yml` using the `import_tasks` feature. Set the `file` variable to `/var/www/html/index.html`.

```
- name: Import the placeholder task file and set the variable
  import_tasks: tasks/placeholder.yml
  vars:
    file: /var/www/html/index.html
```

- ▶ 6. Add a second play to the `playbook.yml` playbook, importing the contents of the `plays/test.yml` playbook.

- 6.1. Add a second play to the `playbook.yml` playbook to validate the web server installation. Import the play from `plays/test.yml`. Set the `url` variable to `http://servera.lab.example.com` as a play variable.

```
- name: Import test play file and set the variable
  import_playbook: plays/test.yml
  vars:
    url: 'http://servera.lab.example.com'
```

6.2. The finished playbook should consist of the following content:

```
---
- name: Configure web server
  hosts: servera.lab.example.com

  tasks:
    - name: Include the environment task file and set the variables
      include_tasks: tasks/environment.yml
      vars:
        package: httpd
        service: httpd

    - name: Import the firewall task file and set the variables
      import_tasks: tasks/firewall.yml
      vars:
        firewall_pkg: firewalld
        firewall_svc: firewalld
        rule:
          - http
          - https

    - name: Import the placeholder task file and set the variable
      import_tasks: tasks/placeholder.yml
      vars:
        file: /var/www/html/index.html

- name: Import test play file and set the variable
  import_playbook: plays/test.yml
  vars:
    url: 'http://servera.lab.example.com'
```

6.3. Save the changes to the `playbook.yml` playbook.

- ▶ 7. Before running the playbook, verify that its syntax is correct by running `ansible-navigator run -m stdout playbook.yml --syntax-check`. Correct any reported errors before moving to the next step.

```
[student@workstation projects-file]$ ansible-navigator run \
> -m stdout playbook.yml --syntax-check
playbook: /home/student/projects-file/playbook.yml
```

- ▶ 8. Run the `playbook.yml` playbook. The output of `ansible-navigator` shows the tasks and plays that are imported and the task file that is included.

```
[student@workstation projects-file]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Configure web server] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Include the environment task file and set the variables] ****
included: /home/student/projects-file/tasks/environment.yml for
servera.lab.example.com

TASK [Install the httpd package] ****
changed: [servera.lab.example.com]

TASK [Start the httpd service] ****
changed: [servera.lab.example.com]

TASK [Install the firewall] ****
ok: [servera.lab.example.com]

TASK [Start the firewall] ****
ok: [servera.lab.example.com]

TASK [Open the port for ['http', 'https']] ****
changed: [servera.lab.example.com] => (item=http)
changed: [servera.lab.example.com] => (item=https)

TASK [Create placeholder file] ****
changed: [servera.lab.example.com]

PLAY [Test web service] ****

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

TASK [connect to internet web server] ****
ok: [workstation]

PLAY RECAP ****
servera.lab.example.com      : ok=8    changed=4    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
workstation                  : ok=2    changed=0    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-file
```

This concludes the section.

▶ Lab

Managing Complex Plays and Playbooks

In this lab, you modify a complex playbook to be easier to manage by using host patterns, includes, and imports.

Outcomes

- Simplify host references in a playbook by specifying host patterns.
- Restructure a playbook so that tasks are imported from external task files.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start projects-review
```

Instructions

You have inherited a playbook from the previous administrator of some web servers. The playbook is used to configure a web service on `servera.lab.example.com`, `serverb.lab.example.com`, `serverc.lab.example.com`, and `serverd.lab.example.com`. The playbook also configures the firewall on the four managed hosts so that web traffic is allowed.

Make the following changes to the `playbook.yml` playbook file so that it is easier to manage.

1. Simplify the list of managed hosts used by the play in the `/home/student/projects-review/playbook.yml` playbook by using a wildcard host pattern.
You have a second playbook, `/home/student/projects-review/host-test.yml`, that contains a play that you can use to test host patterns before you use them in the play in the `playbook.yml` playbook.
2. Restructure the `playbook.yml` playbook so that the first three tasks in its play are kept in an external task file located at `tasks/web_tasks.yml`. Use the `import_tasks` feature to incorporate this task file into the play.
3. Restructure the `playbook.yml` playbook so that the fourth, fifth, and sixth tasks in its play are kept in an external task file located at `tasks/firewall_tasks.yml`. Use the `import_tasks` feature to incorporate this task file into the play.
4. Both the `tasks/web_tasks.yml` file and the `tasks/firewall_tasks.yml` file contain tasks that install packages and enable services. Those could be consolidated into a single task file and you could use variables to control which packages and services are installed and enabled by those tasks.

Move the tasks that install packages and enable services into a new file named `tasks/install_and_enable.yml` and update them to use variables. Replace the original tasks with `import_tasks` statements, passing in appropriate values to the new variables.

5. Confirm that you made the changes to the play in `playbook.yml` correctly, and then run the playbook.

Evaluation

Run the `lab grade projects-review` command from `workstation` to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab grade projects-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-review
```

This concludes the section.

► Solution

Managing Complex Plays and Playbooks

In this lab, you modify a complex playbook to be easier to manage by using host patterns, includes, and imports.

Outcomes

- Simplify host references in a playbook by specifying host patterns.
- Restructure a playbook so that tasks are imported from external task files.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start projects-review
```

Instructions

You have inherited a playbook from the previous administrator of some web servers. The playbook is used to configure a web service on `servera.lab.example.com`, `serverb.lab.example.com`, `serverc.lab.example.com`, and `serverd.lab.example.com`. The playbook also configures the firewall on the four managed hosts so that web traffic is allowed.

Make the following changes to the `playbook.yml` playbook file so that it is easier to manage.

1. Simplify the list of managed hosts used by the play in the `/home/student/projects-review/playbook.yml` playbook by using a wildcard host pattern.
You have a second playbook, `/home/student/projects-review/host-test.yml`, that contains a play that you can use to test host patterns before you use them in the play in the `playbook.yml` playbook.
 - 1.1. Change into the `/home/student/projects-review` directory. Review the `hosts` parameter in the `playbook.yml` file.

```
[student@workstation ~]$ cd ~/projects-review
[student@workstation projects-review]$ cat playbook.yml
---
- name: Install and configure web service
  hosts:
    - servera.lab.example.com
    - serverb.lab.example.com
```

```
- serverc.lab.example.com
- serverd.lab.example.com
...output omitted...
```

- 1.2. Verify that the host pattern `server*.lab.example.com` correctly identifies the four managed hosts that are targeted by the `playbook.yml` playbook. View the contents of the `host-test.yml` playbook, then run the playbook.

```
[student@workstation projects-review]$ cat host-test.yml
---
- name: List inventory hostnames
  hosts: server*.lab.example.com
  gather_facts: no
  tasks:
    - name: List inventory hostnames
      ansible.builtin.debug:
        msg: "{{inventory_hostname}}"

[student@workstation projects-review]$ ansible-navigator run \
> -m stdout host-test.yml

PLAY [List inventory hostnames] ****
TASK [List inventory hostnames] ****
ok: [servera.lab.example.com] => {
    "msg": "servera.lab.example.com"
}
ok: [serverb.lab.example.com] => {
    "msg": "serverb.lab.example.com"
}
ok: [serverc.lab.example.com] => {
    "msg": "serverc.lab.example.com"
}
ok: [serverd.lab.example.com] => {
    "msg": "serverd.lab.example.com"
}

PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
serverb.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
serverc.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
serverd.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

- 1.3. Replace the host list in the `playbook.yml` playbook with the `server*.lab.example.com` host pattern.

```
---
- name: Install and configure web service
  hosts: server*.lab.example.com
...output omitted...
```

2. Restructure the `playbook.yml` playbook so that the first three tasks in its play are kept in an external task file located at `tasks/web_tasks.yml`. Use the `import_tasks` feature to incorporate this task file into the play.

2.1. Create the `tasks` subdirectory.

```
[student@workstation projects-review]$ mkdir tasks
```

- 2.2. Place the contents of the first three tasks in the play in the `playbook.yml` playbook into the `tasks/web_tasks.yml` file. The task file should contain the following content:

```
---
- name: Install httpd
  ansible.builtin.dnf:
    name: httpd
    state: latest

- name: Enable and start httpd
  ansible.builtin.service:
    name: httpd
    enabled: true
    state: started

- name: Tuning configuration installed
  ansible.builtin.copy:
    src: files/tune.conf
    dest: /etc/httpd/conf.d/tune.conf
    owner: root
    group: root
    mode: 0644
  notify:
    - restart httpd
```

- 2.3. Remove the first three tasks from the play in the `playbook.yml` playbook. Put the following lines in their place to import the `tasks/web_tasks.yml` task file.

```
- name: Import the web_tasks.yml task file
  import_tasks: tasks/web_tasks.yml
```

3. Restructure the `playbook.yml` playbook so that the fourth, fifth, and sixth tasks in its play are kept in an external task file located at `tasks/firewall_tasks.yml`. Use the `import_tasks` feature to incorporate this task file into the play.
- 3.1. Place the contents of the three remaining tasks in the play in the `playbook.yml` playbook into the `tasks/firewall_tasks.yml` file. The task file should contain the following content.

```

---
- name: Install firewalld
  ansible.builtin.dnf:
    name: firewalld
    state: latest

- name: Enable and start the firewall
  ansible.builtin.service:
    name: firewalld
    enabled: true
    state: started

- name: Open the port for http
  ansible.posix.firewalld:
    service: http
    immediate: true
    permanent: true
    state: enabled

```

- 3.2. Remove the remaining three tasks from the play in the `playbook.yml` playbook. Put the following lines in their place, which imports the `tasks/firewall_tasks.yml` task file.

```

- name: Import the firewall_tasks.yml task file
  import_tasks: tasks/firewall_tasks.yml

```

4. Both the `tasks/web_tasks.yml` file and the `tasks/firewall_tasks.yml` file contain tasks that install packages and enable services. Those could be consolidated into a single task file and you could use variables to control which packages and services are installed and enabled by those tasks.

Move the tasks that install packages and enable services into a new file named `tasks/install_and_enable.yml` and update them to use variables. Replace the original tasks with `import_tasks` statements, passing in appropriate values to the new variables.

- 4.1. Copy the `ansible.builtin.dnf` and `ansible.builtin.service` tasks from `tasks/web_tasks.yml` into a new file named `tasks/install_and_enable.yml`.

```

---
- name: Install httpd
  ansible.builtin.dnf:
    name: httpd
    state: latest

- name: Enable and start httpd
  ansible.builtin.service:
    name: httpd
    enabled: true
    state: started

```

- 4.2. Replace the package and service names in `tasks/install_and_enable.yml` with the variables `package` and `service`.

```
---
- name: Install {{ package }}
  ansible.builtin.dnf:
    name: "{{ package }}"
    state: latest

- name: Enable and start {{ service }}
  ansible.builtin.service:
    name: "{{ service }}"
    enabled: true
    state: started
```

- 4.3. Replace the `ansible.builtin.dnf` and `ansible.builtin.service` tasks in `tasks/web_tasks.yml` and `tasks/firewall_tasks.yml` with `import_tasks` statements that import `tasks/install_and_enable.yml` and set appropriate values on task variables to install the correct package and start the correct service.

```
---
- name: Install and start httpd
  import_tasks: install_and_enable.yml
  vars:
    package: httpd
    service: httpd
```

```
---
- name: Install and start firewalld
  import_tasks: install_and_enable.yml
  vars:
    package: firewalld
    service: firewalld
```

5. Confirm that you made the changes to the play in `playbook.yml` correctly, and then run the playbook.

- 5.1. Verify that the `playbook.yml` playbook contains the following contents:

```
---
- name: Install and configure web service
  hosts: server*.lab.example.com

  tasks:
    - name: Import the web_tasks.yml task file
      import_tasks: tasks/web_tasks.yml

    - name: Import the firewall_tasks.yml task file
      import_tasks: tasks/firewall_tasks.yml

  handlers:
    - name: restart httpd
```

```
ansible.builtin.service:  
  name: httpd  
  state: restarted
```

- 5.2. Run the `playbook.yml` playbook with `ansible-navigator run --syntax-check` to verify the playbook contains no syntax errors. Correct any reported errors before proceeding.

```
[student@workstation projects-review]$ ansible-navigator run \  
> -m stdout playbook.yml --syntax-check  
playbook: /home/student/projects-review/playbook.yml
```

- 5.3. Run the `playbook.yml` playbook.

```
[student@workstation projects-review]$ ansible-navigator run \  
> -m stdout playbook.yml  
  
PLAY [Install and configure web service] *****  
  
TASK [Gathering Facts] *****  
ok: [serverd.lab.example.com]  
ok: [serverc.lab.example.com]  
ok: [serverb.lab.example.com]  
ok: [servera.lab.example.com]  
  
TASK [Install httpd] *****  
changed: [serverb.lab.example.com]  
changed: [servera.lab.example.com]  
changed: [serverd.lab.example.com]  
changed: [serverc.lab.example.com]  
  
TASK [Enable and start httpd] *****  
changed: [servera.lab.example.com]  
changed: [serverb.lab.example.com]  
changed: [serverd.lab.example.com]  
changed: [serverc.lab.example.com]  
  
TASK [Tuning configuration installed] *****  
changed: [serverd.lab.example.com]  
changed: [serverc.lab.example.com]  
changed: [serverb.lab.example.com]  
changed: [servera.lab.example.com]  
  
TASK [Install firewalld] *****  
ok: [serverb.lab.example.com]  
ok: [servera.lab.example.com]  
ok: [serverd.lab.example.com]  
ok: [serverc.lab.example.com]  
  
TASK [Enable and start firewalld] *****  
ok: [servera.lab.example.com]  
ok: [serverb.lab.example.com]  
ok: [serverc.lab.example.com]  
ok: [serverd.lab.example.com]
```

```
TASK [Open the port for http] ****
changed: [serverd.lab.example.com]
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]
changed: [serverc.lab.example.com]

RUNNING HANDLER [restart httpd] ****
changed: [serverd.lab.example.com]
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=8    changed=5    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
serverb.lab.example.com    : ok=8    changed=5    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
serverc.lab.example.com    : ok=8    changed=5    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
serverd.lab.example.com    : ok=8    changed=5    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
```

Evaluation

Run the `lab grade projects-review` command from `workstation` to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab grade projects-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-review
```

This concludes the section.

Summary

- Host patterns are used to specify the managed hosts to be targeted by plays.
- You can specify a list of multiple host patterns in the `hosts` directive of a play.
- You can use the `import_playbook` feature to incorporate external play files into playbooks.
- You can use the `include_tasks` or `import_tasks` features to incorporate external task files into playbooks.
- When you *include* content, Ansible processes it dynamically as content is reached. When you *import* content, Ansible preprocesses it before the run starts.

Chapter 7

Simplifying Playbooks with Roles and Ansible Content Collections

Goal

Use Ansible Roles and Ansible Content Collections to develop playbooks more quickly and to reuse Ansible code.

Objectives

- Describe the purpose of an Ansible Role, its structure, and how roles are used in playbooks.
- Create a role in a playbook's project directory and run it as part of one of the plays in the playbook.
- Select and retrieve roles from external sources such as Git repositories or Ansible Galaxy, and use them in your playbooks.
- Obtain a set of related roles, supplementary modules, and other content from an Ansible Content Collection and use them in a playbook.
- Write playbooks that take advantage of system roles for Red Hat Enterprise Linux to perform standard operations.

Sections

- Describing Role Structure (and Quiz)
- Creating Roles (and Guided Exercise)
- Deploying Roles from External Content Sources (and Guided Exercise)
- Getting Roles and Modules from Content Collections (and Guided Exercise)
- Reusing Content with System Roles (and Guided Exercise)

Lab

- Simplifying Playbooks with Roles and Ansible Content Collections

Describing Role Structure

Objectives

- Describe the purpose of an Ansible Role, its structure, and how roles are used in playbooks.

Structuring Ansible Playbooks with Roles

As you develop more playbooks, you are likely to discover that you have many opportunities to reuse code from playbooks that you wrote previously. Perhaps, you could repurpose a play to configure a MySQL database for one application to configure a MySQL database for another application, with different hostnames, passwords, and users.

That play might be long and complex, with many included or imported files and tasks and handlers to manage various situations. Copying all that code into another playbook might be nontrivial work.

Ansible *roles* make it easier to reuse Ansible code generically. You can package all the tasks, variables, files, templates, and other resources needed to provision infrastructure or deploy applications in a standardized directory structure. Copy a role from project to project by copying the directory, then call the role within a play.

A well-written role can take variables passed from the playbook. These variables can adjust the behavior of the role, setting all the site-specific hostnames, IP addresses, usernames, secrets, or other locally-specific details.

For example, you might write a role to deploy a database server to support variables that set the hostname, database admin user and password, and other parameters that are customized for your installation.

You also can ensure that reasonable default values are set for those variables in the role, if they are not set in the play that calls the role.

Ansible roles have the following benefits:

- Roles group content together, enabling easy sharing of code with others.
- Roles can define the essential elements of a system type, such as a web server, database server, or Git repository.
- Roles make larger projects more manageable.
- Roles can be developed in parallel by different users.

In addition to writing, using, reusing, and sharing your own roles, you can obtain roles from other sources. You can find roles by using distribution packages, such as Ansible Content Collections. Or, you can download roles from the Red Hat automation hub, a private automation hub, and from the community's Ansible Galaxy website.

Red Hat Enterprise Linux includes some roles in the `rhel-system-roles` package. You learn more about `rhel-system-roles` later in this chapter.

Examining the Ansible Role Structure

An Ansible role is defined by a standardized structure of subdirectories and files.

The top-level directory defines the name of the role itself. Files are organized into subdirectories that are named according to each file's purpose in the role, such as `tasks` and `handlers`.

The `files` and `templates` subdirectories contain files referenced by tasks in other playbooks and task files.

The following `tree` command displays the directory structure of the `user.example` role.

```
[user@host roles]$ tree user.example
user.example/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Ansible Role Subdirectories

Subdirectory	Function
<code>defaults</code>	The <code>main.yml</code> file in this directory contains the default values of role variables that can be overwritten when the role is used. These variables have low precedence and are intended to be changed and customized in plays.
<code>files</code>	This directory contains static files that are referenced by role tasks.
<code>handlers</code>	The <code>main.yml</code> file in this directory contains the role's handler definitions.
<code>meta</code>	The <code>main.yml</code> file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
<code>tasks</code>	The <code>main.yml</code> file in this directory contains the role's task definitions.
<code>templates</code>	This directory contains Jinja2 templates that are referenced by role tasks.
<code>tests</code>	This directory can contain an inventory and <code>test.yml</code> playbook that can be used to test the role.

Subdirectory	Function
vars	The <code>main.yml</code> file in this directory defines the role's variable values. Often these variables are used for internal purposes within the role. These variables have high precedence and are not intended to be changed when used in a playbook.

Not every role has all of these directories.

Defining Variables and Defaults

Role variables are defined by creating a `vars/main.yml` file with key-value pairs in the role directory hierarchy. These variables are referenced in role task files like any other variable: `{{ VAR_NAME }}`. These variables have a high precedence and can not be overridden by inventory variables. These variables are used by the internal functioning of the role.

Default variables enable you to set default values for variables that can be used in a play to configure the role or customize its behavior. These variables are defined by creating a `defaults/main.yml` file with key-value pairs in the role directory hierarchy. Default variables have the lowest precedence of any available variables.

Default variable values can be overridden by any other variable, including inventory variables. These variables are intended to provide the person writing a play that uses the role with a way to customize or control exactly what it is going to do. You can use default variables to provide information to the role that it needs to configure or deploy something properly.

Define a specific variable in either `vars/main.yml` or `defaults/main.yml`, but not in both places. Use default variables when you intend that the variable values might be overridden.



Important

Roles should not have site specific data in them or contain any secrets like passwords or private keys because roles are supposed to be generic, reusable, and freely shareable. Therefore, site specific details should not be hard coded into them.

Secrets should be provided to the role through other means. This requirement is one reason that you might want to set role variables when calling a role. Role variables set in the play could provide the secret, or point to an Ansible Vault encrypted file containing the secret.

Using Ansible Roles in a Play

There are several ways to call roles in a play. The two primary methods are:

- You can include or import them like a task in your `tasks` list.
- You can create a `roles` list that runs specific roles before your play's tasks.

The first method is the most flexible, but the second method is also commonly used and was invented before the first method.

Including and Importing Roles as Tasks

Roles can be added to a play by using an ordinary task. Use the `ansible.builtin.import_role` module to statically import a role, and the `ansible.builtin.include_role` module to dynamically include a role.

The following play demonstrates how you can import a role by using a task with the `ansible.builtin.import_role` module. The example play runs the task `A normal task` first, then imports the `role2` role.

```
- name: Run a role as a task
  hosts: remote.example.com
  tasks:
    - name: A normal task
      ansible.builtin.debug:
        msg: 'first task'
    - name: A task to import role2 here
      ansible.builtin.import_role:
        name: role2
```

With the `ansible.builtin.import_role` module, Ansible treats the role as a static import and parses it during initial playbook processing.

In the preceding example, when the playbook is parsed:

- If `roles/role2/tasks/main.yml` exists, Ansible adds the tasks in that file to the play.
- If `roles/role2/handlers/main.yml` exists, Ansible adds the handlers in that file to the play.
- If `roles/role2/defaults/main.yml` exists, Ansible adds the default variables in that file to the play.
- If `roles/role2/vars/main.yml` exists, Ansible adds the variables in that file to the play (possibly overriding values from role default variables due to precedence).



Important

Because `ansible.builtin.import_role` is processed when the playbook is parsed, the role's handlers, default variables, and role variables are all exposed to all the tasks and roles in the play, and can be accessed by tasks and roles that precede it in the play (even though the role has not run yet).

You can also set variables for the role when you call the task, in the same way that you can set task variables:

```
- name: Run a role as a task
  hosts: remote.example.com
  tasks:
    - name: A task to include role2 here
      ansible.builtin.import_role:
        name: role2
        vars:
          var1: val1
          var2: val2
```

The `ansible.builtin.include_role` module works in a similar way, but it dynamically includes the role when the playbook is running instead of statically importing it when the playbook is initially parsed.

One key difference between the two modules is how they handle task-level keywords, conditionals, and loops:

- `ansible.builtin.import_role` applies the task's conditionals and loops to each of the tasks being imported.
- `ansible.builtin.include_role` applies the task's conditionals and loops to the statement that determines whether the role is included or not.

In addition, when you include a role, its role variables and default variables are *not* exposed to the rest of the play, unlike `ansible.builtin.import_role`.

Using a Roles Section in a Play

Another way you can call roles in a play is to list them in a `roles` section. The `roles` section is very similar to the `tasks` section, except instead of consisting of a list of tasks, it consists of a list of roles.

In the following example play, the `role1` role runs, then the `role2` role runs.

```
---
- name: A play that only has roles
  hosts: remote.example.com
  roles:
    - role: role1
    - role: role2
```

For each role specified, the role's tasks, handlers, variables, and dependencies are imported into the play in the order in which they are listed.

When you use a `roles` section to import roles into a play, the roles run first, before any tasks that you define for that play. Whether the `roles` section is listed before or after the `tasks` section in the play does not matter.

```
- name: Roles always run first
  hosts: remote.example.com
  tasks:
    - name: A task
      ansible.builtin.debug:
        msg: "This task runs after the role."
  roles:
    - role: role1
```

Because roles run first, it generally makes sense to list the `roles` section before the `tasks` section, if you must have both. The preceding play can be rewritten as follows without changing how it runs:

```
- name: A task.
  ansible.builtin.debug:
    msg: "This task runs after the role."
```

**Important**

A `tasks` section in a play is not required. In fact, it is generally a good practice to avoid both `roles` and `tasks` sections in a play to avoid confusion about the order in which roles and tasks run.

If you must have a `tasks` section and roles, it is better to create tasks that use `ansible.builtin.import_roles` and `ansible.builtin.include_roles` to run at the correct points in the play's execution.

The following example sets values for two role variables of `role2`, `var1` and `var2`. Any `defaults` and `vars` variables are overridden when `role2` is used.

```
---
- name: A play that runs the second role with variables
  hosts: remote.example.com
  roles:
    - role: role1
    - role: role2
      var1: val1
      var2: val2
```

Another equivalent playbook syntax that you might see in this case is:

```
---
- name: A play that runs the second role with variables
  hosts: remote.example.com
  roles:
    - role: role1
    - { role: role2, var1: val1, var2: val2 }
```

There are situations in which this can be harder to read, even though it is more compact.

**Important**

Ansible looks for duplicate role lines in the `roles` section. If two roles are listed with exactly the same parameters, the role only runs once.

For example, the following `roles` section only runs `role1` one time:

```
roles:
  - { role: role1, service: "httpd" }
  - { role: role2, var1: true }
  - { role: role1, service: "httpd" }
```

To run the same role a second time, it must have different parameters defined:

```
roles:
  - { role: role1, service: "httpd" }
  - { role: role2, var1: true }
  - { role: role1, service: "postfix" }
```

Special Tasks Sections

There are two special task sections, `pre_tasks` and `post_tasks`, that are occasionally used with `roles` sections. The `pre_tasks` section is a list of tasks, similar to `tasks`, but these tasks run before any of the roles in the `roles` section. If any task in the `pre-tasks` section notify a handler, then those handler tasks run before the roles or normal tasks.

Plays also support a `post_tasks` keyword. These tasks run after the play's `tasks` and any handlers notified by the play's `tasks`.

The following play shows an example with `pre_tasks`, `roles`, `tasks`, `post_tasks` and `handlers`. It is unusual that a play would contain all of these sections.

```
- name: Play to illustrate order of execution
  hosts: remote.example.com
  pre_tasks:
    - name: This task runs first
      ansible.builtin.debug:
        msg: This task is in pre_tasks
      notify: my handler
      changed_when: true
  roles:
    - role: role1
  tasks:
    - name: This task runs after the roles
      ansible.builtin.debug:
        msg: This task is in tasks
      notify: my handler
      changed_when: true
  post_tasks:
    - name: This task runs last
      ansible.builtin.debug:
        msg: This task is in post_tasks
      notify: my handler
      changed_when: true
  handlers:
    - name: my handler
      ansible.builtin.debug:
        msg: Running my handler
```

In the preceding example, an `ansible.builtin.debug` task runs in each `tasks` section and in the role in the `roles` section. Each of those tasks notifies the `my handler` handler, which means the `my handler` task runs three times:

- After all the `pre_tasks` tasks run
- After all the `roles` tasks and `tasks` tasks run
- After all the `post_tasks` run



Note

In general, if you think you need `pre_tasks` and `post_tasks` sections in your play because you are using `roles`, consider importing the roles as tasks and including only a `tasks` section. Alternatively, it might be simpler to have multiple plays in your playbook.



References

Roles – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html

► Quiz

Describing Role Structure

Choose the correct answers to the following questions:

- ▶ 1. **Which one of the following statements best describes roles?**
 - a. Configuration settings that allow specific users to run Ansible Playbooks.
 - b. Playbooks for a data center.
 - c. Collections of YAML task files and supporting items arranged in a specific structure.

- ▶ 2. **Which role subdirectory contains Jinja2 files that are referenced by role tasks?**
 - a. Handlers
 - b. Files
 - c. Templates
 - d. Variables
 - e. Meta

- ▶ 3. **How do you use a role in a play so that its role variables and default variables are exposed to the rest of the play?**
 - a. Use the pre_tasks keyword.
 - b. Use the post_tasks keyword.
 - c. Use the ansible.builtin.import_role module.
 - d. Use the ansible.builtin.include_role module.

- ▶ 4. **Which file in a role's directory hierarchy should contain the default values of variables that might be used as parameters to the role?**
 - a. defaults/main.yml
 - b. meta/main.yml
 - c. vars/main.yml
 - d. The host inventory file.

► Solution

Describing Role Structure

Choose the correct answers to the following questions:

- ▶ 1. **Which one of the following statements best describes roles?**
 - a. Configuration settings that allow specific users to run Ansible Playbooks.
 - b. Playbooks for a data center.
 - c. Collections of YAML task files and supporting items arranged in a specific structure.

- ▶ 2. **Which role subdirectory contains Jinja2 files that are referenced by role tasks?**
 - a. Handlers
 - b. Files
 - c. Templates
 - d. Variables
 - e. Meta

- ▶ 3. **How do you use a role in a play so that its role variables and default variables are exposed to the rest of the play?**
 - a. Use the pre_tasks keyword.
 - b. Use the post_tasks keyword.
 - c. Use the ansible.builtin.import_role module.
 - d. Use the ansible.builtin.include_role module.

- ▶ 4. **Which file in a role's directory hierarchy should contain the default values of variables that might be used as parameters to the role?**
 - a. defaults/main.yml
 - b. meta/main.yml
 - c. vars/main.yml
 - d. The host inventory file.

Creating Roles

Objectives

- Create a role in a playbook's project directory and run it as part of one of the plays in the playbook.

The Role Creation Process

Creating roles in Ansible does not require any special development tools.

Creating and using a role is a three-step process:

1. Create the role directory structure.
2. Define the role content.
3. Use the role in a playbook.

Creating the Role Directory Structure

Ansible looks for roles in a subdirectory called `roles` in the directory containing your Ansible Playbook. Each role has its own directory with a standardized directory structure. This structure allows you to store roles with the playbook and other supporting files.

For example, the following directory structure contains the files that define the `motd` role.

```
[user@host ~]$ tree roles/
roles/
└── motd
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    ├── meta
    │   └── main.yml
    ├── README.md
    ├── tasks
    │   └── main.yml
    └── templates
        └── motd.j2
```

The `README.md` provides a basic human-readable description of the role, documentation, examples of how to use it, and any non-Ansible role requirements. The `meta` subdirectory contains a `main.yml` file that specifies information about the author, license, compatibility, and dependencies for the module.

The `files` subdirectory contains fixed content files and the `templates` subdirectory contains templates that the role can deploy.

The other subdirectories can contain `main.yml` files that define default variable values, handlers, tasks, role metadata, or variables, depending on their subdirectory.

If a subdirectory exists but is empty, such as `handlers` in this example, it is ignored. You can omit the subdirectory altogether if the role does not use a feature. This example omits the `vars` subdirectory.

Creating a Role Skeleton

You can create all the subdirectories and files needed for a new role by using standard Linux commands. Alternatively, command-line utilities exist to automate the process of new role creation.

The `ansible-galaxy` command-line tool (covered in more detail later in this course) is used to manage Ansible roles, including the creation of new roles. You can run `ansible-galaxy init` to create the directory structure for a new role. Specify the role's name as an argument to the command, which creates a subdirectory for the new role in the current working directory.

```
[user@host playbook-project]$ cd roles
[user@host roles]$ ansible-galaxy init my_new_role
- Role my_new_role was created successfully
[user@host roles]$ ls my_new_role/
defaults  files  handlers  meta  README.md  tasks  templates  tests  vars
```

Defining the Role Content

After creating the directory structure, you must write the content of the role. A good place to start is the `ROLENAMESPACE/tasks/main.yml` task file, the main list of tasks that the role runs.

The following `tasks/main.yml` file manages the `/etc/motd` file on managed hosts. It uses the `template` module to deploy the template named `motd.j2` to the managed host. Because the `template` module is configured within a role task, instead of a playbook task, the `motd.j2` template is retrieved from the role's `templates` subdirectory.

```
[user@host ~]$ cat roles/motd/tasks/main.yml
---
# tasks file for motd

- name: deliver motd file
  ansible.builtin.template:
    src: motd.j2
    dest: /etc/motd
    owner: root
    group: root
    mode: 0444
```

The following command displays the contents of the `motd.j2` template of the `motd` role. It references Ansible facts and a `system_owner` variable.

```
[user@host ~]$ cat roles/motd/templates/motd.j2
This is the system {{ ansible_facts['hostname'] }}.

Today's date is: {{ ansible_facts['date_time']['date'] }}.

Only use this system with permission.
You can ask {{ system_owner }} for access.
```

The role defines a default value for the `system_owner` variable. The `defaults/main.yml` file in the role's directory structure is where this value is set.

The following `defaults/main.yml` file sets the `system_owner` variable to `user@host.example.com`. This email address is written in the `/etc/motd` file of managed hosts when this role is applied.

```
[user@host ~]$ cat roles/motd/defaults/main.yml
---
system_owner: user@host.example.com
```

Recommended Practices for Role Content Development

Roles allow you to break down playbooks into multiple files, resulting in reusable code. To maximize the effectiveness of newly developed roles, consider implementing the following recommended practices into your role development:

- Maintain each role in its own version control repository. Ansible works well with Git-based repositories.
- Use variables to configure roles so that you can reuse the role to perform similar tasks in similar circumstances.
- Avoid storing sensitive information in a role, such as passwords or SSH keys. Configure role variables that are used to contain sensitive values when called in a play with default values that are not sensitive. Playbooks that use the role are responsible for defining sensitive variables through Ansible Vault variable files or other methods.
- Use `ansible-galaxy init` command to start your role, then remove any unnecessary files and directories.
- Create and maintain `README.md` and `meta/main.yml` files to document the role's purpose, author, and usage.
- Keep your role focused on a specific purpose or function. Instead of making one role do many things, write more than one role.
- Reuse roles often.

Resist creating new roles for edge configurations. If an existing role accomplishes most of the required configuration, refactor the existing role to integrate the new configuration scenario.

Use integration and regression testing techniques to ensure that the role provides the required new functionality and does not cause problems for existing playbooks.

**Note**

A longer unofficial list of good practices to follow when you write a role is available from https://redhat-cop.github.io/automation-good-practices/#_roles_good_practices_for_ansible.

Changing a Role's Behavior with Variables

A well-written role uses default variables to alter the role's behavior to match a related configuration scenario. Roles that use variables are more generic and reusable in a variety of contexts.

The value of any variable defined in a role's `defaults` directory is overwritten if that same variable is defined:

- In an inventory file, either as a host variable or a group variable.
- In a YAML file under the `group_vars` or `host_vars` directories of a playbook project.
- As a variable nested in the `vars` keyword of a play.
- As a variable when including the role in `roles` keyword of a play.

The following example shows how to use the `motd` role with a different value for the `system_owner` role variable. The value specified, `someone@host.example.com`, replaces the variable reference when the role is applied to a managed host.

```
[user@host ~]$ cat use-motd-role.yml
---
- name: use motd role playbook
  hosts: remote.example.com
  remote_user: devops
  become: true
  vars:
    system_owner: someone@host.example.com
  roles:
    - role: motd
```

When defined in this way, the `system_owner` variable replaces the value of the default variable of the same name. Any variable definitions nested within the `vars` keyword do not replace the value of the same variable if defined in a role's `vars` directory.

The following example also shows how to use the `motd` role with a different value for the `system_owner` role variable. The value specified, `someone@host.example.com`, replaces the variable reference regardless of being defined in the role's `vars` or `defaults` directory.

```
[user@host ~]$ cat use-motd-role.yml
---
- name: use motd role playbook
  hosts: remote.example.com
  remote_user: devops
  become: true
```

```
roles:
  - role: motd
    system_owner: someone@host.example.com
```

Important

Variable precedence can be confusing when working with role variables in a play.

- Most other variables override a role's default variables: inventory variables, play vars, inline *role parameters*, and so on.
- Fewer variables can override variables defined in a role's vars directory. Facts, variables loaded with `include_vars`, registered variables, and role parameters can override these variables. Inventory variables and play vars cannot. This behavior is important because it helps keep your play from accidentally changing the internal functioning of the role.
- Variables declared inline as role parameters have very high precedence; they can also override variables defined in a role's vars directory.

If a role parameter has the same name as a variable set in play vars, a role's vars, or an inventory or playbook variable, the role parameter overrides the other variable.

Defining Role Dependencies

Role dependencies allow a role to include other roles as dependencies.

For example, a role that defines a documentation server might depend upon another role that installs and configures a web server.

Dependencies are defined in the `meta/main.yml` file in the role directory hierarchy.

The following is a sample `meta/main.yml` file.

```
---
dependencies:
  - role: apache
    port: 8080
  - role: postgres
    dbname: serverlist
    admin_user: felix
```

Note

A `meta/main.yml` file might also have a top-level `galaxy_info` key that has a dictionary of other attributes that specify the author, purpose, license, and the versions of Ansible Core and operating systems that the role supports.

By default, if multiple roles have a dependency on a role, and that role is called by different roles in the play multiple times with the same attributes, then the role only runs the first time it appears. This behavior can be overridden by setting the `allow_duplicates` variable to `yes` in your role's `meta/main.yml` file.



Important

Limit your role's dependencies on other roles. Dependencies make it harder to maintain your role, especially if it has many complex dependencies.



References

Using Roles – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html#using-roles

Using Variables – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html

Roles Good Practices for Ansible

https://redhat-cop.github.io/automation-good-practices/#_roles_good_practices_for_ansible

► Guided Exercise

Creating Roles

In this exercise, you create an Ansible role that uses variables, files, templates, tasks, and handlers to deploy a network service.

Outcomes

- Create a role that uses variables and parameters.

The `myvhost` role installs and configures the Apache service on a host. A template named `vhost.conf.j2` is provided to generate `/etc/httpd/conf.d/vhost.conf`.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start role-create
```

Instructions

- 1. Change into the `/home/student/role-create` directory.

```
[student@workstation ~]$ cd ~/role-create  
[student@workstation role-create]$
```

- 2. Create the directory structure for a role called `myvhost`. The role includes fixed files, templates, tasks, and handlers. The `defaults`, `vars`, and `tests` directories are not used in this role, so you can delete them.

```
[student@workstation role-create]$ mkdir -v roles; cd roles  
mkdir: created directory 'roles'  
[student@workstation roles]$ ansible-galaxy init myvhost  
- Role myvhost was created successfully  
[student@workstation roles]$ rm -rfv myvhost/{defaults,vars,tests}  
removed 'myvhost/defaults/main.yml'  
removed directory: 'myvhost/defaults'  
removed 'myvhost/vars/main.yml'  
removed directory: 'myvhost/vars'  
removed 'myvhost/tests/inventory'  
removed 'myvhost/tests/test.yml'  
removed directory: 'myvhost/tests'  
[student@workstation roles]$ cd ..  
[student@workstation role-create]$
```

- 3. Edit the `main.yml` file in the `tasks` subdirectory of the role. The role should perform the following tasks:

- Install the `httpd` package.
- Enable and start the `httpd` service.
- Install the web server configuration file using a template provided by the role.

- 3.1. Edit the `roles/myvhost/tasks/main.yml` file. Include code to use the `ansible.builtin.dnf` module to install the `httpd` package. Ensure that the file contains the following content:

```
---
# tasks file for myvhost

- name: Ensure httpd is installed
  ansible.builtin.dnf:
    name: httpd
    state: latest
```

- 3.2. Add additional code to the `roles/myvhost/tasks/main.yml` file to use the `ansible.builtin.service` module to start and enable the `httpd` service.

```
- name: Ensure httpd is started and enabled
  ansible.builtin.service:
    name: httpd
    state: started
    enabled: true
```

- 3.3. Add another stanza to use the `ansible.builtin.template` module to create `/etc/httpd/conf.d/vhost.conf` on the managed host. The module calls a handler to restart the `httpd` daemon when this file is updated.

```
- name: vhost file is installed
  ansible.builtin.template:
    src: vhost.conf.j2
    dest: /etc/httpd/conf.d/vhost.conf
    owner: root
    group: root
    mode: 0644
  notify:
    - restart httpd
```

- 3.4. Save your changes and exit the `roles/myvhost/tasks/main.yml` file.

- 4. Create the handler for restarting the `httpd` service. Edit the `roles/myvhost/handlers/main.yml` file and include code to use the `ansible.builtin.service` module, then save and exit. Ensure that the file contains the following content:

```
---
# handlers file for myvhost

- name: restart httpd
  ansible.builtin.service:
    name: httpd
    state: restarted
```

- 5. The `vhost.conf.j2` file is a template that is used to configure the Apache web server by using variables:

```
# {{ ansible_managed }}

<VirtualHost *:80>
  ServerAdmin webmaster@{{ ansible_fqdn }}
  ServerName {{ ansible_fqdn }}
  ErrorLog logs/{{ ansible_hostname }}-error.log
  CustomLog logs/{{ ansible_hostname }}-common.log common
  DocumentRoot /var/www/vhosts/{{ ansible_hostname }}/

  <Directory /var/www/vhosts/{{ ansible_hostname }}/>
    Options +Indexes +FollowSymlinks +Includes
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

Move the `vhost.conf.j2` template from the project directory to the role's `templates` subdirectory.

```
[student@workstation role-create]$ mv -v vhost.conf.j2 roles/myvhost/templates/
renamed 'vhost.conf.j2' -> 'roles/myvhost/templates/vhost.conf.j2'
```

- 6. Create the HTML content to be served by the web server.

6.1. Create the `files/html/` directory to store the content in.

```
[student@workstation role-create]$ mkdir -pv files/html
mkdir: created directory 'files'
mkdir: created directory 'files/html'
```

6.2. Create an `index.html` file below that directory with the contents: `simple index`.

```
[student@workstation role-create]$ echo 'simple index' > files/html/index.html
```

- 7. Test the `myvhost` role to make sure that it works properly.

7.1. Write a playbook that uses the role, called `use-vhost-role.yml`. Include a task to copy the HTML content from the `files/html/` directory. Use the `ansible.builtin.copy` module and include a trailing slash (/) after the source directory name. Ensure that the file contains the following content:

```

---
- name: Use myvhost role playbook
hosts: webservers
pre_tasks:
  - name: pre_tasks message
    ansible.builtin.debug:
      msg: 'Ensure web server configuration.'

roles:
  - myvhost

post_tasks:
  - name: HTML content is installed
    ansible.builtin.copy:
      src: files/html/
      dest: "/var/www/vhosts/{{ ansible_hostname }}"

  - name: post_tasks message
    ansible.builtin.debug:
      msg: 'Web server is configured.'

```



Note

The trailing slash causes the source directory and all of its contents to be copied to the managed host.

- 7.2. Before running the `use-vhost-role.yml` playbook, verify that its syntax is correct by running the `ansible-navigator` command with the `--syntax-check` option. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation role-create]$ ansible-navigator run \
> -m stdout use-vhost-role.yml --syntax-check
playbook: /home/student/role-create/use-vhost-role.yml
```

- 7.3. Run the `use-vhost-role.yml` playbook. Review the output to confirm that Ansible performed the actions on the `servera` web server.

```
[student@workstation role-create]$ ansible-navigator run \
> -m stdout use-vhost-role.yml

PLAY [Use myvhost role playbook] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [pre_tasks message] ****
ok: [servera.lab.example.com] => {
    "msg": "Ensure web server configuration."
}

TASK [myvhost : Ensure httpd is installed] ****
```

```

changed: [servera.lab.example.com]

TASK [myvhost : Ensure httpd is started and enabled] ****
changed: [servera.lab.example.com]

TASK [myvhost : vhost file is installed] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [myvhost : restart httpd] ****
changed: [servera.lab.example.com]

TASK [HTML content is installed] ****
changed: [servera.lab.example.com]

TASK [post_tasks message] ****
ok: [servera.lab.example.com] => {
    "msg": "Web server is configured."
}

PLAY RECAP ****
servera.lab.example.com      : ok=8    changed=5    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0

```

- 7.4. Run the `verify-httpd.yml` playbook to confirm that the role worked. The `httpd` package should be installed and the `httpd` service should be enabled and running.

```

[student@workstation role-create]$ ansible-navigator run \
> -m stdout verify-httpd.yml

PLAY [Verify the httpd service] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Verify the httpd service is installed] ****
changed: [servera.lab.example.com]

TASK [Is the httpd service installed] ****
ok: [servera.lab.example.com] => {
    "msg": "{'changed': True, 'stdout': 'httpd-2.4.51-7.el9_0.x86_64',
    'stderr': '', 'rc': 0, 'cmd': ['rpm', '-q', 'httpd'], 'start': '2022-07-18
    15:56:40.412987', 'end': '2022-07-18 15:56:40.429762', 'delta': '0:00:00.016775',
    'msg': '', 'stdout_lines': ['httpd-2.4.51-7.el9_0.x86_64'], 'stderr_lines': [],
    'failed': False}.stdout"
}

TASK [Verify the httpd service is started] ****
changed: [servera.lab.example.com]

TASK [Is the httpd service started] ****
ok: [servera.lab.example.com] => {

```

```
"msg": "{'changed': True, 'stdout': 'active', 'stderr': '', 'rc': 0, 'cmd': ['systemctl', 'is-active', 'httpd'], 'start': '2022-07-18 15:56:40.853778', 'end': '2022-07-18 15:56:40.863193', 'delta': '0:00:00.009415', 'msg': '', 'stdout_lines': ['active'], 'stderr_lines': [], 'failed': False}.stdout"}\n\nTASK [Verify the httpd service is enabled] ****\nchanged: [servera.lab.example.com]\n\nTASK [Is the httpd service enabled] ****\nok: [servera.lab.example.com] => {\n    "msg": "{'changed': True, 'stdout': 'enabled', 'stderr': '', 'rc': 0, 'cmd': ['systemctl', 'is-enabled', 'httpd'], 'start': '2022-07-18 15:56:41.282211', 'end': '2022-07-18 15:56:41.291881', 'delta': '0:00:00.009670', 'msg': '', 'stdout_lines': ['enabled'], 'stderr_lines': [], 'failed': False}.stdout"}\n\nPLAY RECAP ****\nservera.lab.example.com : ok=7    changed=3    unreachable=0    failed=0\nskipped=0    rescued=0    ignored=0
```

- 7.5. Run the `verify-config.yml` playbook to confirm that the Apache configuration file is deployed and that all the variables in the template expanded correctly.

```
[student@workstation role-create]$ ansible-navigator run \
> -m stdout verify-config.yml\n\nPLAY [Verify the httpd config] ****\n\nTASK [Gathering Facts] ****\nok: [servera.lab.example.com]\n\nTASK [Verify the httpd config file is in place] ****\nchanged: [servera.lab.example.com]\n\nTASK [What does the httpd config file contain] ****\nok: [servera.lab.example.com] => {\n    "msg": "{'changed': True, 'stdout': '# Ansible managed\\n\\n<VirtualHost\n*:80>\\n    ServerAdmin webmaster@servera.lab.example.com\\n\nServerName servera.lab.example.com\\n    ErrorLog logs/servera-error.log\\n    CustomLog logs/servera-common.log common\\n    DocumentRoot /var/www/vhosts/servera/\\n\\n    <Directory /var/www/vhosts/servera/>\\n\\tOptions +Indexes +FollowSymlinks +Includes\\n\\tOrder allow,deny\\n\\tAllow from all\\n    </Directory>\\n</VirtualHost>', 'stderr': '', 'rc': 0, 'cmd': ['cat', '/etc/httpd/conf.d/vhost.conf'], 'start': '2022-07-18 16:15:11.441593', 'end': '2022-07-18 16:15:11.445100', 'delta': '0:00:00.003507', 'msg': '', 'stdout_lines': ['# Ansible managed', '', '<VirtualHost *:80>', '    ServerAdmin webmaster@servera.lab.example.com', '    ErrorLog logs/servera-error.log', '    CustomLog logs/servera-common.log common', '    DocumentRoot /var/www/vhosts/servera/', '', '<Directory /var/www/vhosts/servera/>', '\\tOptions +Indexes +FollowSymlinks +Includes', '\\tOrder allow,deny', '\\tAllow from all', '</Directory>', '</VirtualHost>'], 'stderr_lines': [], 'failed': False}.stdout_lines"}\n\n
```

```
PLAY RECAP ****
servera.lab.example.com    : ok=3      changed=1      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0
```

- 7.6. The HTML content is served from a directory called /var/www/vhosts/servera. Run the `verify-content.yml` playbook to confirm that the `index.html` file in that directory contains the string "simple index".

```
[student@workstation role-create]$ ansible-navigator run \
> -m stdout verify-content.yml

PLAY [Verify the index.html file] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Verify the index.html file is in place] ****
changed: [servera.lab.example.com]

TASK [What does the index.html config file contain] ****
ok: [servera.lab.example.com] => {
    "msg": {"'changed': True, 'stdout': 'simple index', 'stderr': '', 'rc': 0, 'cmd': ['cat', '/var/www/vhosts/servera/index.html'], 'start': '2022-07-18 16:24:54.665447', 'end': '2022-07-18 16:24:54.669959', 'delta': '0:00:00.004512', 'msg': '', 'stdout_lines': ['simple index'], 'stderr_lines': [], 'failed': False}.stdout_lines"
}

PLAY RECAP ****
servera.lab.example.com    : ok=3      changed=1      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0
```

- 7.7. Confirm that the web server content is available.

```
[student@workstation role-create]$ curl http://servera.lab.example.com
simple index
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish role-create
```

This concludes the section.

Deploying Roles from External Content Sources

Objectives

- Select and retrieve roles from external sources such as Git repositories or Ansible Galaxy, and use them in your playbooks.

External Content Sources

If you are using roles in your Ansible Playbooks, then you should get those roles from some centrally managed source. This practice ensures that all your projects have the current version of the role and that each project can benefit from bug fixes discovered by other projects with which they share the role.

For example, if the role is private content created and curated by your organization, it might be stored in a Git repository managed by your organization. Alternatively, it might be distributed as a tar archive file from a website or through other means.

In addition, the open source community maintains some roles through the Ansible Galaxy website. These roles are not reviewed or supported officially by Red Hat, but might contain code that your organization finds useful.



Important

It is increasingly common for roles to be packaged as Ansible Content Collections and offered by their authors through various methods:

- In Red Hat Certified Ansible Content Collections from the Red Hat hosted automation hub at <https://console.redhat.com>, or through a private automation hub
- As private content packaged from a private automation hub
- By the community from Ansible Galaxy at <https://galaxy.ansible.com>

This section only covers how to get roles that are not packaged into an Ansible Content Collections.

Ideally, you want to be able to download the latest version of the role into the `roles` directory of your project simply, whenever necessary.

Introducing Ansible Galaxy

A public library of Ansible content written by a variety of Ansible administrators and users is available at <https://galaxy.ansible.com>[Ansible Galaxy].

This library contains thousands of Ansible roles, and it has a searchable database that helps you identify roles that might help you accomplish an administrative task.

The `ansible-galaxy` command that you use to download and manage roles from Ansible Galaxy can also be used to download and manage roles from your own Git repositories.

The Ansible Galaxy Command Line Tool

The `ansible-galaxy` command-line tool can be used to search for, display information about, install, list, remove, or initialize roles.

Installing Roles Using a Requirements File

If you have a playbook that must have specific roles installed, then you can create a `roles/requirements.yml` file in the project directory that specifies which roles are needed. This file acts as a dependency manifest for the playbook project that enables playbooks to be developed and tested separately from any supporting roles.

Then, before you run `ansible-navigator run`, you can use the `ansible-galaxy` command to install those roles in your project's `roles` directory.



Note

If you use automation controller, it automatically downloads roles specified in your `roles/requirements.yml` file when it runs your playbook.

For example, you could have a role in a public repository on a Git server at `https://git.example.com/someuser/someuser.myrole`. A simple `requirements.yml` to install `someuser.myrole` might contain the following content:

```
- src: https://git.example.com/someuser/someuser.myrole
  scm: git
  version: "1.5.0"
```

The `src` attribute specifies the source of the role, in this case the URL for the repository of the role on your Git server. You can also use SSH key-based authentication by specifying something like `git@git.example.com:someuser/someuser.myrole` as provided by your Git repository.

The `scm` attribute indicates that this role is from a Git repository.

The `version` attribute is optional, and specifies the version of the role to install, in this case `1.5.0`. In the case of a role stored in Git, the version can be the name of a branch, a tag, or a Git commit hash. If you do not specify a version, the command uses the latest commit on the default branch.



Important

Specify the version of the role in your `requirements.yml` file, especially for playbooks in production.

If you do not specify a version, you get the latest version of the role.

If the upstream author makes changes to the role that are incompatible with your playbook, it might cause an automation failure or other problems.

To install roles using a role requirements file, run the `ansible-galaxy role install` command from within your project directory. Include the following options:

- `-r roles/requirements.yml` to specify the location of your requirements file
- `-p roles` to install the role into a subdirectory of the `roles` directory

```
[user@host project]$ ansible-galaxy role install -r roles/requirements.yml \
> -p roles
Starting galaxy role install process
- downloading role from https://git.example.com/someuser/someuser.myrole
- extracting myrole to /home/user/project/roles/someuser.myrole
- someuser.myrole (1.5.0) was installed successfully
```



Important

If you do not specify the `-p roles` option, then `ansible-galaxy` uses the first directory in the default `roles_path` setting to determine where to install the role. This defaults to the user's `~/.ansible/roles` directory, which is outside the project directory and unavailable to the execution environment if you use `ansible-navigator` to run your playbooks.

One way to avoid the need to specify `-p roles` is to apply the following setting in the `defaults` section of your project's `ansible.cfg` file:

```
roles_path = roles
```

If you have a `tar` archive file that contains a role, you can use `roles/requirements.yml` to install that file from a URL:

```
# from a role tar ball, given a URL;
# supports 'http', 'https', or 'file' protocols
- src: file:///opt/local/roles/tarrole.tar
  name: tarrole

- src: https://www.example.com/role-archive/someuser.otherrole.tar
  name: someuser.otherrole
```



Note

Red Hat recommends that you use version control with roles, storing them in a version control system such as Git. If a recent change to a role causes problems, using version control allows you to roll back to a previous, stable version of the role.

Finding Community-managed Roles in Ansible Galaxy

The open source Ansible community operates a public server, <https://galaxy.ansible.com>, that contains roles and Ansible Content Collections shared by other Ansible users.



Warning

These roles can be useful, but they are not supported by Red Hat, nor does Red Hat audit or review this code. Volunteers in the Ansible community maintain this site. Use these roles at your own risk.

Browsing Ansible Galaxy for Roles

The Search tab on the left side of the Ansible Galaxy website home page gives you access to information about the roles published on Ansible Galaxy. You can search for an Ansible role by name or by using tags or other role attributes.

Results are presented in descending order of the Best Match score, which is a computed score based on role quality, role popularity, and search criteria. (Content Scoring [https://galaxy.ansible.com/docs/contributing/content_scoring.html] in the Ansible Galaxy documentation has more information on how roles are scored by Ansible Galaxy.)

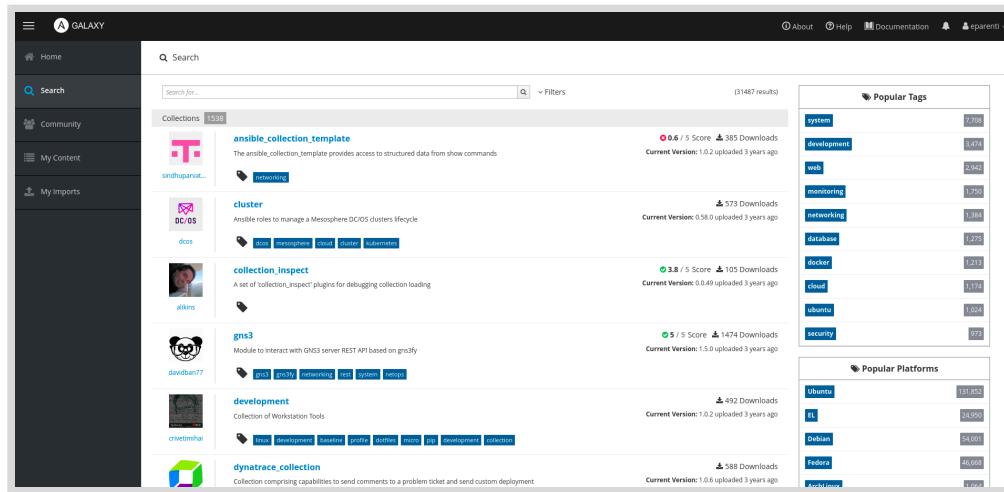


Figure 7.1: Ansible Galaxy search screen

Ansible Galaxy reports the number of times each role has been downloaded from Ansible Galaxy. In addition, Ansible Galaxy also reports the number of watchers, forks, and stars the role's GitHub repository has. You can use this information to help determine how active development is for a role and how popular it is in the community.

The following figure shows the search results that Ansible Galaxy displayed after a keyword search for `redis` was performed.

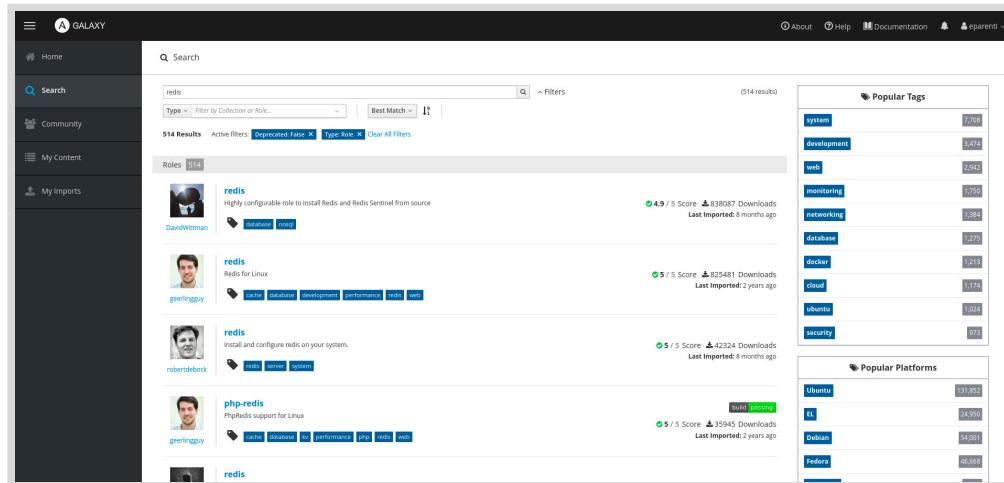


Figure 7.2: Ansible Galaxy search results example

The **Filters** menu to the right of the search box allows searches by type, contributor type, contributor, cloud platform, deprecated, platform, and tags.

Possible platform values include `EL` for Red Hat Enterprise Linux (and closely related distributions such as CentOS) and `Fedora`, among others.

Tags are arbitrary single-word strings set by the role author that describe and categorize the role. You can use tags to find relevant roles. Possible tag values include `system`, `development`, `web`, `monitoring`, and others. In Ansible Galaxy, a role can have up to 20 tags.



Important

In the Ansible Galaxy search interface, keyword searches match words or phrases in the `README` file, content name, or content description. Tag searches, by contrast, specifically match tag values that the author set for the role.

Searching for Roles from the Command Line

The `ansible-galaxy search` subcommand searches Ansible Galaxy for roles.

If you specify a string as an argument, it is used to search Ansible Galaxy for roles by keyword. You can use the `--author`, `--platforms`, and `--galaxy-tags` options to narrow the search results. You can also use those options as the main search key.

For example, the command `ansible-galaxy search --author geerlingguy` displays all roles submitted by the user `geerlingguy`. Results are displayed in alphabetical order, not by descending Best Match score.

The following example displays the names of roles that include `redis`, and are available for the Enterprise Linux (EL) platform.

```
[user@host ~]$ ansible-galaxy search 'redis' --platforms EL

Found 251 roles matching your search:

  Name                                Description
  -----
  0x0i.consul                          Consul - a service discovery,
  mesh and config>
  0x0i.grafana                         Grafana - an analytics and
  monitoring observ>
  0x5a17ed.ansible_role_netbox        Installs and configures NetBox, a
  DCIM suite>
  1it.sudo                             Ansible role for managing sudoers
  adfinis-sygroup.redis                Ansible role for Redis
  AerisCloud.librato                   Install and configure the Librato
  Agent                               Agent
  AerisCloud.redis                     Installs redis on a server
  AlbanAndrieu.java                  Manage Java installation
  alikins.php_pecl                   PHP PECL extension installation.
  ...output omitted...
```

The `ansible-galaxy info` subcommand displays more detailed information about a role. Ansible Galaxy gets this information from a number of places, including the role's `meta/main.yml` file and its GitHub repository.

The following command displays information about the `geerlingguy.redis` role, available from Ansible Galaxy.

```
[user@host ~]$ ansible-galaxy info gearlingguy.redis

Role: gearlingguy.redis
  description: Redis for Linux
  active: True
...output omitted...
  download_count: 825486
  forks_count: 137
  github_branch: master
  github_repo: ansible-role-redis
  github_user: gearlingguy
...output omitted...
  license: license (BSD, MIT)
  min_ansible_version: 2.4
  modified: 2020-11-17T19:20:46.649130Z
  open_issues_count: 5
  path: ('/home/student/.ansible/roles', '/usr/share/ansible/roles', '/etc/
ansible/role>
  role_type: ANS
  stargazers_count: 170
...output omitted...
```

Downloading Roles from Ansible Galaxy

The following example shows how to configure a requirements file that uses a variety of remote sources.

```
[user@host project]$ cat roles/requirements.yml
# from Ansible Galaxy, using the latest version
- src: gearlingguy.redis ①

# from Ansible Galaxy, overriding the name and using a specific version
- src: gearlingguy.redis
  version: "1.5.0" ②
  name: redis_prod

# from any Git based repository, using HTTPS
- src: https://github.com/gearlingguy/ansible-role-nginx.git
  scm: git ③
  version: master
  name: nginx

# from a role tar ball, given a URL;
#   supports 'http', 'https', or 'file' protocols
- src: file:///opt/local/roles/myrole.tar ④
  name: myrole ⑤
```

- ① The `src` keyword specifies the Ansible Galaxy role name. If the role is not hosted on Ansible Galaxy, the `src` keyword indicates the role's URL.
- ② The `version` keyword is used to specify a role's version. The `version` keyword can be any value that corresponds to a branch, tag, or commit hash from the role's software repository.

- ③ If the role is hosted in a source control repository, the `scm` attribute is required.
- ④ If the role is hosted on Ansible Galaxy or as a tar archive, the `scm` keyword is omitted.
- ⑤ The `name` keyword is used to override the local name of the role.

Managing Downloaded Roles

The `ansible-galaxy` command can also manage local roles, such as those roles found in the `roles` directory of a playbook project. The `ansible-galaxy list` subcommand lists the local roles.

```
[user@host project]$ ansible-galaxy list  
# /home/user/project/roles  
- geerlingguy.redis, 1.7.0  
- redis_prod, 1.5.0  
- nginx, master  
- myrole, (unknown version)  
...output omitted...
```

You can remove a role with the `ansible-galaxy remove` subcommand.

```
[user@host ~]$ ansible-galaxy remove nginx  
- successfully removed nginx
```



References

ansible-galaxy – Ansible Documentation

<https://docs.ansible.com/ansible/latest/cli/ansible-galaxy.html>

Red Hat Hybrid Cloud Console | Ansible Automation Platform Dashboard

<https://console.redhat.com/ansible/ansible-dashboard>

► Guided Exercise

Deploying Roles from External Content Sources

In this exercise, you use the `ansible-galaxy` command to download and install an Ansible role.

Outcomes

- Create a requirements file to specify role dependencies for a playbook.
- Install roles specified in a requirements file.
- List downloaded roles by using the `ansible-galaxy` command.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start role-galaxy
```

Instructions

- 1. Change into the `/home/student/role-galaxy` directory.

```
[student@workstation ~]$ cd ~/role-galaxy  
[student@workstation role-galaxy]$
```

- 2. Create a role requirements file in your project directory that downloads the `student.bash_env` role. This role configures the default initialization files for the Bash shell that are used for newly created users and stored in the `/etc/skel` directory. The role adjusts the shell prompt and configures an environment variable that is used by the `less` command.

Create a file called `requirements.yml` in the `roles` subdirectory. The URL of the role's Git repository is: `git@workstation.lab.example.com:student/bash_env`.

To see how the role affects the behavior of production hosts, use the `main` branch of the repository. Set the local name of the role to `student.bash_env`.

Ensure that the `roles/requirements.yml` file contains the following content:

```
---
# requirements.yml

- src: git@workstation.lab.example.com:student/bash_env
  scm: git
  version: main
  name: student.bash_env
```

- ▶ 3. Use the `ansible-galaxy` command to process the new requirements file that installs the `student.bash_env` role.

- 3.1. List the contents of the `roles` subdirectory before the role is installed, so that you can compare its current state to command results.

```
[student@workstation role-galaxy]$ ls roles/
requirements.yml
```

- 3.2. Use the `ansible-galaxy` command to download and install the roles listed in the `roles/requirements.yml` file into the `roles` subdirectory of your project.

```
[student@workstation role-galaxy]$ ansible-galaxy install \
> -r roles/requirements.yml -p roles
Starting galaxy role install process
- extracting student.bash_env to /home/student/role-galaxy/roles/student.bash_env
- student.bash_env (main) was installed successfully
```

- 3.3. List the contents of the `roles` subdirectory after the role is installed. Confirm that a new subdirectory called `student.bash_env`, matching the `name` value specified in the requirements file, is present.

```
[student@workstation role-galaxy]$ ls roles/
requirements.yml  student.bash_env
```

- 3.4. Use the `ansible-galaxy list` command to list the project roles in the `roles` subdirectory.

```
[student@workstation role-galaxy]$ ansible-galaxy list -p roles
# /home/student/role-galaxy/roles
- student.bash_env, main
# /usr/share/ansible/roles
- linux-system-roles.certificate, (unknown version)
- linux-system-roles.cockpit, (unknown version)
- linux-system-roles.crypto_policies, (unknown version)
- linux-system-roles.firewall, (unknown version)
- linux-system-roles.ha_cluster, (unknown version)
- linux-system-roles.kdump, (unknown version)
- linux-system-roles.kernel_settings, (unknown version)
...output omitted...
[WARNING]: - the configured path /home/student/.ansible/roles does not exist.
```

Important

If you do not specify the option `-p roles` to your `ansible-galaxy list` command, the role that you installed is not listed because the `roles` subdirectory is not in your default `roles_path`.

```
[student@workstation role-galaxy]$ ansible-galaxy list
# /usr/share/ansible/roles
- linux-system-roles.certificate, (unknown version)
- linux-system-roles.cockpit, (unknown version)
- linux-system-roles.crypto_policies, (unknown version)
- linux-system-roles.firewall, (unknown version)
- linux-system-roles.ha_cluster, (unknown version)
- linux-system-roles.kdump, (unknown version)
- linux-system-roles.kernel_settings, (unknown version)
...output omitted...
[WARNING]: - the configured path /home/student/.ansible/roles does not exist.
```

- ▶ 4. Create a playbook named `use-bash_env-role.yml` that uses the `student.bash_env` role. The playbook must have the following contents:

```
---
- name: Use student.bash_env role playbook
hosts: devservers
vars:
  default_prompt: '[\u on \h in \W dir]\$ '
pre_tasks:
  - name: Ensure test user does not exist
    ansible.builtin.user:
      name: student2
      state: absent
      force: yes
      remove: yes

roles:
  - student.bash_env

post_tasks:
  - name: Create the test user
```

```
ansible.builtin.user:
  name: student2
  state: present
  password: "{{ 'redhat' | password_hash('sha512', 'mysecretsalt') }}"
```

You must create a user account to see the effects of the configuration change. The `pre_tasks` and `post_tasks` section of the playbook ensure that the `student2` user account is deleted and created each time the playbook is run.

When you run the `use-bash_env-role.yml` playbook, the `student2` account is created with a password of `redhat`.



Note

The `student2` password is generated using a filter. Filters take data and modify it; here, the string `redhat` is modified by passing it to the `password_hash` filter in order to convert the value into a SHA512-protected password hash. Filters are an advanced topic not covered in this course.

- ▶ 5. Run the `use-bash_env-role.yml` playbook.

The `student.bash_env` role creates standard template configuration files in `/etc/skel` on the managed host. The files it creates include `.bashrc`, `.bash_profile`, and `.vimrc`.

```
[student@workstation role-galaxy]$ ansible-navigator run \
> -m stdout use-bash_env-role.yml

PLAY [Use student.bash_env role playbook] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Ensure test user does not exist] ****
ok: [servera.lab.example.com]

TASK [student.bash_env : put away .bashrc] ****
changed: [servera.lab.example.com]

TASK [student.bash_env : put away .bash_profile] ****
changed: [servera.lab.example.com]

TASK [student.bash_env : put away .vimrc] ****
changed: [servera.lab.example.com]

TASK [Create the test user] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=6    changed=4    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

- ▶ 6. Use SSH to log in to the `servera` machine as the `student2` user. Observe the custom prompt for the `student2` user, and then disconnect from the `servera` machine.

```
[student@workstation role-galaxy]$ ssh student2@servera
...output omitted...
[student2 on servera in ~ dir]$ exit
logout
Connection to servera closed.
[student@workstation role-galaxy]$
```

- ▶ 7. Run the playbook using the development version of the `student.bash_env` role.

The development version of the role is located in the `dev` branch of the Git repository. The development version of the role uses a new variable, `prompt_color`.

Before running the playbook, add the `prompt_color` variable to the `vars` section of the playbook and set its value to `blue`.

- 7.1. Update the `roles/requirements.yml` file, and set the `version` value to `dev`. Ensure that the `roles/requirements.yml` file contains the following content:

```
---
# requirements.yml

- src: git@workstation.lab.example.com:student/bash_env
  scm: git
  version: dev
  name: student.bash_env
```

- 7.2. Modify the `~/role-galaxy/ansible.cfg` and add the `roles_path` setting to the `[defaults]` section of the file. This sets the default roles path and enable you to omit the `-p roles` option when typing `ansible-galaxy` commands.

When completed, the file contains then following content:

```
[defaults]
inventory=inventory
remote_user=devops
become_ask_pass=False
roles_path=roles

[privilege_escalation]
become=True
become_method=sudo
become_user=root
```

- 7.3. Remove the existing version of the `student.bash_env` role from the `roles` subdirectory.

```
[student@workstation role-galaxy]$ ansible-galaxy remove student.bash_env
- successfully removed student.bash_env
```

- 7.4. Use the `ansible-galaxy install` command to install the role by using the updated requirements file.

```
[student@workstation role-galaxy]$ ansible-galaxy install \
> -r roles/requirements.yml
Starting galaxy role install process
- extracting student.bash_env to /home/student/role-galaxy/roles/student.bash_env
- student.bash_env (dev) was installed successfully
```

- 7.5. Modify the `use-bash_env-role.yml` file. Add the `prompt_color` variable with a value of blue to the `vars` section of the playbook. Ensure that the file contains the following content:

```
---
- name: Use student.bash_env role playbook
hosts: devservers
vars:
  prompt_color: blue
  default_prompt: '[\u on \h in \w dir]\$ '
pre_tasks:
...output omitted...
```

- 7.6. Run the `use-bash_env-role.yml` playbook.

```
[student@workstation role-galaxy]$ ansible-navigator run \
> -m stdout use-bash_env-role.yml

PLAY [Use student.bash_env role playbook] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Ensure test user does not exist] ****
changed: [servera.lab.example.com]

TASK [student.bash_env : put away .bashrc] ****
changed: [servera.lab.example.com]

TASK [student.bash_env : put away .bash_profile] ****
ok: [servera.lab.example.com]

TASK [student.bash_env : put away .vimrc] ****
ok: [servera.lab.example.com]

TASK [Create the test user] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=6      changed=3      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0
```

- 8. Use SSH to log in to the `servera` machine as the `student2` user. The custom prompt for the `student2` user now displays with blue characters.

```
[student@workstation role-galaxy]$ ssh student2@servera  
...output omitted...  
[student2 on servera in ~ dir]$
```

► **9.** Log out from servera.

```
[student2 on servera in ~ dir]$ exit  
logout  
Connection to servera closed.  
[student@workstation role-galaxy]$
```

Finish

On the workstation machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish role-galaxy
```

This concludes the section.

Getting Roles and Modules from Content Collections

Objectives

- Obtain a set of related roles, supplementary modules, and other content from an Ansible Content Collection and use them in a playbook.

Ansible Content Collections

When Ansible was first developed, all the modules that it used were included as part of the core software package. As the number of modules increased, it became harder for the upstream project to manage all of these modules. Every module required a unique name and synchronizing module updates with updates to the core Ansible code.

With *Ansible Content Collections*, Ansible code updates are separated from updates to modules and plug-ins. An Ansible Content Collection provides a set of related modules, roles, and other plug-ins that you can use in your playbooks. This approach enables vendors and developers to maintain and distribute their collections at their own pace, independently of Ansible releases.

For example:

- The `redhat.insights` content collection provides modules and roles that you can use to register a system with Red Hat Insights for Red Hat Enterprise Linux.
- The `cisco.ios` content collection, supported and maintained by Cisco, provides modules and plug-ins that manage Cisco IOS network appliances.
- The `community.crypto` content collection provides modules that create SSL/TLS certificates.

Ansible Content Collections also provide flexibility. You can install only the content you need instead of installing all supported modules.

You can also select a specific version of a collection (possibly an earlier or later one) or choose between a version of a collection supported by Red Hat or vendors, or one provided by the community.

Ansible 2.9 and later support Ansible Content Collections. Upstream Ansible unbundled most modules from the core Ansible code in Ansible Base 2.10 and Ansible Core 2.11 and placed them in collections. Red Hat Ansible Automation Platform 2.2 provides automation execution environments based on Ansible Core 2.13 that inherit this feature.



Note

You can develop your own collections to provide custom roles and modules to your teams. Creating Ansible Content Collections of your own is outside the scope of this course. You can learn more about how to create Ansible Content Collections in the course that follows this one, *Developing Advanced Automation with Red Hat Ansible Automation Platform (DO374)*.

Namespaces for Ansible Content Collections

To make it easier to specify collections and their contents by name, collection names are organized into *namespaces*. Vendors, partners, developers, and content creators can use namespaces to assign unique names to their collections without conflicting with other collections.

The namespace is the first part of a collection name. For example, all the collections that the Ansible community maintains are in the `community` namespace, and have names like `community.crypto`, `community.postgresql`, and `community.rabbitmq`. Likewise, Ansible Content Collections that Red Hat directly maintains and supports might use the `redhat` namespace, and have names like `redhat.rhv`, `redhat.satellite`, and `redhat.insights`.

Selecting Sources of Ansible Content Collections

Regardless of whether you are using `ansible-navigator` with the minimal automation execution environment or `ansible-playbook` on bare metal Ansible Core, you always have at least one Ansible Content Collection available to you: `ansible.builtin`.

In addition, your automation execution environment might have additional automation execution environments built into it, for example, the default execution environment used by Red Hat Ansible Automation Platform 2.2, `ee-supported-rhel8`.

If you need to have additional Ansible Content Collections, you can add them to the `collections` subdirectory of your Ansible project. You might obtain Ansible Content Collections from several sources:

Automation Hub

Automation hub is a service provided by Red Hat to distribute Red Hat Certified Ansible Content Collections that are supported by Red Hat and ecosystem partners. As an end customer, you can open a support ticket with Red Hat for these Ansible Content Collections that will be addressed by Red Hat and the ecosystem partner.

You need a valid Red Hat Ansible Automation Platform subscription to access automation hub. Use the automation hub web UI at <https://console.redhat.com/ansible/automation-hub/> to browse these collections.

Private Automation Hub

Your organization might have its own on-site private automation hub, and might also use that hub to distribute its own Ansible Content Collections. Private automation hub is included with Red Hat Ansible Automation Platform.

Ansible Galaxy

Ansible Galaxy is a community-supported website that hosts Ansible Content Collections that have been submitted by a variety of Ansible developers and users. Ansible Galaxy is a public library that provides no formal support guarantees and that is not curated by Red Hat. For example, the `community.crypto`, `community.postgresql`, and `community.rabbitmq` collections are all available from that platform.

Use the Ansible Galaxy web UI at <https://galaxy.ansible.com/> to search it for collections.

Third-Party Git Repository or Archive File

You can also download Ansible Content Collections from a Git repository or a local or remote tar archive file, much like you can download roles.

Installing Ansible Content Collections

Before your playbooks can use content from an Ansible Content Collection, you must ensure that the collection is available in your automation execution environment.

The Ansible configuration `collections_paths` setting specifies a colon separated list of paths on the system where Ansible looks for installed collections.

You can set this directive in the `ansible.cfg` configuration file.



Important

The following default value references the `collections_paths` directive.

```
~/ansible/collections:/usr/share/ansible/collections
```

If you set `collections_paths` to some other value in your Ansible project's `ansible.cfg` file and eliminate those two directories, then `ansible-navigator` cannot find the Ansible Content Collections provided inside the automation execution environment in its version of those directories.

The following example uses the `ansible-galaxy collection install` command to download and install the `community.crypto` Ansible Content Collection. The `-p collections` option installs the collection in the local `collections` subdirectory.

```
[user@controlnode ~]$ ansible-galaxy collection install community.crypto \
> -p collections
```



Important

You must specify the `-p collections` option or `ansible-galaxy` installs the collection based on your current `collections_paths` setting, or into your `~/ansible/collections/` directory on the control node by default. The `ansible-navigator` command does not load this directory into the automation execution environment, although this directory is available to Ansible commands that use the control node as the execution environment, such as `ansible-playbook`.

When you install the `community.crypto` collection, you could see a warning that your Ansible project's playbooks might not find the collection because the specified path is not part of the configured Ansible collections path.

You can safely ignore this warning because your Ansible project checks the local `collections` subdirectory before checking directories specified by the `collections_paths` setting and can use the collections stored there.

The command can also install a collection from a local or a remote `tar` archive, or a Git repository. A Git repository must have a valid `galaxy.yml` or `MANIFEST.json` file that provides metadata about the collection, such as its namespace and version number. For example, see the `community.general` collection at <https://github.com/ansible-collections/community.general>.

```
[user@controlnode ~]$ ansible-galaxy collection install \
> /tmp/community-dns-1.2.0.tar.gz -p collections
...output omitted...
[user@controlnode ~]$ ansible-galaxy collection install \
> http://www.example.com/redhat-insights-1.0.5.tar.gz -p collections
...output omitted...
[user@controlnode ~]$ ansible-galaxy collection install \
> git@git.example.com:organization/repo_name.git -p collections
```

Installing Ansible Content Collections with a Requirements File

If your Ansible project needs additional Ansible Content Collections, you can create a `collections/requirements.yml` file in the project directory that lists all the collections that the project requires. Automation controller detects this file and automatically installs the specified collections before running your playbooks.

A requirements file for Ansible Content Collections is a YAML file that consists of a `collections` dictionary key that has the list of collections to install as its value. Each list item can also specify the particular version of the collection to install, as shown in the following example:

```
---
collections: ①
  - name: community.crypto ②
    - name: ansible.posix
      version: 1.2.0 ③
  - name: /tmp/community-dns-1.2.0.tar.gz ④
  - name: http://www.example.com/redhat-insights-1.0.5.tar.gz ⑤
  - name: git+https://github.com/ansible-collections/community.general.git ⑥
    version: main
```

- ① The value of this dictionary key is the list of collections that are required by the Ansible project.
- ② Install the `community.crypto` Ansible Content Collection from the first available source. The next part of this section covers how to configure sources.
- ③ Install version `1.2.0` of the `ansible.posix` Ansible Content Collection from the first available source. It is a good practice to specify the version whenever you can.
- ④ Install an Ansible Content Collection from a particular local `tar` archive file.
- ⑤ Install an Ansible Content Collection from the `tar` archive at the specified remote URL.
- ⑥ Install the `community.general` Ansible Content Collection from a public Git repository, selecting the version in the `main` branch. The `version` directive can be a branch, tag, or commit hash.

The `ansible-galaxy` command can then use the `collections/requirements.yml` file to install all those collections. Specify the requirements file with the `--requirements-file` (or `-`

r) option, and use the -p collections option to install the Ansible Content Collection into the collections subdirectory.

```
[root@controlnode ~]# ansible-galaxy collection install \
> -r collections/requirements.yml -p collections
```

Configuring Ansible Content Collection Sources

By default, the `ansible-galaxy` command uses Ansible Galaxy at <https://galaxy.ansible.com/> to download Ansible Content Collections. You might not want to use this command, preferring automation hub or your own private automation hub. Alternatively, you might want to try automation hub first, and then try Ansible Galaxy.

You can configure the sources that `ansible-galaxy` uses to get Ansible Content Collections in your Ansible project's `ansible.cfg` file. The relevant parts of that file might look like the following example:

```
...output omitted...
[galaxy]
server_list = automation_hub, galaxy ①

[galaxy_server.automation_hub]
url=https://console.redhat.com/api/automation-hub/ ②
auth_url=https://sso.redhat.com/auth/realms/redhat-external/protocol/openid-
connect/token ③
token=eyJh...Jf0o ④

[galaxy_server.galaxy]
url=https://galaxy.ansible.com/
```

- ➊ List all the repositories that the `ansible-galaxy` command must use in order. For each name you define, add a `[galaxy_server.name]` section to this file that provides the connection parameters. This example configures the `ansible-galaxy` command to get Ansible Content Collections from automation hub first (`[galaxy_server.automation_hub]`). If a collection is not available from that source, then the `ansible-galaxy` command tries to get the collection from the Ansible Galaxy website (`[galaxy_server.galaxy]`).
- ➋ Provide the URL to access the repository.
- ➌ Provide the URL for authentication.
- ➍ To access automation hub, you need an authentication token associated with your account. Use the automation hub web UI at <https://console.redhat.com/ansible/automation-hub/token/> to generate that token. For more details on that process, see the links in the References section.

Instead of a token, you can use the `username` and `password` parameters to provide your customer portal username and password.

```
...output omitted...
[galaxy_server.automation_hub]
url=https://cloud.redhat.com/api/automation-hub/
username=operator1
password=Sup3r53cR3t
...output omitted...
```

However, you might not want to expose your credentials in the `ansible.cfg` file because the file could potentially get committed when using version control.

It is preferable to remove the authentication parameters from the `ansible.cfg` file and define them in environment variables, as shown in the following example:

```
export ANSIBLE_GALAXY_SERVER_<server_id>_<key>=value
```

server_id

Server identifier in uppercase. The server identifier is the name you used in the `server_list` parameter and in the name of the `[galaxy_server.server_id]` section.

key

Name of the parameter in uppercase.

The following example provides the `token` parameter as an environment variable:

```
[user@controlnode ~]$ cat ansible.cfg
...output omitted...
[galaxy_server.automation_hub]
url=https://cloud.redhat.com/api/automation-hub/
auth_url=https://sso.redhat.com/auth/realms/redhat-external/protocol/openid-
connect/token
[user@controlnode ~]$ export \
> ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_TOKEN='eyJh...Jf0o'
[user@controlnode ~]$ ansible-galaxy collection install ansible.posix \
> -p collections
```

Using Resources from Ansible Content Collections

After you install an Ansible Content Collection in your Ansible project, you can use it with that project's Ansible Playbooks.

You can use the `ansible-navigator collections` command in your Ansible project directory to list all the collections that are installed in your automation execution environment. This includes the Ansible Content Collections in your project's `collections` subdirectory.

You can select the line number of the collection in the interactive mode of `ansible-navigator` to view its contents. Then, you can select the line number of a module, role, or other plug-in to see its documentation. You can also use other tools, such as `ansible-navigator doc`, with the FQCN of a module to view that module's documentation.

The following playbook invokes the `mysql_user` module from the `community.mysql` collection for a task.

```

---
- name: Create the operator1 user in the test database
  hosts: db.example.com

  tasks:
    - name: Ensure the operator1 database user is defined
      community.mysql.mysql_user:
        name: operator1
        password: Secret0451
        priv: '.:ALL'
        state: present

```

The following playbook uses the `organizations` role from the `redhat.satellite` collection.

```

---
- name: Add the test organizations to Red Hat Satellite
  hosts: localhost

  tasks:
    - name: Ensure the organizations exist
      include_role:
        name: redhat.satellite.organizations
  vars:
    satellite_server_url: https://sat.example.com
    satellite_username: admin
    satellite_password: Sup3r53cr3t
    satellite_organizations:
      - name: test1
        label: tst1
        state: present
        description: Test organization 1
      - name: test2
        label: tst2
        state: present
        description: Test organization 2

```



References

Ansible Automation Platform Certified Content

<https://access.redhat.com/articles/3642632>

Ansible Certified Content FAQ

<https://access.redhat.com/articles/4916901>

Using collections – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/collections_using.html

Galaxy User Guide – Ansible Documentation

https://docs.ansible.com/ansible/latest/galaxy/user_guide.html

► Guided Exercise

Getting Roles and Modules from Content Collections

In this exercise, you install an Ansible Content Collection and use a role or module from that content collection in a playbook.

Outcomes

- Use the `ansible-galaxy` command to install an Ansible Content Collection.
- Use a `requirements.yml` file to install multiple Ansible Content Collections.
- Run playbooks that use roles and modules from Ansible Content Collections.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start role-collections
```

Instructions

- 1. Install and then inspect the `gls.utils` collection.

1.1. Change into the `/home/student/role-collections` directory.

```
[student@workstation ~]$ cd ~/role-collections  
[student@workstation role-collections]$
```

1.2. Install the `gls.utils` collection from the `tar` archive file in the `/home/student/role-collections/` directory.

```
[student@workstation role-collections]$ ansible-galaxy collection \  
> install /home/student/role-collections/gls-utils-0.0.1.tar.gz -p collections  
Starting galaxy collection install process  
Process install dependency map  
Starting collection install process  
Installing 'gls.utils:0.0.1' to '/home/student/role-collections/collections/  
ansible_collections/gls/utils'  
gls.utils:0.0.1 was installed successfully
```

The preceding command installs the `gls.utils` Ansible Content Collection in the `/home/student/role-collections/collections/ansible_collections/gls/utils` directory.

- 1.3. Use the `ansible-navigator collections` command to open the text-based user interface (TUI).

```
[student@workstation role-collections]$ ansible-navigator collections

      Name          Version Shadowed Type      Path
0|amazon.aws      3.2.0  False   contained /usr/share/ansible/collections/
ansible_collections/amazon/aws
1|ansible.builtin 2.13.3 False   contained /usr/lib/python3.9/site-packages/
ansible
...output omitted...
17|gls.utils       0.0.1  False   bind_mount /home/student/role-collections/
collections/ansible_collections/gls/utils
...output omitted...
```

- 1.4. Select the `gls.utils` collection to list the roles that the `gls.utils` Ansible Content Collection provides.

```
Gls.utils  Type     Added Deprecated Description
0|backup    role    Unknown    Unknown    Curriculum Developer
1|myping    module   historical False      Try to connect to host, verify a
usable python and return C(pong) on success
2|restore   role    Unknown    Unknown    Curriculum Developer
```

In the preceding output, notice that the collection provides two roles: `backup` and `restore`.

- 1.5. Select the `backup` role to read its documentation.

```
Image: gls.utils.backup
Description: Curriculum Developer
...output omitted...
24| backup
25| =====
26|
27| This role backups up the files and directories provided using the
`backup_files` variable.
28| The backup is identified with a given name (`backup_id`) and can be restored
using the `gls.utils.restore` role.
29|
30| If a backup with the same name already exists, then the role immediately
returns.
31|
32|
33| Requirements
34| -----
35|
36| None
37|
38|
39| Role Variables
40| -----
```

```
41|
42| The role accepts the following variables:
...output omitted...
```

- 1.6. Press Esc to return to the preceding screen. The modules provided by the role are also indicated by `ansible-navigator`.

Gls.utils	Type	Added	Deprecated	Description
0 <code>backup</code>	role	Unknown	Unknown	Curriculum Developer
1 <code>myping</code>	module	historical	False	Try to connect to host, verify a usable python and return C(pong) on success
2 <code>restore</code>	role	Unknown	Unknown	Curriculum Developer

The `gls.utils` collection provides the `newping` module.

- 1.7. Type :q to exit the `ansible-navigator collections` TUI, then use the `ansible-navigator doc` command to display the documentation for the `gls.utils.newping` module.

```
[student@workstation role-collections]$ ansible-navigator doc \
> -m stdout gls.utils.newping
> GLS.UTILS.MYPING      (/home/student/role-collections/collections/
ansible_collections/gls/uti>
```

A trivial test module, this module always returns 'pong' on successful contact. It does not make sense in playbooks, but it is useful from `/usr/bin/ansible` to verify the ability to login and that a usable Python is configured. This is NOT ICMP ping, this is just a trivial test module that requires Python on the remote-node. For Windows targets, use the `[ansible.windows.win_ping]` module instead. For Network targets, use the `[ansible.netcommon.net_ping]` module instead.

...output omitted...

Type :q to exit `ansible-navigator doc`.

- 2. Complete and then run the `/home/student/role-collections/bck.yml` playbook. That playbook uses the `gls.utils.newping` module and the `gls.utils.backup` role.

- 2.1. Edit the `bck.yml` playbook. In the first task, run the `gls.utils.newping` module.

```
...output omitted...
tasks:
  - name: Ensure the machine is up
    gls.utils.newping:
      data: pong
...output omitted...
```

Do not close the file yet.

- 2.2. In the second task, run the `gls.utils.backup` role. When you finish editing the file, save and close it.

```
...output omitted...
- name: Ensure configuration files are saved
  include_role:
    name: gls.utils.backup
  vars:
    backup_id: backup_etc
    backup_files:
      - /etc/sysconfig
      - /etc/yum.repos.d
```

The resulting file must contain the following contents:

```
---
- name: Backup the system configuration
  hosts: servera.lab.example.com
  become: true
  gather_facts: false

  tasks:
    - name: Ensure the machine is up
      gls.utils.newping:
        data: pong

    - name: Ensure configuration files are saved
      include_role:
        name: gls.utils.backup
      vars:
        backup_id: backup_etc
        backup_files:
          - /etc/sysconfig
          - /etc/yum.repos.d
```

2.3. Verify the syntax of the `bck.yml` playbook. If you get any errors, compare your playbook to the preceding example.

```
[student@workstation role-collections]$ ansible-navigator run \
> -m stdout bck.yml --syntax-check
playbook: /home/student/role-collections/bck.yml
```

2.4. Run the playbook.

```
[student@workstation role-collections]$ ansible-navigator run \
> -m stdout bck.yml

PLAY [Backup the system configuration] ****
TASK [Ensure the machine is up] ****
ok: [servera.lab.example.com]

TASK [Ensure configuration files are saved] ****
TASK [gls.utils.backup : Ensure the backup directory exists] ****
```

```

changed: [servera.lab.example.com]

TASK [gls.utils.backup : Ensure the backup exists] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=3    changed=2    unreachable=0    failed=0
skipped=0      rescued=0    ignored=0

```

- ▶ 3. In the second part of this exercise, install the Ansible Content Collections specified by the Ansible project's `requirements.yml` file.

To test your work, run the `new_system.yml` playbook. That playbook uses the `redhat.insights.insights_client` and `redhat.rhel_system_roles.selinux` roles to configure Red Hat Insights and SELinux on the `servera` machine.

- 3.1. Review the `requirements.yml` file. The file lists three Ansible Content Collections to install from tar archive files that are located in your Ansible project directory. The `redhat.insights` collection currently requires the unsupported `community.general.ini_file` module.

The tar archive files are pre-downloaded to your project directory from the Red Hat automation hub at <https://console.redhat.com/ansible/automation-hub>.

```

---
collections:
  - name: /home/student/role-collections/redhat-insights-1.0.7.tar.gz
  - name: /home/student/role-collections/redhat-rhel_system_roles-1.19.3.tar.gz
  - name: /home/student/role-collections/community-general-5.5.0.tar.gz

```

- 3.2. Use the `ansible-galaxy` command with the `requirements.yml` file to install the collections.

```

[student@workstation role-collections]$ ansible-galaxy collection install \
> -r requirements.yml -p collections
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Installing 'redhat.insights:1.0.7' to '/home/student/role-collections/collections/
ansible_collections/redhat/insights'
redhat.insights:1.0.7 was installed successfully
Installing 'redhat.rhel_system_roles:1.19.3' to '/home/student/role-collections/
collections/ansible_collections/redhat/rhel_system_roles'
redhat.rhel_system_roles:1.19.3 was installed successfully
Installing 'community.general:5.5.0' to '/home/student/role-collections/
collections/ansible_collections/community/general'
community.general:5.5.0 was installed successfully

```

- 3.3. Use the `ansible-galaxy collection list` command to verify that the collections are installed.

```

[student@workstation role-collections]$ ansible-galaxy collection list
...output omitted...

```

```
# /home/student/role-collections/collections/ansible_collections
Collection          Version
-----
community.general   5.5.0
gls.utils           0.0.1
redhat.insights    1.0.7
redhat.rhel_system_roles 1.19.3
```

There may be other collections installed under `/usr/share/ansible/collections/ansible_collections`.

- 3.4. Use the `ansible-navigator collections` command to view the `insights_client` role contained in the `redhat.insights` collection.

When you run the `ansible-navigator collections` command, it displays the following output in its TUI:

```
[student@workstation role-collections]$ ansible-navigator collections

  Name          Version Shadowed Type      Path
0|amazon.aws    3.2.0  False   contained  /usr/share/ansible/collections/
ansible_collections/amazon/aws
1|ansible.builtin 2.13.3 False   contained  /usr/lib/python3.9/site-packages/
ansible
...output omitted...
23|redhat.insights  1.0.7 False   bind_mount /home/student/role-collections/
collections/ansible_collections/redhat/insights
...output omitted...
```

Enter :23 to select the `redhat.insights` Ansible Content Collection. The following output is displayed by `ansible-navigator` in its TUI:

	Name	Type	Added	Deprecated	Description
0 compliance	Redhat.insights	role	Unknown	Unknown	Install and configure Red Hat Insights Client
1 insights	Red Hat Insights Client	inventory	None	False	insights inventory source
2 insights_client	Red Hat Insights Client	role	Unknown	Unknown	Install and configure
3 insights_config		module	None	False	This module handles initial configuration of the insights client on install
4 insights_register		module	None	False	This module registers the insights client

Enter :2 to select the documentation for the `insights_client` role from the Ansible Content Collection. The `ansible-navigator` command displays the following output in its TUI:

```
Image: redhat.insights.insights_client
Description: Install and configure Red Hat Insights Client
0|---
1|argument_specs: {}
2|argument_specs_path: ''
3|defaults: {}
```

```

4|defaults_path: ''
5|full_name: redhat.insights.insights_client
6|info:
7|  galaxy_info:
8|    author: Red Hat, Inc
9|    categories:
10|      - packaging
11|      - system
12|    company: Red Hat, Inc.
13|    dependencies: []
14|    description: Install and configure Red Hat Insights Client
...output omitted...

```

Type :q to exit the ansible-navigator TUI.

- 3.5. Review the `new_system.yml` playbook. This playbook uses roles from the `redhat.insights` and `redhat.rhel_system_roles` collections.

```

---
- name: Configure the system
  hosts: servera.lab.example.com
  become: true
  gather_facts: true

  tasks:
    - name: Ensure the system is registered with Insights
      include_role:
        name: redhat.insights.insights_client
      vars:
        auto_config: false
        insights_proxy: http://proxy.example.com:8080

    - name: Ensure SELinux mode is Enforcing
      include_role:
        name: redhat.rhel_system_roles.selinux
      vars:
        selinux_state: enforcing

```

- 3.6. Run the `new_system.yml` playbook in check mode to confirm that you correctly installed the required collections.

```

[student@workstation role-collections]$ ansible-navigator run \
> -m stdout new_system.yml --check

...output omitted...

RUNNING HANDLER [redhat.insights.insights_client : Run insights-client] ****
skipping: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=17    changed=3    unreachable=0    failed=0
                               skipped=18   rescued=0    ignored=0

```



Important

Because the classroom systems are not registered with Red Hat Subscription Management and might not have internet access, the `new_system.yml` playbook cannot complete successfully. However, to confirm that you correctly installed the required collections, you can still run the playbook in check mode.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish role-collections
```

This concludes the section.

Reusing Content with System Roles

Objectives

- Write playbooks that take advantage of system roles for Red Hat Enterprise Linux to perform standard operations.

System Roles

System roles are a set of Ansible roles that you can use to configure and manage various components, subsystems, and software packages included with Red Hat Enterprise Linux. System roles provide automation for many common system configuration tasks, including time synchronization, networking, firewall, tuning, and logging.

These roles are intended to provide an automation API that is consistent across multiple major and minor releases of Red Hat Enterprise Linux. The Knowledgebase article at <https://access.redhat.com/articles/3050101> documents the versions of Red Hat Enterprise Linux on your managed hosts that specific roles support.

Simplified Configuration Management

System roles can help you simplify automation across multiple versions of Red Hat Enterprise Linux. For example, the recommended time synchronization service is different in different versions of Red Hat Enterprise Linux.

- The `chronyd` service is preferred in RHEL 9.
- The `ntpd` service is preferred in RHEL 6.

In an environment with a mixture of Red Hat Enterprise Linux versions, you must manage the configuration files for both services.

You can use an Ansible Playbook that runs the `redhat.rhel_system_roles.timesync` role to configure time synchronization for managed hosts running either version of Red Hat Enterprise Linux.

By using system roles, you no longer need to maintain configuration files for both services.

Support for System Roles

System roles are provided as the Red Hat Certified Ansible Content Collection `redhat.rhel_system_roles` through automation hub for Red Hat Ansible Automation Platform customers, as part of their subscription.

In addition, the roles are provided as an RPM package (`rhel-system-roles`) with Red Hat Enterprise Linux 9 for use with the version of Ansible Core provided by that operating system. These system roles have the same lifecycle support benefits that come with a Red Hat Enterprise Linux subscription.

Most system roles are "Fully Supported" and have stable interfaces. For any "Fully Supported" system role, Red Hat endeavors to ensure that the names of role variables and how they work are unchanged in future versions. Therefore, playbook refactoring due to system role updates should be minimal.

Some system roles are in "Technology Preview". These are tested and are stable, but might be subject to future changes that are incompatible with the current state of the role. Integration testing is recommended for playbooks that incorporate any "Technology Preview" role. Playbooks might require refactoring if role variables change in a future version of the role.

You can access documentation for the system roles in `ansible-navigator`, or by reviewing the documentation on the Red Hat Customer Portal at *Administration and configuration tasks using System Roles in RHEL* [https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html-single/administration_and_configuration_tasks_using_system_roles_in_rhel/index].

Installing the System Roles Ansible Content Collection

You can install the `redhat.rhel_system_roles` Ansible Content Collection with the `ansible-galaxy collection install` command. An Ansible project can specify this dependency by creating a `collections/requirements.yml` file. The following example assumes that your project is set up to pull Ansible Content Collections from automation hub.

```
---
```

```
collections:
  - name: redhat.rhel_system_roles
```

The following command installs any required collections for a given project into the project's `collections/` directory.

```
[user@demo demo-project]$ ansible-galaxy collection \
> install -p collections/ -r collections/requirements.yml
...output omitted...
```



Note

Upstream development is done through the Linux System Roles project; their development versions of the system roles are provided through Ansible Galaxy as the `fedora.linux_system_roles` Ansible Content Collection. These roles are not supported by Red Hat.

Example: Time Synchronization Role

Suppose you need to configure NTP time synchronization on your web servers. You could write automation yourself to perform each of the necessary tasks. But the system roles collection includes a role that can perform the configuration, `redhat.rhel_system_roles.timesync`.

The documentation for this role describes all the variables that affect the role's behavior and provides three playbook snippets that illustrate different time synchronization configurations.

To manually configure NTP servers, the role has a variable named `timesync_ntp_servers`. This variable defines a list of NTP servers to use. Each item in the list is made up of one or more attributes. The two key attributes are `hostname` and `iburst`.

Attribute	Purpose
<code>hostname</code>	The hostname of an NTP server with which to synchronize.

Attribute	Purpose
<code>iburst</code>	A Boolean that enables or disables fast initial synchronization. Defaults to <code>no</code> in the role, but you should normally set this to <code>yes</code> .

Given this information, the following example is a play that uses the `redhat.rhel_system_roles.timesync` role to configure managed hosts to get time from three NTP servers by using fast initial synchronization.

```
- name: Time Synchronization Play
hosts: webservers
vars:
  timesync_ntp_servers:
    - hostname: 0.rhel.pool.ntp.org
      iburst: yes
    - hostname: 1.rhel.pool.ntp.org
      iburst: yes
    - hostname: 2.rhel.pool.ntp.org
      iburst: yes

  roles:
    - redhat.rhel_system_roles.timesync
```

This example sets the role variables in a `vars` section of the play, but a better practice might be to configure them as inventory variables for hosts or host groups.

Consider an example project with the following structure:

```
[user@demo demo-project]$ tree
.
├── ansible.cfg
├── group_vars
│   └── webservers
│       └── timesync.yml①
└── inventory
└── timesync_playbook.yml②
```

- ① Defines the time synchronization variables overriding the role defaults for hosts in group `webservers` in the inventory. This file would look something like:

```
timesync_ntp_servers:
  - hostname: 0.rhel.pool.ntp.org
    iburst: yes
  - hostname: 1.rhel.pool.ntp.org
    iburst: yes
  - hostname: 2.rhel.pool.ntp.org
    iburst: yes
```

- ② The content of the playbook simplifies to:

```
- name: Time Synchronization Play
hosts: webservers
roles:
  - redhat.rhel_system_roles.timesync
```

This structure cleanly separates the role, the playbook code, and configuration settings. The playbook code is simple, easy to read, and should not require complex refactoring. The role content is maintained and supported by Red Hat. All the settings are handled as inventory variables.

This structure also supports a dynamic, heterogeneous environment. Hosts with new time synchronization requirements might be placed in a new host group. Appropriate variables are defined in a YAML file and placed in the appropriate `group_vars` (or `host_vars`) subdirectory.

Example: SELinux Role

As another example, the `redhat.rhel_system_roles.selinux` role simplifies management of SELinux configuration settings. This role is implemented using the SELinux-related Ansible modules. The advantage of using this role instead of writing your own tasks is that it relieves you from the responsibility of writing those tasks. Instead, you provide variables to the role to configure it, and the maintained code in the role ensures your desired SELinux configuration is applied.

This role can perform the following tasks:

- Set enforcing or permissive mode
- Run `restorecon` on parts of the file system hierarchy
- Set SELinux Boolean values
- Set SELinux file contexts persistently
- Set SELinux user mappings

Calling the SELinux Role

Sometimes, the SELinux role must ensure that the managed hosts are rebooted in order to completely apply its changes. However, the role does not ever reboot hosts itself, enabling you to control how the reboot is handled. Therefore, it is a little more complicated than usual to properly use this role in a play.

The role sets a Boolean variable, `selinux_reboot_required`, to `true` and fails if a reboot is needed. You can use a `block/rescue` structure to recover from the failure by failing the play if that variable is not set to `true`, or rebooting the managed host and rerunning the role if it is `true`. The block in your play should look something like this:

```
- name: Apply SELinux role
block:
  - include_role:
      name: redhat.rhel_system_roles.selinux
rescue:
  - name: Check for failure for other reasons than required reboot
    ansible.builtin.fail:
      when: not selinux_reboot_required

  - name: Restart managed host
    ansible.builtin.reboot:
```

```
- name: Reapply SELinux role to complete changes
  include_role:
    name: redhat.rhel_system_roles.selinux
```

Configuring the SELinux Role

The variables used to configure the `redhat.rhel_system_roles.selinux` role are described in the role's documentation. The following examples show some ways to use this role.

The `selinux_state` variable sets the mode that SELinux runs in. It can be set to `enforcing`, `permissive`, or `disabled`. If it is not set, the mode is not changed.

```
selinux_state: enforcing
```

The `selinux_booleans` variable takes a list of SELinux Boolean values to adjust. Each item in the list is a dictionary of variables: the name of the Boolean, the `state` (whether it should be on or off), and whether the setting should be `persistent` across reboots.

This example sets `httpd_enable_homedirs` to `on` persistently:

```
selinux_booleans:
- name: 'httpd_enable_homedirs'
  state: 'on'
  persistent: 'yes'
```

The `selinux_fcontexts` variable takes a list of file contexts to persistently set (or remove), and works much like the `selinux_fcontext` command.

The following example ensures the policy has a rule to set the default SELinux type for all files under `/srv/www` to `httpd_sys_content_t`.

```
selinux_fcontexts:
- target: '/srv/www(/.*)?'
  setype: 'httpd_sys_content_t'
  state: 'present'
```

The `selinux_restore_dirs` variable specifies a list of directories on which to run the `restorecon` command:

```
selinux_restore_dirs:
- /srv/www
```

The `selinux_ports` variable takes a list of ports that should have a specific SELinux type.

```
selinux_ports:
- ports: '82'
  setype: 'http_port_t'
  proto: 'tcp'
  state: 'present'
```

There are other variables and options for this role. See the role documentation for more information.

Using System Roles with Ansible Core Only



Note

This section is relevant if you plan to use system roles without a Red Hat Ansible Automation Platform subscription, by using the version of Ansible Core that is provided with Red Hat Enterprise Linux 9.

If you do not have a subscription to Red Hat Ansible Automation Platform, but you do have Red Hat Enterprise Linux systems, you can use system roles with the version of Ansible Core provided with RHEL.

That version of Ansible Core is only supported for use with system roles and other automation code provided by Red Hat. In addition, it does not include `ansible-navigator`, so you have to use tools like `ansible-playbook` that treat your control node as the execution environment to run your automation.

Because `ansible-playbook` does not use execution environments, the only Ansible Content Collection that you have available by default is `ansible.builtin` collection. Other packages included with Red Hat Enterprise Linux might add additional Ansible Content Collections to the control node.

Installing the System Roles RPM Package

Make sure that your control node is registered with Red Hat Subscription Manager and has a Red Hat Enterprise Linux subscription. You should also install the `ansible-core` RPM package.

To install the `rhel-system-roles` RPM package, make sure that the AppStream package repository is enabled. For Red Hat Enterprise Linux 9 on the x86_64 processor architecture, this is the `rhel-9-for-x86_64-appstream-rpms` repository. Then, you can install the package.

```
[user@controlnode ~]$ sudo dnf install rhel-system-roles
```



Important

If you use Ansible Core from a basic Red Hat Enterprise Linux installation for your control node, and do not have a Red Hat Ansible Automation Platform subscription on that node, then your control node should be a fully updated installation of the most recent version of Red Hat Enterprise Linux. You should also use the most recent version of the `ansible-core` and `rhel-system-roles` packages.

After installation, the collection is installed in the `/usr/share/ansible/collections/ansible_collections/redhat/rhel_system_roles` directory on your control node. The individual roles are also installed in the `/usr/share/ansible/roles` directory for backward compatibility:

```
[user@controlnode ~]$ ls -1F /usr/share/ansible/roles/
linux-system-roles.certificate@
linux-system-roles.cockpit@
linux-system-roles.crypto_policies@
linux-system-roles.firewall@
linux-system-roles.ha_cluster@
```

```

linux-system-roles.kdump@
linux-system-roles.kernel_settings@
linux-system-roles.logging@
linux-system-roles.metrics@
linux-system-roles.nbde_client@
linux-system-roles.nbde_server@
linux-system-roles.network@
linux-system-roles.postfix@
linux-system-roles.selinux@
linux-system-roles.ssh@
linux-system-roles.sshd@
linux-system-roles.storage@
linux-system-roles.timesync@
linux-system-roles.tlog@
linux-system-roles.vpn@
rhel-system-roles.certificate/
rhel-system-roles.cockpit/
rhel-system-roles.crypto_policies/
rhel-system-roles.firewall/
rhel-system-roles.ha_cluster/
rhel-system-roles.kdump/
rhel-system-roles.kernel_settings/
rhel-system-roles.logging/
rhel-system-roles.metrics/
rhel-system-roles.nbde_client/
rhel-system-roles.nbde_server/
rhel-system-roles.network/
rhel-system-roles.postfix/
rhel-system-roles.selinux/
rhel-system-roles.ssh/
rhel-system-roles.sshd/
rhel-system-roles.storage/
rhel-system-roles.timesync/
rhel-system-roles.tlog/
rhel-system-roles.vpn/

```

The corresponding upstream name of each role is linked to the system role. This enables individual roles to be referenced in a playbook by either name.



Important

If you are using `ansible-playbook` to run your playbook, and your playbook refers to a system role that was installed using the RPM package's FQCN, you must use the `redhat.rhel_system_roles` version of its name. For example, you could refer to the `firewall` role as:

- `redhat.rhel_system_roles.firewall` (its FQCN in the collection)
- `rhel-system-roles.firewall` (its name as an independent role)
- `linux-system-roles.firewall` (its name as the upstream independent role)

You cannot use `fedora.linux_system_roles.firewall` because the `fedora.linux_system_roles` collection is not installed on the system.

In addition, the independent role names only work if `/usr/share/ansible/roles` is in your `roles_path` setting.

Accessing Documentation for System Roles

If you are working with system roles in Red Hat Enterprise Linux and do not have `ansible-navigator` available to you, there are other ways to get documentation about system roles.

The official documentation for system roles is located at *Administration and configuration tasks using System Roles in RHEL* [https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html-single/administration_and_configuration_tasks_using_system_roles_in_rhel/index].

However, if you installed the system roles from the RPM package, documentation is also available under the `/usr/share/doc/rhel-system-roles/` directory.

```
[user@controlnode ~]$ ls -1 /usr/share/doc/rhel-system-roles/
certificate
cockpit
collection
crypto_policies
firewall
ha_cluster
kdump
kernel_settings
logging
metrics
nbde_client
nbde_server
network
postfix
selinux
ssh
sshd
storage
timesync
tlog
vpn
```

Each role's documentation directory contains a `README.md` file. The `README.md` file contains a description of the role, along with information on how to use it.

The `README.md` file also describes role variables that affect the behavior of the role. Often the `README.md` file contains a playbook snippet that demonstrates variable settings for a common configuration scenario.

Some role documentation directories contain example playbooks. When using a role for the first time, review any additional example playbooks in the documentation directory.

Running Playbooks Without Automation Content Navigator

You can use the `ansible-playbook` command to run a playbook that uses system roles in Red Hat Enterprise Linux when you do not have Red Hat Ansible Automation Platform or `ansible-navigator``.

The syntax of `ansible-playbook` is very similar to `ansible-navigator run -m stdout`, and takes many of the same options.

```
[user@controlnode ~]$ ansible-playbook playbook.yml
```



References

Red Hat Enterprise Linux (RHEL) System Roles

<https://access.redhat.com/articles/3050101>

Red Hat Enterprise Linux System Roles Ansible Collection

https://console.redhat.com/ansible/automation-hub/repo/published/redhat/rhel_system_roles

Scope of support for the Ansible Core package included in the RHEL 9 and RHEL 8.6 and later AppStream repositories

<https://access.redhat.com/articles/6325611>

Using Ansible in RHEL 9

<https://access.redhat.com/articles/6393321>

Linux System Roles

<https://linux-system-roles.github.io/>

Linux System Roles Ansible Collection

https://galaxy.ansible.com/fedora/linux_system_roles

► Guided Exercise

Reusing Content with System Roles

In this exercise, you use one of the system roles in conjunction with tasks to configure time synchronization and the time zone on your managed hosts.

Outcomes

- Install the system roles for Red Hat Enterprise Linux.
- Find and use the system roles documentation.
- Use the `redhat.rhel_system_roles.timesync` role in a playbook to configure time synchronization on remote hosts.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start role-system
```

Instructions

- 1. Change into the `/home/student/role-system` directory.

```
[student@workstation ~]$ cd ~/role-system  
[student@workstation role-system]$
```

- 2. Install the system roles on your control node, `workstation.lab.example.com`. Confirm that the system roles have been installed by using `ansible-galaxy`.

- 2.1. Use the `ansible-galaxy collection list` command to list the installed collections.

```
[student@workstation role-system]$ ansible-galaxy collection list
```

- 2.2. Create the `collections` directory.

```
[student@workstation role-system]$ mkdir -p collections
```

- 2.3. The `./collections` directory is not a default search location for collections. Add the `collections_paths` key to the `ansible.cfg` file, so that the `./collections` directory is searched.

```
[defaults]
inventory=~/inventory
remote_user=devops
collections_paths=~/collections:~/ansible/collections:/usr/share/ansible/collections
```

- 2.4. Use the `ansible-galaxy command` to install the `redhat.rhel_system_roles` collection from the provided tarball.

```
[student@workstation role-system]$ ansible-galaxy collection \
> install -p collections/ redhat-rhel_system_roles-1.19.3.tar.gz
Process install dependency map
Starting collection install process
Installing 'redhat.rhel_system_roles:1.19.3' to '/home/student/role-system/
collections/ansible_collections/redhat/rhel_system_roles'
redhat.rhel_system_roles:1.19.3 was installed successfully
```

- 2.5. Use the `ansible-galaxy collection list` command to verify that the system roles are now available.

```
[student@workstation role-system]$ ansible-galaxy collection list

# /home/student/role-system/collections/ansible_collections
Collection          Version
-----
redhat.rhel_system_roles 1.19.3
```

- ▶ 3. Create the `configure_time.yml` playbook with one play that targets the `database_servers` host group and runs the `redhat.rhel_system_roles.timesync` role in its `roles` section.

```
---
- name: Time Synchronization
  hosts: database_servers

  roles:
    - redhat.rhel_system_roles.timesync
```

- ▶ 4. The role documentation contains a description of each role variable, including the default value for the variable. Determine the role variables that you must override to meet the requirements for time synchronization.

Place role variable values in a file named `timesync.yml`. Because these variable values apply to all hosts in the inventory, place the `timesync.yml` file in the `group_vars/all` subdirectory.

- 4.1. Review the `Role Variables` section of the `README.md` file for the `redhat.rhel_system_roles.timesync` role.

```
[student@workstation role-system]$ cat \
> collections/ansible_collections/redhat/rhel_system_roles/roles/timesync/README.md
...output omitted...
Role Variables
```

```
-----
...output omitted...
# List of NTP servers
timesync_ntp_servers:
  - hostname: foo.example.com      # Hostname or address of the server
    minpoll: 4                      # Minimum polling interval (default 6)
    maxpoll: 8                      # Maximum polling interval (default 10)
    iburst: yes                     # Flag enabling fast initial synchronization
                                    # (default no)
    pool: no                        # Flag indicating that each resolved address
                                    # of the hostname is a separate NTP server
                                    # (default no)
...output omitted...
# Name of the package which should be installed and configured for NTP.
# Possible values are "chrony" and "ntp". If not defined, the currently active
# or enabled service will be configured. If no service is active or enabled, a
# package specific to the system and its version will be selected.
timesync_ntp_provider: chrony
...output omitted...
```

4.2. Create the group_vars/all subdirectory.

```
[student@workstation role-system]$ mkdir -pv group_vars/all
mkdir: created directory 'group_vars'
mkdir: created directory 'group_vars/all'
```

4.3. Create a group_vars/all/timesync.yml file, adding variable definitions to satisfy the time synchronization requirements. The file now contains:

```
---
#redhat.rhel_system_roles.timesync variables for all hosts

timesync_ntp_provider: chrony

timesync_ntp_servers:
  - hostname: classroom.example.com
    iburst: yes
```

- ▶ 5. Add two tasks to the configure_time.yml file to get and conditionally set the time zone for each host. Ensure that both tasks run after the redhat.rhel_system_roles.timesync role.

Because hosts do not belong to the same time zone, use a variable (`host_timezone`) for the time zone name.

5.1. Create a post_tasks section in the configure_time.yml playbook, then add the first task.

```
post_tasks:
  - name: Get time zone
    ansible.builtin.command: timedatectl show
    register: current_timezone
    changed_when: false
```

- 5.2. Add a second task to set the time zone, but only when the time zone is incorrect. Because system logging and other services use the system time zone, reboot each host when the time zone is modified. Add a `notify` keyword to the task, with an associated value of `reboot host`. The `post_tasks` section of the play should now read:

```
- name: Set time zone
ansible.builtin.command: "timedatectl set-timezone {{ host_timezone }}"
when: host_timezone not in current_timezone.stdout
notify: reboot host
```

- 5.3. Add the `reboot host` handler to the `Time Synchronization` play. The complete playbook now contains:

```
---
- name: Time Synchronization
hosts: database_servers

roles:
- redhat.rhel_system_roles.timesync

post_tasks:
- name: Get time zone
  ansible.builtin.command: timedatectl show
  register: current_timezone
  changed_when: false

- name: Set time zone
  ansible.builtin.command: "timedatectl set-timezone {{ host_timezone }}"
  when: host_timezone not in current_timezone.stdout
  notify: reboot host

handlers:
- name: reboot host
  ansible.builtin.reboot:
```

- 6. For each data center, create a file named `timezone.yml` that contains an appropriate value for the `host_timezone` variable. Use the `timedatectl list-timezones` command to find the valid time zone string for each data center.

- 6.1. Create the `group_vars` subdirectories for the `na_datacenter` and `europe_datacenter` host groups.

```
[student@workstation role-system]$ mkdir -pv \
> group_vars/{na_datacenter,europe_datacenter}
mkdir: created directory 'group_vars/na_datacenter'
mkdir: created directory 'group_vars/europe_datacenter'
```

- 6.2. Use the `timedatectl list-timezones` command to determine the time zone for both the US and European data centers:

```
[student@workstation role-system]$ timedatectl list-timezones | grep Chicago
America/Chicago
[student@workstation role-system]$ timedatectl list-timezones | grep Helsinki
Europe/Helsinki
```

6.3. Create the `timezone.yml` for both data centers:

```
[student@workstation role-system]$ echo "host_timezone: America/Chicago" > \
> group_vars/na_datacenter/timezone.yml
[student@workstation role-system]$ echo "host_timezone: Europe/Helsinki" > \
> group_vars/europe_datacenter/timezone.yml
```

▶ 7. Run the `configure_time.yml` playbook.

7.1. Use the `ansible-navigator run --syntax-check` command to validate the playbook syntax.

```
[student@workstation role-system]$ ansible-navigator run \
> -m stdout configure_time.yml --syntax-check
playbook: /home/student/role-system/configure_time.yml
```

7.2. Run the `configure_time.yml` playbook.

```
[student@workstation role-system]$ ansible-navigator run \
> -m stdout configure_time.yml

PLAY [Time Synchronization] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.timesync : Set version specific variables] ****
...output omitted...

TASK [redhat.rhel_system_roles.timesync : Enable timemaster] ****
skipping: [servera.lab.example.com]
skipping: [serverb.lab.example.com]

RUNNING HANDLER [redhat.rhel_system_roles.timesync : restart chronyrd] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Get time zone] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Set time zone] ****
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]

RUNNING HANDLER [reboot host] ****
```

```
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=18    changed=6     unreachable=0    failed=0
skipped=25    rescued=0    ignored=0
serverb.lab.example.com      : ok=18    changed=6     unreachable=0    failed=0
skipped=25    rescued=0    ignored=0
```

- ▶ 8. Verify the time zone settings of each server. Use the following commands to see the output of the `date` command on the `servera` and `serverb` machines.



Note

The actual time zones listed might vary depending on the time of year, and whether daylight savings is active.

```
[student@workstation role-system]$ ssh servera date
Tue Aug 16 07:43:33 PM CDT 2022
[student@workstation role-system]$ ssh serverb date
Wed Aug 17 03:43:41 AM EEST 2022
```

Each server has a time zone setting based on its geographic location.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish role-system
```

This concludes the section.

▶ Lab

Simplifying Playbooks with Roles and Ansible Content Collections

In this lab, you create Ansible roles that use variables, files, templates, tasks, and handlers.

Outcomes

- Create Ansible roles that use variables, files, templates, tasks, and handlers to configure a development web server.
- In a playbook, use a role that is hosted in a remote repository.
- Use a system role for Red Hat Enterprise Linux in a playbook.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start role-review
```

Instructions

Your organization must provide a single web server to host development code for all its web developers. You are tasked with writing a playbook to configure this development web server.

The development web server must satisfy several requirements:

- The development server configuration matches the production server configuration. The production server is configured by using an Ansible role, developed by the organization's infrastructure team.
- Each developer is given a directory on the development server to host code and content. Each developer's content is accessed by using an assigned, nonstandard port.
- SELinux is set to enforcing mode and is using the targeted policy.

Your playbook must:

- Use a role to configure directories and ports for each developer on the web server. You must write this role.

This role has a dependency on a role to configure Apache that was written by your organization. You should define the dependency, specifying version v1.4 of the role. The URL of the dependency's Git repository is: `git@workstation.lab.example.com:infra/apache`

- Use the `rhel-system-roles.selinux` role to configure SELinux to allow external hosts to access the nonstandard HTTP ports used by your web server. You are provided with an

`selinux.yml` variable file that you can use in the `group_vars` directory to pass the correct settings to the role.

1. Change into the `/home/student/role-review` directory.
2. Use the `ansible-galaxy collection install` command to install the `redhat.rhel_system_roles` collection from the provided `.tar` archive file, then verify that it was installed correctly.
3. Create a playbook named `web_dev_server.yml` with a single play named `Configure Dev Web Server`. Configure the play to target the `dev_webserver` host group. Do not add any roles or tasks to the play yet.

Ensure that the play forces handlers to execute, because you might encounter an error when developing the playbook.
4. Verify the syntax of the playbook, then run the playbook. The syntax check should pass and the playbook should run successfully. The play only gathers facts because it has no roles or tasks yet.
5. Make sure to install any roles that are dependencies of roles in your playbook.

For example, the `apache.developer_configs` role that you write later in this lab depends on the `infra.apache` role.

Create a `roles/requirements.yml` file. This file installs the role from the Git repository at `git@workstation.lab.example.com:infra/apache`, use version `v1.4`, and name it `infra.apache` locally.

You can assume that your SSH keys are configured to allow you to get roles from this repository automatically. Install the role with the `ansible-galaxy` command.
6. Initialize a new role named `apache.developer_configs` in the `roles` subdirectory.

Add the `infra.apache` role as a dependency for the new role, using the same information for name, source, version, and version control system as the `roles/requirements.yml` file.

The `developer_tasks.yml` file in the project directory contains tasks for the role. Move this file to the correct location in the role directory hierarchy for a tasks file that is used by this role, replacing the existing file in that location.

The `developer.conf.j2` file in the project directory is a Jinja2 template that is used by the tasks file. Move this file to the correct location for template files that are used by this role.
7. The `apache.developer_configs` role creates a user account and configures a web server instance for the list of users defined in the variable named `web_developers`.

The `web_developers.yml` file in the project directory defines the `web_developers` variable, which is the list of users for the role.

Review this file and put it in the correct location to define the `web_developers` variable for the development web server host group from your inventory.
8. Add the `apache.developer_configs` role to the play in the `web_dev_server.yml` playbook.
9. Verify the syntax of the playbook, and then run the playbook. The syntax check should pass, but the playbook should fail when the `infra.apache` role attempts to restart Apache HTTPD.
10. Apache HTTPD failed to restart in the preceding step because the network ports it uses for your developers are labeled with the wrong SELinux contexts.

You can use the provided variable file, `selinux.yml`, with the `rhel-system-roles.selinux` role to fix the issue.

Create a `pre_tasks` section for your play in the `web_dev_server.yml` playbook. In that section, use a task to include the `rhel-system-roles.selinux` role in a `block/rescue` structure, so that it is properly applied.

Move the `selinux.yml` file to the correct location so that its variables are set for the `dev_webserver` host group.

11. Verify the final `web_dev_server.yml` playbook and run a syntax check. The syntax check should pass.
Validate that the `web_dev_server.yml` playbook passes a syntax check.
12. Run the `web_dev_server.yml` playbook. It should succeed.
13. Test the configuration of the development web server. Verify that all endpoints are accessible and serving each developer's content.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade role-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish role-review
```

This concludes the section.

► Solution

Simplifying Playbooks with Roles and Ansible Content Collections

In this lab, you create Ansible roles that use variables, files, templates, tasks, and handlers.

Outcomes

- Create Ansible roles that use variables, files, templates, tasks, and handlers to configure a development web server.
- In a playbook, use a role that is hosted in a remote repository.
- Use a system role for Red Hat Enterprise Linux in a playbook.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start role-review
```

Instructions

Your organization must provide a single web server to host development code for all its web developers. You are tasked with writing a playbook to configure this development web server.

The development web server must satisfy several requirements:

- The development server configuration matches the production server configuration. The production server is configured by using an Ansible role, developed by the organization's infrastructure team.
- Each developer is given a directory on the development server to host code and content. Each developer's content is accessed by using an assigned, nonstandard port.
- SELinux is set to enforcing mode and is using the targeted policy.

Your playbook must:

- Use a role to configure directories and ports for each developer on the web server. You must write this role.

This role has a dependency on a role to configure Apache that was written by your organization. You should define the dependency, specifying version v1.4 of the role. The URL of the dependency's Git repository is: `git@workstation.lab.example.com:infra/apache`

- Use the `rhel-system-roles.selinux` role to configure SELinux to allow external hosts to access the nonstandard HTTP ports used by your web server. You are provided with an

`selinux.yml` variable file that you can use in the `group_vars` directory to pass the correct settings to the role.

1. Change into the `/home/student/role-review` directory.

```
[student@workstation ~]$ cd ~/role-review  
[student@workstation role-review]$
```

2. Use the `ansible-galaxy collection install` command to install the `redhat.rhel_system_roles` collection from the provided tar archive file, then verify that it was installed correctly.

- 2.1. Use the `ansible-galaxy collection install` command to install the `redhat.rhel_system_roles` collection from the provided tar archive file.

```
[student@workstation role-review]$ ansible-galaxy collection \  
> install -p collections/ redhat-rhel_system_roles-1.19.3.tar.gz  
Starting galaxy collection install process  
Process install dependency map  
Starting collection install process  
Installing 'redhat.rhel_system_roles:1.19.3' to '/home/student/role-review/  
collections/ansible_collections/redhat/rhel_system_roles'  
redhat.rhel_system_roles:1.19.3 was installed successfully
```

- 2.2. Use the `ansible-galaxy collection list` command to verify that the collection is installed correctly and the system roles are now available.

```
[student@workstation role-review]$ ansible-galaxy collection list  
  
# /home/student/role-system/collections/ansible_collections  
Collection          Version  
-----  
redhat.rhel_system_roles 1.19.3  
  
...output omitted...
```

3. Create a playbook named `web_dev_server.yml` with a single play named `Configure Dev Web Server`. Configure the play to target the `dev_webserver` host group. Do not add any roles or tasks to the play yet.
Ensure that the play forces handlers to execute, because you might encounter an error when developing the playbook.

The completed `/home/student/role-review/web_dev_server.yml` playbook contains the following content:

```
---  
- name: Configure Dev Web Server  
  hosts: dev_webserver  
  force_handlers: yes
```

4. Verify the syntax of the playbook, then run the playbook. The syntax check should pass and the playbook should run successfully. The play only gathers facts because it has no roles or tasks yet.

```
[student@workstation role-review]$ ansible-navigator run \
> -m stdout web_dev_server.yml --syntax-check
playbook: /home/student/role-review/web_dev_server.yml
[student@workstation role-review]$ ansible-navigator run \
> -m stdout web_dev_server.yml

PLAY [Configure Dev Web Server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

5. Make sure to install any roles that are dependencies of roles in your playbook.

For example, the `apache.developer_configs` role that you write later in this lab depends on the `infra.apache` role.

Create a `roles/requirements.yml` file. This file installs the role from the Git repository at `git@workstation.lab.example.com:infra/apache`, use version `v1.4`, and name it `infra.apache` locally.

You can assume that your SSH keys are configured to allow you to get roles from this repository automatically. Install the role with the `ansible-galaxy` command.

- 5.1. Create a `roles` subdirectory for the playbook project.

```
[student@workstation role-review]$ mkdir -v roles
mkdir: created directory 'roles'
```

- 5.2. Create a `roles/requirements.yml` file and add an entry for the `infra.apache` role. Use version `v1.4` from the role's git repository.

The completed `roles/requirements.yml` file contains the following content:

```
- name: infra.apache
  src: git@workstation.lab.example.com:infra/apache
  scm: git
  version: v1.4
```

- 5.3. Install the project dependencies.

```
[student@workstation role-review]$ ansible-galaxy install \
> -r roles/requirements.yml -p roles
Starting galaxy role install process
- extracting infra.apache to /home/student/role-review/roles/infra.apache
- infra.apache (v1.4) was installed successfully
```

6. Initialize a new role named `apache.developer_configs` in the `roles` subdirectory.

Add the `infra.apache` role as a dependency for the new role, using the same information for name, source, version, and version control system as the `roles/requirements.yml` file.

The `developer_tasks.yml` file in the project directory contains tasks for the role. Move this file to the correct location in the role directory hierarchy for a tasks file that is used by this role, replacing the existing file in that location.

The `developer.conf.j2` file in the project directory is a Jinja2 template that is used by the tasks file. Move this file to the correct location for template files that are used by this role.

- 6.1. Use the `ansible-galaxy init` command to create a role skeleton for the `apache.developer_configs` role.

```
[student@workstation role-review]$ cd roles  
[student@workstation roles]$ ansible-galaxy init apache.developer_configs  
- Role apache.developer_configs was created successfully  
[student@workstation roles]$ cd ..  
[student@workstation role-review]$
```

- 6.2. Modify the `roles/apache.developer_configs/meta/main.yml` file of the `apache.developer_configs` role to reflect a dependency on the `infra.apache` role.

After editing, the `dependencies` variable is defined as follows:

```
dependencies:  
- name: infra.apache  
  src: git@workstation.lab.example.com:infra/apache  
  scm: git  
  version: v1.4
```

- 6.3. Overwrite the role's `tasks/main.yml` file with the `developer_tasks.yml` file.

```
[student@workstation role-review]$ mv -v developer_tasks.yml \  
> roles/apache.developer_configs/tasks/main.yml  
renamed 'developer_tasks.yml' -> 'roles/apache.developer_configs/tasks/main.yml'
```

- 6.4. Place the `developer.conf.j2` file in the role's `templates` directory.

```
[student@workstation role-review]$ mv -v developer.conf.j2 \  
> roles/apache.developer_configs/templates/  
renamed 'developer.conf.j2' -> 'roles/apache.developer_configs/templates/developer.conf.j2'
```

7. The `apache.developer_configs` role creates a user account and configures a web server instance for the list of users defined in the variable named `web_developers`.

The `web_developers.yml` file in the project directory defines the `web_developers` variable, which is the list of users for the role.

Review this file and put it in the correct location to define the `web_developers` variable for the development web server host group from your inventory.

- 7.1. Review the `web_developers.yml` file.

```
---
web_developers:
  - username: jdoe
    name: John Doe
    user_port: 9081
  - username: jdoe2
    name: Jane Doe
    user_port: 9082
```

A name, `username`, and `user_port` variable is defined for each web developer.

- 7.2. Place the `web_developers.yml` file in the `group_vars/dev_webserver` subdirectory.

```
[student@workstation role-review]$ mkdir -pv group_vars/dev_webserver
mkdir: created directory 'group_vars'
mkdir: created directory 'group_vars/dev_webserver'
[student@workstation role-review]$ mv -v web_developers.yml \
> group_vars/dev_webserver/
renamed 'web_developers.yml' -> 'group_vars/dev_webserver/web_developers.yml'
```

8. Add the `apache.developer_configs` role to the play in the `web_dev_server.yml` playbook.

Ensure that the `/home/student/role-review/web_dev_server.yml` playbook contains the following content:

```
---
- name: Configure Dev Web Server
  hosts: dev_webserver
  force_handlers: yes
  roles:
    - apache.developer_configs
```

9. Verify the syntax of the playbook, and then run the playbook. The syntax check should pass, but the playbook should fail when the `infra.apache` role attempts to restart Apache HTTPD.

```
[student@workstation role-review]$ ansible-navigator run \
> -m stdout web_dev_server.yml --syntax-check
playbook: /home/student/role-review/web_dev_server.yml
[student@workstation role-review]$ ansible-navigator run \
> -m stdout web_dev_server.yml

PLAY [Configure Dev Web Server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

...output omitted...

TASK [apache.developer_configs : Create user accounts] ****
```

```

changed: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John
Doe', 'user_port': 9081})
changed: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane
Doe', 'user_port': 9082})

...output omitted...

RUNNING HANDLER [infra.apache : restart firewalld] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [infra.apache : restart apache] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "msg": "Unable to
restart service httpd: Job for httpd.service failed because the control process
exited with error code.\nSee \\"systemctl status httpd.service\\" and \\"journalctl
-xeu httpd.service\\" for details.\n"}

NO MORE HOSTS LEFT ****

PLAY RECAP ****
servera.lab.example.com      : ok=14    changed=12    unreachable=0    failed=1
                           skipped=0   rescued=0    ignored=0
Please review the log for errors.

```

An error occurs when the `httpd` service is restarted. The `httpd` service daemon cannot bind to the non-standard HTTP ports, due to the SELinux context on those ports.

10. Apache HTTPD failed to restart in the preceding step because the network ports it uses for your developers are labeled with the wrong SELinux contexts.

You can use the provided variable file, `selinux.yml`, with the `rhel-system-roles.selinux` role to fix the issue.

Create a `pre_tasks` section for your play in the `web_dev_server.yml` playbook. In that section, use a task to include the `rhel-system-roles.selinux` role in a `block/rescue` structure, so that it is properly applied.

Move the `selinux.yml` file to the correct location so that its variables are set for the `dev_webserver` host group.

- 10.1. Add the `pre_tasks` section to the end of the play in the `web_dev_server.yml` playbook.

You can review the block in `/usr/share/doc/rhel-system-roles/selinux/example-selinux-playbook.yml` for a basic outline of how to apply the role.

Ensure that the `pre_tasks` section contains the following content:

```

pre_tasks:
  - name: Verify SELinux configuration
    block:
      - include_role:
          name: redhat.rhel_system_roles.selinux
    rescue:
      # Fail if failed for a different reason than selinux_reboot_required.
      - name: Handle general failure
        ansible.builtin.fail:
          msg: "SELinux role failed."
          when: not selinux_reboot_required

```

```

- name: Restart managed host
  ansible.builtin.reboot:
    msg: "Ansible rebooting system for updates."

- name: Reapply SELinux role to complete changes
  include_role:
    name: redhat.rhel_system_roles.selinux

```

- 10.2. The `selinux.yml` file contains variable definitions for the `rhel-system-roles.selinux` role. Use the file to define variables for the play's host group.

```

[student@workstation role-review]$ cat selinux.yml
---
# variables used by rhel-system-roles.selinux

selinux_policy: targeted
selinux_state: enforcing

selinux_ports:
- ports:
  - "9081"
  - "9082"
  proto: 'tcp'
  setype: 'http_port_t'
  state: 'present'

[student@workstation role-review]$ mv -v selinux.yml \
> group_vars/dev_webserver/
renamed 'selinux.yml' -> 'group_vars/dev_webserver/selinux.yml'

```

11. Verify the final `web_dev_server.yml` playbook and run a syntax check. The syntax check should pass.

Ensure that the final `web_dev_server.yml` playbook contains the following content:

```

---
- name: Configure Dev Web Server
  hosts: dev_webserver
  force_handlers: yes
  roles:
    - apache.developer_configs
  pre_tasks:
    - name: Verify SELinux configuration
      block:
        - include_role:
          name: redhat.rhel_system_roles.selinux
  rescue:
    # Fail if failed for a different reason than selinux_reboot_required.
    - name: Handle general failure
      ansible.builtin.fail:
        msg: "SELinux role failed."
        when: not selinux_reboot_required

    - name: Restart managed host

```

```
ansible.builtin.reboot:
  msg: "Ansible rebooting system for updates."

  - name: Reapply SELinux role to complete changes
    include_role:
      name: redhat.rhel_system_roles.selinux
```

**Note**

The `ansible-navigator run` command runs the play's tasks in the correct order, regardless of whether `pre_tasks` is at the end of the play or in the "correct" position of execution order in the playbook file.

It still runs the play's tasks in the correct order.

Validate that the `web_dev_server.yml` playbook passes a syntax check.

```
[student@workstation role-review]$ ansible-navigator run \
> -m stdout web_dev_server.yml --syntax-check
playbook: /home/student/role-review/web_dev_server.yml
```

- Run the `web_dev_server.yml` playbook. It should succeed.

```
[student@workstation role-review]$ ansible-navigator run \
> -m stdout web_dev_server.yml

PLAY [Configure Dev Web Server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [include_role : redhat.rhel_system_roles.selinux] ****
...output omitted...

TASK [redhat.rhel_system_roles.selinux : Set an SELinux label on a port] ****
changed: [servera.lab.example.com] => (item={'ports': ['9081', '9082'], 'proto': 'tcp', 'setype': 'http_port_t', 'state': 'present'})

TASK [redhat.rhel_system_roles.selinux : Set Linux user to SELinux user mapping]
 ***
 
TASK [redhat.rhel_system_roles.selinux : Get SELinux modules facts] ****
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.selinux : include_tasks] ****
skipping: [servera.lab.example.com]

TASK [infra.apache : Apache Package is installed] ****
ok: [servera.lab.example.com]

TASK [infra.apache : Apache Service is started] ****
changed: [servera.lab.example.com]
```

```

...output omitted...

TASK [apache.developer_configs : Create user accounts] ****
ok: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John Doe',
  'user_port': 9081})
ok: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane Doe',
  'user_port': 9082})

TASK [apache.developer_configs : Give student access to all accounts] ****
ok: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John Doe',
  'user_port': 9081})
ok: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane Doe',
  'user_port': 9082})

TASK [apache.developer_configs : Create content directory] ****
ok: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John Doe',
  'user_port': 9081})
ok: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane Doe',
  'user_port': 9082})

TASK [apache.developer_configs : Create skeleton index.html if needed] ****
ok: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John Doe',
  'user_port': 9081})
ok: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane Doe',
  'user_port': 9082})

TASK [apache.developer_configs : Set firewall port] ****
ok: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John Doe',
  'user_port': 9081})
ok: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane Doe',
  'user_port': 9082})

TASK [apache.developer_configs : Copy Per-Developer Config files] ****
ok: [servera.lab.example.com] => (item={'username': 'jdoe', 'name': 'John Doe',
  'user_port': 9081})
ok: [servera.lab.example.com] => (item={'username': 'jdoe2', 'name': 'Jane Doe',
  'user_port': 9082})

PLAY RECAP ****
servera.lab.example.com      : ok=21    changed=2     unreachable=0    failed=0
                               skipped=16   rescued=0    ignored=0

```

13. Test the configuration of the development web server. Verify that all endpoints are accessible and serving each developer's content.

```

[student@workstation role-review]$ curl servera
This is the production server on servera.lab.example.com
[student@workstation role-review]$ curl servera:9081
This is index.html for user: John Doe (jdoe)
[student@workstation role-review]$ curl servera:9082
This is index.html for user: Jane Doe (jdoe2)
[student@workstation role-review]$

```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade role-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish role-review
```

This concludes the section.

Summary

- *Ansible roles* help you to reuse and share Ansible code.
- Ansible Content Collections_ distribute related roles, modules, and other Ansible plug-ins that you can use in Ansible projects and automation execution environments.
- You use the `ansible-galaxy` command to manage Ansible roles and Ansible Content Collections.
- Red Hat provides Red Hat Certified Ansible Content Collections through a cloud-based service, *automation hub*. Red Hat and its partners support these Ansible Content Collections.
- You can distribute Red Hat Certified Ansible Content Collections or your own Ansible Content Collections from an on-premise *private automation hub*.
- *Ansible Galaxy* provides a library of third-party Ansible roles and Ansible Content Collections that are managed by the community and unsupported by Red Hat.
- *System roles* (`redhat.rhel_system_roles`) are provided as an RPM package or as an Ansible Content Collection that consists of several roles intended to help you configure managed host subsystems on multiple versions of Red Hat Enterprise Linux.
- Roles and Ansible Content Collections that a project needs can be specified by `requirements.yml` files, and can be installed in your Ansible project manually by using `ansible-galaxy`.

Chapter 8

Troubleshooting Ansible

Goal

Troubleshoot playbooks and managed hosts.

Objectives

- Troubleshoot generic issues with a new playbook and repair them.
- Troubleshoot failures on managed hosts when running a playbook.

Sections

- Troubleshooting Playbooks (and Guided Exercise)
- Troubleshooting Ansible Managed Hosts (and Guided Exercise)

Lab

- Troubleshooting Ansible

Troubleshooting Playbooks

Objectives

- Troubleshoot generic issues with a new playbook and repair them.

Debugging Playbooks

The output provided by the `ansible-navigator run` command is a good starting point for troubleshooting issues with your plays and the hosts on which they run.

Consider the following output from a playbook run:

```
PLAY [Service Deployment] ****
...output omitted...
TASK: [Install a service] ****
ok: [demoservera] => {
    "msg": "demoservera"
}
ok: [demoserverb] => {
    "msg": "demoserverb"
}

PLAY RECAP ****
demoservera    : ok=2      changed=0      unreachable=0      failed=0      skipped=0
                  rescued=0     ignored=0
demoserverb    : ok=2      changed=0      unreachable=0      failed=0      skipped=0
                  rescued=0     ignored=0
```

The previous output shows a PLAY header with the name of the play being run, followed by one or more TASK headers for the tasks in that play. Each of the TASK headers represents the associated task in the play, and it is run on all the managed hosts specified by the play's `hosts` parameter.

As each managed host runs each task, the name of the managed host is displayed under the TASK header, along with the task result for that host. Task results can be `ok`, `fatal`, `changed`, or `skipping`.

At the bottom of the output for each play, the PLAY RECAP section displays the number of tasks run for each managed host, by task result.

You can increase the verbosity of the output from `ansible-navigator run` by adding one or more `-v` options. The `ansible-navigator run -v` command provides additional debugging information, with up to four total levels.

Verbosity Configuration

Option	Description
<code>-v</code>	The output data is displayed.

Option	Description
-vv	Both the output and input data are displayed.
-vvv	Includes information about connections to managed hosts.
-vvvv	Includes additional information, such as the scripts that are executed on each remote host, and the user that is executing each script.

Examining Values of Variables with the Debug Module

You can use the `ansible.builtin.debug` module to provide insight into what is happening in the play. You can create a task that uses this module to display the value for a given variable at a specific point in the play. This can help you to debug tasks that use variables to communicate with each other (for example, using the output of a task as the input to the following one).

The following examples use the `msg` and `var` settings inside `ansible.builtin.debug` tasks. This first example displays the value at run time of the `ansible_facts['memfree_mb']` fact as part of a message printed to the output of `ansible-navigator run`.

```
- name: Display free memory
  ansible.builtin.debug:
    msg: "Free memory for this system is {{ ansible_facts['memfree_mb'] }}"
```

This second example displays the value of the `output` variable.

```
- name: Display the "output" variable
  ansible.builtin.debug:
    var: output
    verbosity: 2
```

The `verbosity` parameter controls when the `ansible.builtin.debug` module is executed. The value correlates to the number of `-v` options that are specified when the playbook is run. For example, if `-vv` is specified, and `verbosity` is set to 2 for a task, then that task is included in the debug output. The default value of the `verbosity` parameter is 0.

Reviewing Playbooks for Errors

Several issues can occur during a playbook run, many related to the syntax of either the playbook or any of the templates it uses, or due to connectivity issues with the managed hosts (for example, an error in the host name of the managed host in the inventory file).

A number of tools are available that you can use to review your playbook for syntax errors and other problems before you run it.

Checking Playbook Syntax for Problems

The `ansible-navigator run` command accepts the `--syntax-check` option, which tests your playbook for syntax errors instead of actually running it.

It is a good practice to validate the syntax of your playbook before using it or if you are having problems with it.

```
[student@demo ~]$ ansible-navigator run \
> -m stdout playbook.yml --syntax-check
```

Checking Playbooks for Issues and Following Good Practices

One of the best ways to make it easier for you to debug playbooks is for you to follow good practices when writing them in the first place. Some recommended practices for playbook development include:

- Use a concise description of the play's or task's purpose to name plays and tasks. The play name or task name is displayed when the playbook is executed. This also helps document what each play or task is supposed to accomplish, and possibly why it is needed.
- Use comments to add additional inline documentation about tasks.
- Make effective use of vertical white space. In general, organize task attributes vertically to make them easier to read.
- Consistent horizontal indentation is critical. Use spaces, not tabs, to avoid indentation errors. Set up your text editor to insert spaces when you press the Tab key to make this easier.
- Try to keep the playbook as simple as possible. Only use the features that you need.

Some Ansible practitioners at Red Hat have been working on an *unofficial* set of recommended practices for creating Ansible automation content, based on their own experiences in the field. Although not officially endorsed by Red Hat at this time, it can be a useful start for developing good practices of your own. See *Good Practices for Ansible* [<https://redhat-cop.github.io/automation-good-practices/>].

To help you follow good practices like these, Red Hat Ansible Automation Platform 2 provides a tool, `ansible-lint`, that uses a set of predefined rules to look for possible issues with your playbook. Not all the issues that it reports break your playbook, but a reported issue might indicate the presence of a more serious error.



Important

The `ansible-lint` command is a Technology Preview in Red Hat Ansible Automation Platform 2.2. Red Hat does not yet fully support this tool; for details, see the Knowledgebase article "What does a "Technology Preview" feature mean?" [<https://access.redhat.com/solutions/21101>].

For example, assume that you have the following playbook, `site.yml`:

```
- name: Configure servers with Ansible tools
hosts: all

tasks:
  - name: Make sure tools are installed
    package:
      name:
        - ansible-doc
        - ansible-navigator
```

Run the `ansible-lint site.yml` command to validate it. You might get the following output as a result:

```

WARNING Overriding detected file kind 'yaml' with 'playbook' for given positional
argument: site.yml
WARNING Listing 4 violation(s) that are fatal
yaml: trailing spaces (trailing-spaces) ①
site.yml:2

fqcn-builtins: Use FQCN for builtin actions. ②
site.yml:5 Task/Handler: Make sure tools are installed

yaml: trailing spaces (trailing-spaces) ③
site.yml:7

yaml: too many blank lines (1 > 0) (empty-lines) ④
site.yml:10

You can skip specific rules or tags by adding them to your configuration file:
# .config/ansible-lint.yml
warn_list: # or 'skip_list' to silence them completely
  - fqcn-builtins # Use FQCN for builtin actions.
  - yaml # Violations reported by yamllint.

Finished with 4 failure(s), 0 warning(s) on 1 files.

```

This run of `ansible-lint` found four style issues:

- ① Line 2 of the playbook (`hosts: all`) apparently has trailing white space, detected by the `yaml` rule. It is not a problem with the playbook directly, but many developers prefer not to have trailing white space in files stored in version control to avoid unnecessary differences as files are edited.
- ② Line 5 of the playbook (`package:`) does not use a FQCN for the module name on that task. It should be `ansible.builtin.package:` instead. This was detected by the `fqcn-builtins` rule.
- ③ Line 7 of the playbook also apparently has trailing white space.
- ④ The playbook ends with one or more blank lines, detected by the `yaml` rule.

The `ansible-lint` tool uses a local configuration file, which is either the `.ansible-lint` or `.config/ansible-lint.yml` file in the current directory. You can edit this configuration file to convert rule failures to warnings (by adding them as a list to the `warn_list` directive) or skip the checks entirely (by adding them as a list to the `skip_list` directive).

If you have a syntax error in the playbook, `ansible-lint` reports it just like `ansible-navigator run --syntax-check` does.

After you correct these style issues, the `ansible-lint site.yml` report is as follows:

```

WARNING Overriding detected file kind 'yaml' with 'playbook' for given positional
argument: site.yml

```

This is an advisory message that you can ignore, and the lack of other output indicates that `ansible-lint` did not detect any other style issues.

For more information on `ansible-lint`, see <https://docs.ansible.com/lint.html> and the `ansible-lint --help` command.



Important

The `ansible-lint` command evaluates your playbook based on the software on your workstation. It does not use the automation execution environment container that is used by `ansible-navigator`.

The `ansible-navigator` command has an experimental `lint` option that runs `ansible-lint` in your automation execution environment, but the `ansible-lint` tool needs to be installed inside the automation execution environment's container image for the option to work. This is currently not the case with the default execution environment. You need a custom execution environment to run `ansible-navigator lint` at this time.

In addition, the version of `ansible-lint` provided with Red Hat Ansible Automation Platform 2.2 assumes that your playbooks are using Ansible Core 2.13, which is the version currently used by the default execution environment. It does not support earlier Ansible 2.9 playbooks.

Reviewing Playbook Artifacts and Log Files

Red Hat Ansible Automation Platform can log the output of playbook runs that you make from the command line in a number of different ways.

- `ansible-navigator` can produce *playbook artifacts* that store information about runs of playbooks in JSON format.
- You can log information about playbook runs to a text file in a location on the system to which you can write.

Playbook Artifacts from Automation Content Navigator

The `ansible-navigator` command produces playbook artifact files by default each time you use it to run a playbook. These files record information about the playbook run, and can be used to review the results of the run when it completes, to troubleshoot issues, or be kept for compliance purposes.

Each playbook artifact file is named based on the name of the playbook you ran, followed by the word `artifact`, and then the time stamp of when the playbook was run, ending with the `.json` file extension.

For example, if you run the command `ansible-navigator run site.yml` at 20:00 UTC on July 22, 2022, the resulting file name of the artifact file could be:

```
site-artifact-2022-07-22T20:00:04.019343+00:00.json
```

You can review the contents of these files with the `ansible-navigator replay` command. If you include the `-m stdout` option, then the output of the playbook run is printed to your terminal as if it had just run. However, if you omit that option, you can examine the results of the run interactively.

For example, you run the following playbook, `site.yml`, and it fails but you do not know why. You run `ansible-navigator run site.yml --syntax-check` and the `ansible-lint` command, but neither command reports any issues.

```
- name: Configure servers with Ansible tools
  hosts: all

  tasks:
    - name: Make sure tools are installed
      ansible.builtin.package:
        name:
          - ansible-doc
          - ansible-navigator
```

To troubleshoot further, you run `ansible-navigator replay` in interactive mode on the resulting artifact file, which opens the following output in your terminal:

The screenshot shows the initial replay screen of the Ansible Navigator. It displays a summary table with the following data:

Play name	Ok	Changed	Unreachable	Failed	Skipped	Ignored	In progress	Task count	Progress
Configure se	2	0	0	2	0	0	0	4	Complete

At the bottom of the screen, there is a terminal-style footer with navigation keys: `^b/PgUp` page up, `^f/PgDn` page down, `↑↓ scroll`, `esc` back, `[0-9]` goto, `:help help`, and a red bar indicating `Failed`.

Figure 8.1: Initial replay screen

If you enter `:0` to view the play, the following output is printed:

The screenshot shows the detailed results of the play, listing tasks for each host. The data is presented in a table:

Result	Host	Number	Changed	Task	Task action	Duration
0 Ok	server-2.example.com	0	False	Gathering Facts	gather_facts	1s
1 Ok	server-1.example.com	1	False	Gathering Facts	gather_facts	1s
2 Failed	server-2.example.com	2	False	Make sure tools are ansible.builtin.packag	gather_facts	4s
3 Failed	server-1.example.com	3	False	Make sure tools are ansible.builtin.packag	gather_facts	2s

At the bottom of the screen, there is a terminal-style footer with navigation keys: `^b/PgUp` page up, `^f/PgDn` page down, `↑↓ scroll`, `esc` back, `[0-9]` goto, `:help he`, and a red bar indicating `Failed`.

Figure 8.2: Play results by machine and task

It looks like the task `Make sure tools are installed` failed on both the `server-1.example.com` and `server-2.example.com` hosts. By entering `:2`, you can look at the failure for the `server-2.example.com` host:

```
Play name: Configure servers with Ansible tools:2
Task name: Make sure tools are installed
Failed: server-2.example.com Failed to install some of the specified packages
0 ---
1 duration: 4.184921
2 end: '2022-07-25T18:38:40.786573'
3 event_loop: null
4 host: server-2.example.com
5 ignore_errors: null
6 play: Configure servers with Ansible tools
7 play_pattern: all
8 playbook: /home/student/project/site.yml
9 remote_addr: server-2.example.com
10 res:
11   _ansible_no_log: null
12   changed: false
13   failures:
14     - No package ansible-doc available.
15   invocation:
^b/PgUp page up^f/PgDn page down^l scroll^c back- previous+ next[0-9] goto:help Failed
```

Figure 8.3: Task results for a specific machine

The task is attempting to use the `ansible.builtin.package` module to install the `ansible-doc` package, and that package is not available in the RPM package repositories used by the `server-2.example.com` host, so the task failed. (You might discover that the `ansible-doc` command is now provided as part of the `ansible-navigator` RPM package as the `ansible-navigator doc` command, and changing the task accordingly fixes the problem.)

Another useful thing to know is that you can look at the results of a successful `Gathering Facts` task and the debugging output includes the values of all the facts that were gathered:

```
Play name: Configure servers with Ansible tools:1
Task name: Gathering Facts
Ok: server-1.example.com
247   sectors: '2097151'
248   sectorsize: '512'
249   size: 1024.00 MB
250   support_discard: '0'
251   vendor: QEMU
252   virtual: 1
253   ansible_distribution: RedHat
254   ansible_distribution_file_parsed: true
255   ansible_distribution_file_path: /etc/redhat-release
256   ansible_distribution_file_search_string: Red Hat
257   ansible_distribution_file_variety: RedHat
258   ansible_distribution_major_version: '9'
259   ansible_distribution_release: Plow
260   ansible_distribution_version: '9.0'
261   ansible_dns:
262     nameservers:
^b/PgUp page up^f/PgDn page down^l scroll^c back- previous+ next[0-9] goto:help Failed
```

Figure 8.4: Task results for successful fact gathering

This can help you debug issues involving Ansible facts without adding a task to the play that uses the `ansible.builtin.debug` module to print out fact values.

**Important**

You might not want to save playbook artifacts for several reasons.

- You are concerned about sensitive information being saved in the log file.
- You need to provide interactive input, such as a password, to `ansible-navigator` for some reason.
- You do not want the files to clutter up the project directory.

You can keep the files from being generated by creating an `ansible-navigator.yml` file in the project directory that disables the playbook artifacts:

```
ansible-navigator:
  playbook-artifact:
    enable: false
```

Logging Output to a Text File

Ansible provides a built-in logging infrastructure that can be configured through the `log_path` parameter in the `default` section of the `ansible.cfg` configuration file, or through the `$ANSIBLE_LOG_PATH` environment variable. If any or both are configured, Ansible stores output from `ansible-navigator` commands as text in the log file configured. This mechanism also works with earlier tools such as `ansible-playbook`.

If you configure Ansible to write log files to `/var/log`, then Red Hat recommends that you configure `logrotate` to manage them.

**References****Configuring Ansible – Ansible Documentation**

https://docs.ansible.com/ansible/latest/installation_guide/intro_configuration.html

ansible.builtin.debug module – Print statements during execution – Ansible Documentation

https://docs.ansible.com/ansible/latest/modules/debug_module.html

Tips and tricks – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html

Good Practices for Ansible

<https://redhat-cop.github.io/automation-good-practices/>

Ansible Lint Documentation

<https://docs.ansible.com/lint.html>

► Guided Exercise

Troubleshooting Playbooks

In this exercise, you troubleshoot a playbook that has been given to you that does not work properly.

Outcomes

- You should be able to troubleshoot and resolve issues in playbooks.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start troubleshoot-playbook
```

Instructions

- 1. Change into the `/home/student/troubleshoot-playbook/` directory.

```
[student@workstation ~]$ cd ~/troubleshoot-playbook/
[student@workstation troubleshoot-playbook]$
```

- 2. Create a file named `ansible.cfg` in the current directory. Configure the `log_path` parameter to write Ansible logs to the `/home/student/troubleshoot-playbook/ansible.log` file. Configure the `inventory` parameter to use the `/home/student/troubleshoot-playbook/inventory` file deployed by the `lab` script.

The completed `ansible.cfg` file should contain the following:

```
[defaults]
log_path = /home/student/troubleshoot-playbook/ansible.log
inventory = /home/student/troubleshoot-playbook/inventory
```

- 3. Run the `samba.yml` playbook. It fails with an error.

This playbook would set up a Samba server if everything were correct. However, the run fails because the `random_var` variable definition is not in double quotation marks. (If a colon is part of a value, the value must be protected by single or double quotation marks.) Read the error message to see how `ansible-navigator run` reports the problem. Notice that the variable `random_var` is assigned a value that contains a colon and is not protected by double quotation marks.

```
[student@workstation troubleshoot-playbook]$ ansible-navigator run \
> -m stdout samba.yml
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got
from each:
JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  mapping values are not allowed in this context

The error appears to be in '/home/student/troubleshoot-playbook/samba.yml': line
 8, column 30, but may be elsewhere in the file depending on the exact syntax
problem.

The offending line appears to be:

  install_state: installed
    random_var: This is colon: test
                  ^ here
```

- ▶ 4. Confirm that `ansible-navigator` has logged the error to the `/home/student/troubleshoot-playbook/ansible.log` file.

```
[student@workstation troubleshoot-playbook]$ tail ansible.log

The error appears to be in '/home/student/troubleshoot-playbook/samba.yml': line
 8, column 30, but may be elsewhere in the file depending on the exact syntax
problem.

The offending line appears to be:

  install_state: installed
    random_var: This is colon: test
                  ^ here
```

- ▶ 5. Edit the `samba.yml` playbook and correct the error by adding double quotation marks to the entire value being assigned to `random_var`. The corrected version of the playbook contains the following content:

```
...output omitted...
vars:
  install_state: installed
  random_var: "This is colon: test"
...output omitted...
```

- 6. Verify the syntax of the playbook using the `--syntax-check` option. The `ansible-navigator` command issues another error because there is too much white space indenting the last task, `deliver samba config`.

```
[student@workstation troubleshoot-playbook]$ ansible-navigator run \
> -m stdout samba.yml --syntax-check
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got
from each:
JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  did not find expected '-' indicator

The error appears to be in '/home/student/troubleshoot-playbook/samba.yml': line
  38, column 6, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

  - name: Deliver samba config
    ^ here
Please review the log for errors.
```

- 7. Edit the playbook and remove the extra space for all lines in that task. The corrected playbook should appear as follows:

```
...output omitted...
- name: Configure firewall for samba
  ansible.posix.firewalld:
    state: enabled
    permanent: true
    immediate: true
    service: samba

- name: Deliver samba config
  ansible.builtin.template:
    src: samba.j2
    dest: /etc/samba/smb.conf
    owner: root
    group: root
    mode: 0644
```

- 8. Run the playbook using the `--syntax-check` option. The `ansible-navigator` command issues an error because the `install_state` variable is being used as a parameter in the `install samba` task and is not quoted. (If a Jinja2 expression is at the start of a value, the value must be protected by double quotation marks.)

```
[student@workstation troubleshoot-playbook]$ ansible-navigator run \
> -m stdout samba.yml --syntax-check
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got
from each:
JSON: Expecting value: line 1 column 1 (char 0)
```

```
Syntax Error while loading YAML.
  found unacceptable key (unhashable type: 'AnsibleMapping')

The error appears to be in '/home/student/troubleshoot-playbook/samba.yml': line
 14, column 17, but may be elsewhere in the file depending on the exact syntax
problem.

The offending line appears to be:

    name: samba
    state: {{ install_state }}
      ^ 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 }}"

Please review the log for errors.
```

- ▶ 9. Edit the playbook and correct the `install_samba` task. The reference to the `install_state` variable should be in double quotation marks. The resulting file should consist of the following content:

```
...output omitted...
tasks:
  - name: Install samba
    ansible.builtin.dnf:
      name: samba
      state: "{{ install_state }}"
...output omitted...
```

- ▶ 10. Run the playbook using the `--syntax-check` option. It should not show any additional syntax errors.

```
[student@workstation troubleshoot-playbook]$ ansible-navigator run \
> -m stdout samba.yml --syntax-check
playbook: /home/student/troubleshoot-playbook/samba.yml
```

- ▶ 11. Run the playbook again. The `debug` `install_state` variable task returns the message `The state for the samba service is installed`. This task makes use of the `ansible.builtin.debug` module, and displays the value of the `install_state`

variable. An error is also shown in the `deliver_samba_config` task, because no `samba.j2` file is available in the `/home/student/troubleshoot-playbook/` directory.

```
[student@workstation troubleshoot-playbook]$ ansible-navigator run \
> -m stdout samba.yml

PLAY [Install a samba server] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install samba] ****
changed: [servera.lab.example.com]

TASK [Install firewalld] ****
ok: [servera.lab.example.com]

TASK [Debug install_state variable] ****
ok: [servera.lab.example.com] => {
    "msg": "The state for the samba service is installed"
}

TASK [Start firewalld] ****
ok: [servera.lab.example.com]

TASK [Configure firewall for samba] ****
changed: [servera.lab.example.com]

TASK [Deliver samba config] ****
An exception occurred during task execution. To see the full traceback, use -vvv.
The error was: If you are using a module and expect the file to exist on the
remote, see the remote_src option
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "msg": "Could not
  find or access 'samba.j2'\nSearched in:\n\t/home/student/troubleshoot-playbook/
  templates/samba.j2\n\t/home/student/troubleshoot-playbook/samba.j2\n\t/home/
  student/troubleshoot-playbook/templates/samba.j2\n\t/home/student/troubleshoot-
  playbook/samba.j2 on the Ansible Controller.\nIf you are using a module and expect
  the file to exist on the remote, see the remote_src option"}

PLAY RECAP ****
servera.lab.example.com      : ok=6      changed=2      unreachable=0      failed=1
                               skipped=0     rescued=0     ignored=0
Please review the log for errors.
```

- 12. Edit the playbook and correct the `src` parameter in the `deliver samba config` task to be `samba.conf.j2`. The finished file should consist of the following content:

```
...output omitted...
- name: Deliver samba config
  ansible.builtin.template:
    src: samba.conf.j2
    dest: /etc/samba/smb.conf
    owner: root
...output omitted...
```

- 13. Run the playbook again and all tasks should succeed.

```
[student@workstation troubleshoot-playbook]$ ansible-navigator run \
> -m stdout samba.yml

PLAY [Install a samba server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install samba] ****
ok: [servera.lab.example.com]

TASK [Install firewalld] ****
ok: [servera.lab.example.com]

TASK [Debug install_state variable] ****
ok: [servera.lab.example.com] => {
    "msg": "The state for the samba service is installed"
}

TASK [Start firewalld] ****
ok: [servera.lab.example.com]

TASK [Configure firewall for samba] ****
ok: [servera.lab.example.com]

TASK [Deliver samba config] ****
changed: [servera.lab.example.com]

TASK [Start samba] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=8      changed=2      unreachable=0      failed=0
skipped=0      rescued=0      ignored=0
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish troubleshoot-playbook
```

This concludes the section.

Troubleshooting Ansible Managed Hosts

Objectives

- Troubleshoot failures on managed hosts when running a playbook.

Troubleshooting Connections

Many common problems when using Ansible to manage hosts are associated with connections to the host and with configuration problems around the remote user and privilege escalation.

Problems Authenticating to Managed Hosts

If you are having problems authenticating to a managed host, make sure that you have `remote_user` set correctly in your configuration file or in your play.

For example, you might see the following output when running a playbook that is designed to connect to the remote `root` user account:

```
[student@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [Install a samba server] ****
TASK [Gathering Facts] ****
fatal: [host.lab.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: developer@host: Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).", "unreachable": true}

PLAY RECAP ****
host.lab.example.com    : ok=0      changed=0      unreachable=1    failed=0
                      skipped=0    rescued=0    ignored=0
Please review the log for errors.
```

In this case, `ansible-navigator` is trying to connect as the `developer` user account, according to the preceding output. One reason this might happen is if `ansible.cfg` has been configured in the project to set the `remote_user` to the `developer` user instead of the `root` user.

Another reason you could see a "permission denied" error like this is if you do not have the correct SSH keys set up, or did not provide the correct password for that user.

```
[root@controlnode ~]# ansible-navigator run \
> -m stdout playbook.yml

TASK [Gathering Facts] ****
fatal: [host.lab.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: root@host: Permission denied (publickey,gssapi-keyex,gssapi-with-mic).", "unreachable": true}
```

```
PLAY RECAP ****
host : ok=0    changed=0    unreachable=1    failed=0    skipped=0    rescued=0
      ignored=0
```

Please review the log for errors.

In the preceding example, the playbook is attempting to connect to the host machine as the `root` user but the SSH key for the `root` user on the `controlnode` machine has not been added to the `authorized_keys` file for the `root` user on the host machine.



Note

To summarize, you could see similar "permission denied" errors in the following situations:

- You try to connect as the wrong `remote_user` for your authentication credentials
- You connect as the correct `remote_user` but the authentication credentials are missing or incorrect

Problems with Name or Address Resolution

A more subtle problem has to do with inventory settings. For a complex server with multiple network addresses, you might need to use a particular address or DNS name when connecting to that system. You might not want to use that address as the machine's inventory name for better readability. You can set a host inventory variable, `ansible_host`, that overrides the inventory name with a different name or IP address and be used by Ansible to connect to that host. This variable could be set in the `host_vars` file or directory for that host, or could be set in the inventory file itself.

For example, the following inventory entry configures Ansible to connect to `192.0.2.4` when processing the `web4.phx.example.com` host:

```
web4.phx.example.com ansible_host=192.0.2.4
```

This is a useful way to control how Ansible connects to managed hosts. However, it can also cause problems if the value of `ansible_host` is incorrect.

Problems with Privilege Escalation

If your playbook connects as a `remote_user` and then uses privilege escalation to become the `root` user (or some other user), make sure that `become` is set properly, and that you are using the correct value for the `become_user` directive. The setting for `become_user` is `root` by default.

If the remote user needs to provide a `sudo` password, you should confirm that you are providing the correct `sudo` password, and that `sudo` on the managed host is configured correctly.

```
[user@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

TASK [Gathering Facts] ****
fatal: [host]: FAILED! => {"msg": "Missing sudo password"}
```

```
PLAY RECAP ****
host : ok=0    changed=0    unreachable=0    failed=1    skipped=0
      rescued=0   ignored=0

Please review the log for errors.
```

In the preceding example, the playbook is attempting to run sudo on the host machine but it fails. The `remote_user` is not set up to run sudo commands without a password on the host machine. Either sudo on the host machine is not properly configured, or it is supposed to require a sudo password and you neglected to provide one when running the playbook.



Important

Normally, `ansible-navigator` runs as root inside its automation execution environment. However, the root user in the container has access to SSH keys provided by the user that ran `ansible-navigator` on the workstation. This can be slightly confusing when you are trying to debug `remote_user` and `become` directives, especially if you are used to the earlier `ansible-playbook` command that runs as the user on the workstation.

Problems with Python on Managed Hosts

For normal operation, Ansible requires a Python interpreter to be installed on managed hosts running Linux. Ansible attempts to locate a Python interpreter on each Linux managed host the first time a module is run on that host.

```
[user@controlnode ~]$ ansible-navigator run \
> -m stdout playbook.yml

TASK [Gathering Facts] ****
fatal: [host]: FAILED! => {"ansible_facts": {}, "changed": false,
"failed_modules": {"ansible.legacy.setup": {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "failed": true,
"module_stderr": "Shared connection to host closed.\r\n", "module_stdout": "/bin/sh: 1: /usr/bin/python: not found\r\n", "msg": "The module failed to execute correctly, you probably need to set the interpreter.\nSee stdout/stderr for the exact error", "rc": 127, "warnings": ["No python interpreters found for host
host (tried ['python3.10', 'python3.9', 'python3.8', 'python3.7', 'python3.6', 'python3.5', '/usr/bin/python3', '/usr/libexec/platform-python', 'python2.7', 'python2.6', '/usr/bin/python', 'python'])"]}}, "msg": "The following modules failed to execute: ansible.legacy.setup\r\n"}

PLAY RECAP ****
host : ok=0    changed=0    unreachable=0    failed=1    skipped=0    rescued=0
      ignored=0

Please review the log for errors.
```

In the preceding example, the playbook fails because Ansible is unable to find a suitable Python interpreter on the host machine.

Using Check Mode as a Testing Tool

You can use the `ansible-navigator run --check` command to run "smoke tests" on a playbook. This option runs the playbook, connecting to the managed hosts normally but without making changes to them.

If a module used within the playbook supports "check mode", then the changes that would have been made to the managed hosts are displayed but not performed. If check mode is not supported by a module, then `ansible-navigator` does not display the predicted changes, but the module still takes no action.

```
[student@demo ~]$ ansible-navigator run \
> -m stdout playbook.yml --check
```



Important

The `ansible-navigator run --check` command might not work properly if your tasks use conditionals. One reason for this might be that the conditionals depend on some preceding task in the play actually running so that the condition evaluates correctly.

You can force tasks to always run in check mode or to always run normally with the `check_mode` setting. If a task has `check_mode: yes` set, it always runs in its check mode and does not perform any action, even if you do not pass the `--check` option to `ansible-navigator`. Likewise, if a task has `check_mode: no` set, it always runs normally, even if you pass `--check` to `ansible-navigator`.

The following task always runs in check mode, and does not make changes to managed hosts.

```
tasks:
  - name: task always in check mode
    ansible.builtin.shell: uname -a
    check_mode: yes
```

The following task always runs normally, even when started with `ansible-navigator run --check`.

```
tasks:
  - name: task always runs even in check mode
    ansible.builtin.shell: uname -a
    check_mode: no
```

This can be useful because you can run most of a playbook normally and test individual tasks with `check_mode: yes`. Many plays use facts or set variables to conditionally run tasks. Conditional tasks might fail if a fact or variable is undefined, due to the task that collects them or sets them not executing on a managed node. You can use `check_mode: no` on tasks that gather facts or set variables but do not otherwise change the managed node. This enables the play to proceed further when using `--check` mode.

A task can determine if the playbook is running in check mode by testing the value of the magic variable `ansible_check_mode`. This Boolean variable is set to `true` if the playbook is running in check mode.

**Warning**

Tasks that have `check_mode: no` set run even when the playbook is run with `ansible-navigator run --check`. Therefore, you cannot trust that the `--check` option makes no changes to managed hosts, without inspecting the playbook and any roles or tasks associated with it.

**Note**

If you have older playbooks that use `always_run: yes` to force tasks to run normally even in check mode, you need to replace that code with `check_mode: no` in Ansible 2.6 and later.

The `ansible-navigator` command also provides a `--diff` option. This option reports the changes made to the template files on managed hosts. If used with the `--check` option, those changes are displayed in the command's output but not actually made.

```
[student@demo ~]$ ansible-navigator run \
> -m stdout playbook.yml --check --diff
```

Testing with Modules

Some modules can provide additional information about the status of a managed host. The following list includes some Ansible modules that can be used to test and debug issues on managed hosts.

The `ansible.builtin.uri` module provides a way to verify that a RESTful API is returning the required content.

```
tasks:
  - ansible.builtin.uri:
      url: http://api.myapp.example.com
      return_content: yes
      register: apiresponse

  - ansible.builtin.fail:
      msg: 'version was not provided'
      when: "'version' not in apiresponse.content"
```

The `ansible.builtin.script` module runs a script on managed hosts, and fails if the return code for that script is nonzero. The script must exist in the Ansible project and is transferred to and run on the managed hosts.

```
tasks:
  - ansible.builtin.script: scripts/check_free_memory --min 2G
```

The `ansible.builtin.stat` module gathers facts for a file much like the `stat` command. You can use it to register a variable and then test to determine if the file exists or to get other information about the file. If the file does not exist, the `ansible.builtin.stat` task does not fail, but its registered variable reports `false` for `*['stat']['exists']`.

In this example, an application is still running if `/var/run/app.lock` exists, in which case the play should abort.

```
tasks:
  - name: Check if /var/run/app.lock exists
    ansible.builtin.stat:
      path: /var/run/app.lock
      register: lock

  - name: Fail if the application is running
    ansible.builtin.fail:
      when: lock['stat']['exists']
```

The `ansible.builtin.assert` module is an alternative to the `ansible.builtin.fail` module. The `ansible.builtin.assert` module supports a `that` option that takes a list of conditionals. If any of those conditionals are false, the task fails. You can use the `success_msg` and `fail_msg` options to customize the message it prints if it reports success or failure.

The following example repeats the preceding one, but uses `ansible.builtin.assert` instead of the `ansible.builtin.fail` module:

```
tasks:
  - name: Check if /var/run/app.lock exists
    ansible.builtin.stat:
      path: /var/run/app.lock
      register: lock

  - name: Fail if the application is running
    ansible.builtin.assert:
      that:
        - not lock['stat']['exists']
```

Running Ad Hoc Commands with Ansible

An ad hoc command is a way of executing a single Ansible task quickly, one that you do not need to save to run again later. They are simple, online operations that can be run without writing a playbook.

Ad hoc commands do not run inside an automation execution environment container. Instead, they run using Ansible software, roles, and collections installed directly on your workstation. To use ad hoc Ansible Core 2.13 commands, you need to install the `ansible-core` RPM package on your workstation.



Important

The `ansible-core` RPM package provides only the modules in the `ansible.builtin` Ansible Content Collection. If you need modules from other collections, you need to install those on your workstation separately.

Ad hoc commands are useful for quick tests and troubleshooting. For example, you can use an ad hoc command to make sure that hosts are reachable using the `ansible.builtin.ping` module. You could use another ad hoc command to view resource usage on a group of hosts using the `ansible.builtin.command` module.

Ad hoc commands do have their limits, and in general you want to use Ansible Playbooks to realize the full power of Ansible.

Use the `ansible` command to run ad hoc commands:

```
[user@controlnode ~]$ ansible host-pattern -m module [-a 'module arguments'] \
> [-i inventory]
```

The `host-pattern` argument is used to specify the managed hosts against which the ad hoc command should be run. The `-i` option is used to specify a different inventory location to use from the default in the current Ansible configuration file. The `-m` option specifies the module that Ansible should run on the targeted hosts. The `-a` option takes a list of arguments for the module as a quoted string.



Note

If you use the `ansible` command but do not specify a module with the `-m` option, the `ansible.builtin.command` module is used by default. It is always best to specify the module you intend to use, even if you intend to use the `ansible.builtin.command` module.

Ansible ad hoc commands can be useful, but should be kept to troubleshooting and one-time use cases. For example, if you are aware of multiple pending network changes, it is more efficient to create a playbook with an `ansible.builtin.ping` task that you can run multiple times, compared to typing out a one-time use ad hoc command multiple times.

Testing Managed Hosts Using Ad Hoc Commands

The following examples illustrate some tests that can be made on a managed host using ad hoc commands.

You have used the `ansible.builtin.ping` module to test whether you can connect to managed hosts. Depending on the options that you pass, you can also use it to test whether privilege escalation and credentials are correctly configured.

```
[student@demo ~]$ ansible demohost -m ansible.builtin.ping
demohost | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
[student@demo ~]$ ansible demohost -m ansible.builtin.ping --become
demohost | FAILED! => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "module_stderr": "sudo: a password is required\n",
    "module_stdout": "",
```

```
"msg": "MODULE FAILURE\nSee stdout/stderr for the exact error",
"rc": 1
}
```

This example returns the current available space on the disks configured on the demohost managed host. That can be useful to confirm that the file system on the managed host is not full.

```
[student@demo ~]$ ansible demohost -m ansible.builtin.command -a 'df'
```

This example returns the current available free memory on the demohost managed host.

```
[student@demo ~]$ ansible demohost -m ansible.builtin.command -a 'free -m'
```



References

Check Mode ("Dry Run") – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_checkmode.html

Testing Strategies – Ansible Documentation

https://docs.ansible.com/ansible/latest/reference_appendices/test_strategies.html

► Guided Exercise

Troubleshooting Ansible Managed Hosts

In this exercise, you troubleshoot task failures that are occurring on one of your managed hosts when running a playbook.

Outcomes

- You should be able to troubleshoot managed hosts.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start troubleshoot-host
```

Instructions

- 1. Change into the `/home/student/troubleshoot-host/` directory.

```
[student@workstation ~]$ cd ~/troubleshoot-host/  
[student@workstation troubleshoot-host]$
```

- 2. Run the `mailrelay.yml` playbook using check mode.

```
[student@workstation troubleshoot-host]$ ansible-navigator run \  
> -m stdout mailrelay.yml --check  
PLAY [Create mail relay servers] ****  
...output omitted...  
  
TASK [Check main.cf file] ****  
ok: [servera.lab.example.com]  
  
TASK [Verify main.cf file exists] ****  
ok: [servera.lab.example.com] => {  
    "msg": "The main.cf file exists"  
}  
...output omitted...  
  
TASK [Start and enable mail services] ****  
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "msg": "Could not  
find the requested service postfix: host"}
```

```
...output omitted...
PLAY RECAP ****
servera.lab.example.com : ok=5    changed=2    unreachable=0    failed=1
                          skipped=0   rescued=0   ignored=0
```

The verify main.cf file exists task uses the ansible.builtin.stat module. It confirms that main.cf exists on the servera.lab.example.com host.

The Start and enable mail services task failed. It could not start the postfix service because you ran the playbook using check mode and therefore the play did not install the postfix package.



Important

The task failed because earlier tasks in the play did not ensure that postfix was installed on the servera host, because you ran the playbook in check mode. This failure happened because the playbook did not actually make changes to the host that it normally would have if you ran it normally.

- ▶ 3. Run the playbook again, but without specifying check mode. The error in the Start and enable mail services task should disappear and the playbook should run successfully.

```
[student@workstation troubleshoot-host]$ ansible-navigator run \
> -m stdout mailrelay.yml

PLAY [Create mail relay servers] ****
...output omitted...

TASK [Check main.cf file] ****
ok: [servera.lab.example.com]

TASK [Verify main.cf file exists] ****
ok: [servera.lab.example.com] => {
    "msg": "The main.cf file exists"
}

TASK [Start and enable mail services] ****
changed: [servera.lab.example.com]
...output omitted...

PLAY RECAP ****
servera.lab.example.com : ok=8    changed=5    unreachable=0    failed=0
                          skipped=1   rescued=0   ignored=0
```

- 4. Edit the `mailrelay.yml` playbook and add a task to enable the `smtp` service through the firewall. Add the task as the last task, before the handlers.

```
...output omitted...
- name: Postfix firewalld config
  ansible.posix.firewalld:
    state: enabled
    permanent: true
    immediate: true
    service: smtp
...output omitted...
```

- 5. Run the `mailrelay.yml` playbook. The `postfix firewalld config` task runs with no errors.

```
[student@workstation troubleshoot-host]$ ansible-navigator run \
> -m stdout mailrelay.yml
PLAY [Create mail relay servers] ****
...output omitted...
TASK [Postfix firewalld config] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=8      changed=2      unreachable=0      failed=0
skipped=1      rescued=0      ignored=0
```

- 6. Use `telnet` to test if the SMTP service is listening on port TCP/25 on the `servera.lab.example.com` host. Disconnect when you are finished.

```
[student@workstation troubleshoot-host]$ telnet servera.lab.example.com 25
Trying 172.25.250.10...
Connected to servera.lab.example.com.
Escape character is '^J'.
220 servera.lab.example.com ESMTP Postfix
quit
221 2.0.0 Bye
Connection closed by foreign host.
```

- 7. Run the `samba.yml` playbook. The first task fails with an error related to an SSH connection problem.

```
[student@workstation troubleshoot-host]$ ansible-navigator run \
> -m stdout samba.yml
PLAY [Install a samba server] ****
TASK [Gathering Facts] ****
fatal: [servera.lab.exammpole.com]: UNREACHABLE! => {"changed": false,
  "msg": "Failed to connect to the host via ssh: ssh: connect to host
servera.lab.exammpole.com port 22: Connection timed out", "unreachable": true}
```

```
PLAY RECAP ****
servera.lab.example.com    : ok=0      changed=0      unreachable=1      failed=0
                           skipped=0    rescued=0    ignored=0
Please review the log for errors.
```

- 8. Make sure that you can connect to the `servera.lab.example.com` managed host as the `devops` user using SSH, and that the correct SSH keys are in place. Log off again when you have finished.

```
[student@workstation troubleshoot-host]$ ssh devops@servera.lab.example.com
...output omitted...
[devops@servera ~]$ exit
logout
Connection to servera.lab.example.com closed.
```

That is working normally.

- 9. Test to see if you can run modules on the `servera.lab.example.com` managed host by using an ad hoc command that runs the `ansible.builtin.ping` module.

```
[student@workstation troubleshoot-host]$ ansible servera.lab.example.com \
> -m ansible.builtin.ping
servera.lab.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Based on the preceding output, that is also working, and successfully connected to the managed host.

This should suggest to you that the problem is not with the SSH configuration and credentials, or with the ad hoc command that you used. So the question now is why the ad hoc command worked and the `ansible-navigator` command did not. There might be a problem with the play in the playbook, or with the inventory.

- 10. Rerun the `samba.yml` playbook with `-vvvv` to get more information about the run. An error is issued because the `servera.lab.example.com` managed host is not reachable.

```
[student@workstation troubleshoot-host]$ ansible-navigator run \
> -m stdout -vvvv samba.yml
ansible-playbook [core 2.13.0]
  config file = /home/student/troubleshoot-host/ansible.cfg
  configured module search path = ['/home/runner/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.9/site-packages/ansible
  ansible collection location = /home/runner/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/bin/ansible-playbook
  python version = 3.9.7 (default, Sep 13 2021, 08:18:39) [GCC 8.5.0 20210514 (Red Hat 8.5.0-3)]
```

```
jinja version = 3.0.3
libyaml = True
Using /home/student/troubleshoot-host/ansible.cfg as config file
...output omitted...

PLAYBOOK: samba.yml ****
Positional arguments: /home/student/troubleshoot-host/samba.yml
verbosity: 4
connection: smart
timeout: 10
become_method: sudo
tags: ('all',)
inventory: ('/home/student/troubleshoot-host/inventory',)
forks: 5
1 plays in /home/student/troubleshoot-host/samba.yml

PLAY [Install a samba server] ****

TASK [Gathering Facts] ****
task path: /home/student/troubleshoot-host/samba.yml:2
<servera.lab.exammpole.com> ESTABLISH SSH CONNECTION FOR USER: devops
...output omitted...
fatal: [servera.lab.exammpole.com]: UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: OpenSSH_8.0p1, OpenSSL 1.1.1k
FIPS 25 Mar 2021\r\ndebug1: Reading configuration data /home/runner/.ssh/
config\r\ndebug1: /home/runner/.ssh/config line 1: Applying options for *\r\n
\ndebug1: Reading configuration data /etc/ssh/ssh_config\r\ndebug3: /etc/ssh/
ssh_config line 52: Including file /etc/ssh/ssh_config.d/05-redhat.conf depth
0\r\ndebug1: Reading configuration data /etc/ssh/ssh_config.d/05-redhat.conf\r
\ndebug2: checking match for 'final all' host servera.lab.exammpole.com originally
servera.lab.exammpole.com\r\ndebug3: /etc/ssh/ssh_config.d/05-redhat.conf
line 3: not matched 'final'\r\ndebug2: match not found\r\ndebug3: /etc/ssh/
ssh_config.d/05-redhat.conf line 5: Including file /etc/crypto-policies/back-
ends/openssl.config depth 1 (parse only)\r\ndebug1: Reading configuration
data /etc/crypto-policies/back-ends/openssl.config\r\ndebug3: gss kex names ok:
[gss-curve25519-sha256-,gss-nistp256-sha256-,gss-group14-sha256-,gss-group16-
sha512-,gss-gex-sha1-,gss-group14-sha1-]\r\ndebug3: kex names ok: [curve25519-
sha256,curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-
sha2-nistp521,diffie-hellman-group-exchange-sha256,diffie-hellman-group14-
sha256,diffie-hellman-group16-sha512,diffie-hellman-group18-sha512,diffie-
hellman-group-exchange-sha1,diffie-hellman-group14-sha1]\r\ndebug1: configuration
requests final Match pass\r\ndebug1: re-parsing configuration\r\ndebug1: Reading
configuration data /home/runner/.ssh/config\r\ndebug1: /home/runner/.ssh/
config line 1: Applying options for *\r\ndebug2: add_identity_file: ignoring
duplicate key ~/.ssh/lab_rsa\r\ndebug1: Reading configuration data /etc/ssh/
ssh_config\r\ndebug3: /etc/ssh/ssh_config line 52: Including file /etc/ssh/
ssh_config.d/05-redhat.conf depth 0\r\ndebug1: Reading configuration data /etc/
ssh/ssh_config.d/05-redhat.conf\r\ndebug2: checking match for 'final all' host
servera.lab.exammpole.com originally servera.lab.exammpole.com\r\ndebug3: /etc/
ssh/ssh_config.d/05-redhat.conf line 3: matched 'final'\r\ndebug2: match found
\r\ndebug3: /etc/ssh/ssh_config.d/05-redhat.conf line 5: Including file /etc/
crypto-policies/back-ends/openssl.config depth 1\r\ndebug1: Reading configuration
data /etc/crypto-policies/back-ends/openssl.config\r\ndebug3: gss kex names ok:
[gss-curve25519-sha256-,gss-nistp256-sha256-,gss-group14-sha256-,gss-group16-
```

Chapter 8 | Troubleshooting Ansible

```
sha512-,gss-gex-sha1-,gss-group14-sha1-]\r\ndebug3: kex names ok: [curve25519-
sha256,curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-
sha2-nistp521,diffie-hellman-group-exchange-sha256,diffie-hellman-group14-
sha256,diffie-hellman-group16-sha512,diffie-hellman-group18-sha512,diffie-hellman-
group-exchange-sha1,diffie-hellman-group14-sha1]\r\ndebug1: auto-mux: Trying
existing master\r\ndebug1: Control socket \"/home/runner/.ansible/cp/d4775f48c9\" does not exist\r\ndebug2: resolving \"servera.lab.exammple.com\" port 22\r\ndebug2: ssh_connect_direct\r\ndebug1: Connecting to servera.lab.exammple.com [3.130.253.23] port 22.\r\ndebug2: fd 3 setting O_NONBLOCK\r\ndebug1: connect to address 3.130.253.23 port 22: Connection timed out\r\ndebug1: Connecting to servera.lab.exammple.com [3.130.204.160] port 22.\r\ndebug2: fd 3 setting O_NONBLOCK\r\ndebug1: connect to address 3.130.204.160 port 22: Connection timed out\r\ndebug1: nssh: connect to host servera.lab.exammple.com port 22: Connection timed out",
"unreachable": true
}

PLAY RECAP ****
servera.lab.exammple.com : ok=0    changed=0    unreachable=1    failed=0
skipped=0    rescued=0    ignored=0
Please review the log for errors.
```

▶ **11.** Investigate the inventory file for errors.

If you look at the `[samba_servers]` group, `servera.lab.example.com` is misspelled (with an extra m). Correct this error as shown below:

```
[samba_servers]
servera.lab.example.com
...output omitted...
```

▶ **12.** Run the playbook again and all tasks should succeed.

```
[student@workstation troubleshoot-host]$ ansible-navigator run \
> -m stdout samba.yml

PLAY [Install a samba server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install samba] ****
changed: [servera.lab.example.com]

TASK [Install firewalld] ****
ok: [servera.lab.example.com]

TASK [Debug install_state variable] ****
ok: [servera.lab.example.com] => {
    "msg": "The state for the samba service is installed"
}

TASK [Start firewalld] ****
ok: [servera.lab.example.com]
```

```
TASK [Configure firewall for samba] ****
changed: [servera.lab.example.com]

TASK [Deliver samba config] ****
changed: [servera.lab.example.com]

TASK [Start samba] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=8    changed=4    unreachable=0    failed=0
skipped=0      rescued=0    ignored=0
```

Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish troubleshoot-host
```

This concludes the section.

► Lab

Troubleshooting Ansible

In this lab, you troubleshoot problems that occur when you try to run a playbook that has been provided to you.

Outcomes

- Troubleshoot playbooks.
- Troubleshoot managed hosts.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start troubleshoot-review
```

This command ensures that Ansible is installed on the `workstation` machine. It also creates the `/home/student/troubleshoot-review/` directory and populates it with the `ansible.cfg`, `inventory`, `index.html`, `secure-web.yml`, and `vhosts.conf` files.

Instructions

In the `/home/student/troubleshoot-review` directory, there is a playbook named `secure-web.yml`. This playbook contains one play that is supposed to set up Apache HTTPD with TLS/SSL for hosts in the `webservers` group. The `serverb.lab.example.com` node is supposed to be the only host in the `webservers` group right now. Ansible can connect to that host using the remote `devops` account and SSH keys that have already been set up. That user can also become `root` on the managed host without a `sudo` password.

Unfortunately, several problems exist that you need to fix before you can run the playbook successfully.

1. From the `/home/student/troubleshoot-review` directory, validate the syntax of the `secure-web.yml` playbook. Fix the issue that is reported.
2. Validate the syntax of the `secure-web.yml` playbook again. It still has a problem. Fix the issue that is reported.
3. Validate the syntax of the `secure-web.yml` playbook again. Another problem is detected. Fix the issue that is reported.
4. Validate the syntax of the `secure-web.yml` playbook a fourth time. It should not show any syntax errors.
5. Run the `secure-web.yml` playbook. Ansible is not able to connect to the `serverb.lab.example.com` host. Two problems prevent a successful connection. Fix both problems.

**Important**

If you resolve these issues without looking at the solution, it is possible that you solve both issues at the same time, or in a different order than shown in the solution.

6. Run the `secure-web.yml` playbook again. The connection to the `serverb.lab.example.com` host works now, but there is a new issue. Fix the issue that is reported.
7. Run the `secure-web.yml` playbook one more time. It should complete successfully. Use an ad hoc command to verify that the `httpd` service is running on the `serverb.lab.example.com` host.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade troubleshoot-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish troubleshoot-review
```

This concludes the section.

► Solution

Troubleshooting Ansible

In this lab, you troubleshoot problems that occur when you try to run a playbook that has been provided to you.

Outcomes

- Troubleshoot playbooks.
- Troubleshoot managed hosts.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start troubleshoot-review
```

This command ensures that Ansible is installed on the `workstation` machine. It also creates the `/home/student/troubleshoot-review/` directory and populates it with the `ansible.cfg`, `inventory`, `index.html`, `secure-web.yml`, and `vhosts.conf` files.

Instructions

In the `/home/student/troubleshoot-review` directory, there is a playbook named `secure-web.yml`. This playbook contains one play that is supposed to set up Apache HTTPD with TLS/SSL for hosts in the `webservers` group. The `serverb.lab.example.com` node is supposed to be the only host in the `webservers` group right now. Ansible can connect to that host using the remote `devops` account and SSH keys that have already been set up. That user can also become `root` on the managed host without a `sudo` password.

Unfortunately, several problems exist that you need to fix before you can run the playbook successfully.

1. From the `/home/student/troubleshoot-review` directory, validate the syntax of the `secure-web.yml` playbook. Fix the issue that is reported.

- 1.1. Change into the `/home/student/troubleshoot-review` directory.

```
[student@workstation ~]$ cd ~/troubleshoot-review/  
[student@workstation troubleshoot-review]$
```

- 1.2. Validate the syntax of the `secure-web.yml` playbook. This playbook sets up Apache HTTPD with TLS/SSL for hosts in the `webservers` group when everything is correct.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \
> -m stdout secure-web.yml --syntax-check
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got
from each:
JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  mapping values are not allowed in this context

The error appears to be in '/home/student/troubleshoot-review/secure-web.yml':
  line 7, column 30, but may be elsewhere in the file depending on the exact syntax
problem.

The offending line appears to be:

vars:
  random_var: This is colon: test
          ^ here
```

- 1.3. In the value for a variable, colons need to be protected by quoting the string. Correct the syntax issue in the definition of the `random_var` variable by adding double quotation marks to the `This is colon: test` string. The resulting change should appear as follows:

```
...output omitted...
vars:
  random_var: "This is colon: test"
...output omitted...
```

2. Validate the syntax of the `secure-web.yml` playbook again. It still has a problem. Fix the issue that is reported.

 - 2.1. Validate the syntax of `secure-web.yml` using `ansible-navigator run -m stdout --syntax-check` again.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \
> -m stdout secure-web.yml --syntax-check
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got
from each:
JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  did not find expected '-' indicator

The error appears to be in '/home/student/troubleshoot-review/secure-web.yml':
  line 38, column 10, but may be elsewhere in the file depending on the exact
syntax problem.

The offending line appears to be:
```

```
- name: Start and enable web services
^ here
```

- 2.2. Correct any syntax issues in the indentation. Remove the extra space at the beginning of the *start and enable web services* task elements. The resulting change should appear as follows:

```
...output omitted...
args:
  creates: /etc/pki/tls/certs/serverb.lab.example.com.crt

  - name: Start and enable web services
    ansible.builtin.service:
      name: httpd
      state: started
      enabled: yes

  - name: Deliver content
    ansible.builtin.copy:
      dest: /var/www/vhosts/serverb-secure
      src: html/
...output omitted...
```

3. Validate the syntax of the `secure-web.yml` playbook again. Another problem is detected. Fix the issue that is reported.

- 3.1. Validate the syntax of the `secure-web.yml` playbook.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \
> -m stdout secure-web.yml --syntax-check
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got
from each:
JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  found unacceptable key (unhashable type: 'AnsibleMapping')

The error appears to be in '/home/student/troubleshoot-review/secure-web.yml':
  line 13, column 20, but may
  be elsewhere in the file depending on the exact syntax problem.
```

The offending line appears to be:

```
ansible.builtin.dnf:
  name: {{ item }}
  ^ 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 }}"
```

- 3.2. Correct the `item` variable in the `install web server packages` task. A value must be protected by double quotation marks if braces appear at the start of the value. Add double quotation marks to `{{ item }}`. The resulting change should appear as follows:

```
...output omitted...  
  - name: Install web server packages  
    ansible.builtin.dnf:  
      name: "{{ item }}"  
      state: latest  
      notify:  
        - Restart services  
      loop:  
        - httpd  
        - mod_ssl  
...output omitted...
```

4. Validate the syntax of the `secure-web.yml` playbook a fourth time. It should not show any syntax errors.

- 4.1. Review the syntax of the `secure-web.yml` playbook. It should not show any syntax errors.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \  
> -m stdout secure-web.yml --syntax-check  
playbook: /home/student/troubleshoot-review/secure-web.yml
```

5. Run the `secure-web.yml` playbook. Ansible is not able to connect to the `serverb.lab.example.com` host. Two problems prevent a successful connection. Fix both problems.



Important

If you resolve these issues without looking at the solution, it is possible that you solve both issues at the same time, or in a different order than shown in the solution.

- 5.1. Run the `secure-web.yml` playbook. This fails with an error.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \  
> -m stdout secure-web.yml  
  
PLAY [Create secure web service] ****  
  
TASK [Gathering Facts] ****  
fatal: [serverb.lab.example.com]: UNREACHABLE! => {"changed": false, "msg":  
  "Failed to connect to the host via ssh: students@serverc.lab.example.com:  
  Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).",  
  "unreachable": true}
```

```
PLAY RECAP ****
serverb.lab.example.com : ok=0    changed=0    unreachable=1    failed=0
skipped=0   rescued=0   ignored=0
Please review the log for errors.
```

For some reason, when `ansible-navigator` tried to connect to the `serverb.lab.example.com` host, it instead attempted to connect to the `serverc.lab.example.com` host as the `students` user.

- Run the `secure-web.yml` playbook again, adding the `-vvv` parameter to increase the verbosity of the debug output.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \
> -m stdout secure-web.yml -vvv
...output omitted...
TASK [Gathering Facts] ****
task path: /home/student/troubleshoot-review/secure-web.yml:3
<serverc.lab.example.com> ESTABLISH SSH CONNECTION FOR USER: students
<serverc.lab.example.com> SSH: EXEC ssh -C -o ControlMaster=auto -o
ControlPersist=60s -o StrictHostKeyChecking=no -o KbdInteractiveAuthentication=no
-o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey
-o PasswordAuthentication=no -o 'User="students"' -o ConnectTimeout=10 -o
'ControlPath="/home/runner/.ansible/cp/bc0c05136a"' serverc.lab.example.com '/
bin/sh -c """echo ~students && sleep 0"""
<serverc.lab.example.com> (255, b'', b'students@serverc.lab.example.com:
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).\r\n')
...output omitted...
```

You can identify two problems from the verbose debug output.

- The `ansible-navigator` command is attempting to connect as the `students` user, when it should be using `devops` as the user.
- The `ansible-navigator` command is attempting to connect to the `serverc.lab.example.com` host instead of the `serverb.lab.example.com` host.

- Inspect the `inventory` file. It sets an `ansible_host` host variable that causes connections for the `serverb.lab.example.com` managed host to be incorrectly directed to the `serverc.lab.example.com` managed host.

Delete the `ansible_host` host variable so that the file has the following contents:

```
[webservers]
serverb.lab.example.com
```

- Edit the `secure-web.yml` playbook to ensure that `devops` is the `remote_user` for the play. The first lines of the playbook should appear as follows:

```
---
# start of secure web server playbook
- name: Create secure web service
  hosts: webservers
  remote_user: devops
...output omitted...
```

6. Run the `secure-web.yml` playbook again. The connection to the `serverb.lab.example.com` host works now, but there is a new issue. Fix the issue that is reported.
 - 6.1. Run the `secure-web.yml` playbook, adding the `-vvv` parameter to increase the verbosity of the debug output.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \
> -m stdout secure-web.yml -vvv
...output omitted...
failed: [serverb.lab.example.com] (item=mod_ssl) => {
    "ansible_loop_var": "item",
    "changed": false,
...output omitted...
},
"item": "mod_ssl",
"msg": "This command has to be run under the root user.",
"results": []
}
...output omitted...
```

The play is not being run with privilege escalation.

- 6.2. Edit the play to make sure that it has `become: true` set. The resulting change should appear as follows:

```
---
# start of secure web server playbook
- name: Create secure web service
  hosts: webservers
  remote_user: devops
  become: true
...output omitted...
```

7. Run the `secure-web.yml` playbook one more time. It should complete successfully. Use an ad hoc command to verify that the `httpd` service is running on the `serverb.lab.example.com` host.

- 7.1. Run the `secure-web.yml` playbook.

```
[student@workstation troubleshoot-review]$ ansible-navigator run \
> -m stdout secure-web.yml

PLAY [Create secure web service] ****
...output omitted...

TASK [Install web server packages] ****
ok: [serverb.lab.example.com] => (item=httpd)
ok: [serverb.lab.example.com] => (item=mod_ssl)
...output omitted...

TASK [Httpd_conf_syntax variable] ****
ok: [serverb.lab.example.com] => {
```

```
"msg": "The httpd_conf_syntax variable value is {'changed': True, 'stdout': '', 'stderr': 'Syntax OK', 'rc': 0, 'cmd': ['/sbin/httpd', '-t'], 'start': '2022-07-14 17:39:51.096013', 'end': '2022-07-14 17:39:51.134925', 'delta': '0:00:00.038912', 'msg': '', 'stdout_lines': [], 'stderr_lines': ['Syntax OK'], 'failed': False, 'failed_when_result': False}"  
}  
...output omitted...  
  
RUNNING HANDLER [Restart services] ****  
changed: [serverb.lab.example.com]  
  
PLAY RECAP ****  
serverb.lab.example.com : ok=11    changed=7    unreachable=0    failed=0  
skipped=0    rescued=0    ignored=0
```

- 7.2. Use an ad hoc command to determine the state of the `httpd` service on managed hosts in the `webservers` host group. The `httpd` service should now be running on the `serverb.lab.example.com` host.

```
[student@workstation troubleshoot-review]$ ansible webservers -u devops -b \  
> -m command -a 'systemctl status httpd'  
serverb.lab.example.com | CHANGED | rc=0 >>  
● httpd.service - The Apache HTTP Server  
    Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor  
    preset: disabled)  
    Active: active (running) since Thu 2022-07-14 17:39:53 EDT; 3min 11s  
...output omitted...
```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade troubleshoot-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish troubleshoot-review
```

This concludes the section.

Summary

- The `ansible-navigator` command can produce playbook artifact files that store information about runs in JSON format.
- Use the `ansible-navigator replay` command to review play execution.
- The `ansible.builtin.debug` module can provide additional debugging information when you run a playbook (for example, the current value for a variable).
- You can specify the `-v` option of the `ansible-navigator run` command one or more times to provide several levels of additional output verbosity. This is useful for debugging Ansible tasks when running a playbook.
- The `--check` option enables Ansible modules that support check mode to display the changes to be performed, instead of applying those changes to the managed hosts.

Chapter 9

Automating Linux Administration Tasks

Goal

Automate common Linux system administration tasks with Ansible.

Objectives

- Subscribe systems, configure software channels and repositories, enable module streams, and manage RPM packages on managed hosts.
- Manage Linux users and groups, configure SSH, and modify Sudo configuration on managed hosts.
- Manage service startup, schedule processes with `at`, `cron`, and `systemd`, reboot managed hosts with `reboot`, and control the default boot target on managed hosts.
- Partition storage devices, configure LVM, format partitions or logical volumes, mount file systems, and add swap spaces.
- Configure network settings and name resolution on managed hosts, and collect network-related Ansible facts.

Sections

- Managing Software and Subscriptions (and Guided Exercise)
- Managing Users and Authentication (and Guided Exercise)
- Managing the Boot Process and Scheduled Processes (and Guided Exercise)
- Managing Storage (and Guided Exercise)
- Managing Network Configuration (and Guided Exercise)

Lab

- Automating Linux Administration Tasks

Managing Software and Subscriptions

Objectives

- Subscribe systems, configure software channels and repositories, enable module streams, and manage RPM packages on managed hosts.

Managing Packages with Ansible

The `ansible.builtin.dnf` Ansible module uses `dnf` on the managed hosts to handle package operations. The following example is a playbook that installs the `httpd` package on the `servera.lab.example.com` managed host.

```
---
- name: Install the required packages on the web server
  hosts: servera.lab.example.com
  tasks:
    - name: Install the httpd packages
      ansible.builtin.dnf:
        name: httpd      ①
        state: present  ②
```

- ① The `name` keyword specifies the name of the package to install.
- ② The `state` keyword indicates the expected state of the package on the managed host:

`present`

Ansible installs the package if it is not already installed.

`absent`

Ansible removes the package if it is installed.

`latest`

Ansible updates the package if it is not already at the most recent available version. If the package is not installed, Ansible installs it.

The following table compares some uses of the `ansible.builtin.dnf` Ansible module with the equivalent `dnf` command.

Ansible task	DNF command
<pre>- name: Install httpd ansible.builtin.dnf: name: httpd state: present</pre>	<code>dnf install httpd</code>

Ansible task	DNF command
<pre>- name: Install or upgrade httpd ansible.builtin.dnf: name: httpd state: latest</pre>	dnf upgrade httpd or dnf install httpd if the package is not yet installed.
<pre>- name: Upgrade all packages ansible.builtin.dnf: name: '*' state: latest</pre>	dnf upgrade
<pre>- name: Remove httpd ansible.builtin.dnf: name: httpd state: absent</pre>	dnf remove httpd
<pre>- name: Install Development Tools ansible.builtin.dnf: name: '@Development Tools' ① state: present</pre> <p>① With the <code>ansible.builtin.dnf</code> Ansible module, you must prefix group names with the @ character. Remember that you can retrieve the list of groups with the <code>dnf group list</code> command.</p>	dnf group install "Development Tools"
<pre>- name: Remove Development Tools ansible.builtin.dnf: name: '@Development Tools' state: absent</pre>	dnf group remove "Development Tools"
<pre>- name: Install perl DNF module ansible.builtin.dnf: name: '@perl:5.26/minimal' ① state: present</pre> <p>① To manage a package module, prefix its name with the @ character. The syntax is the same as with the <code>dnf</code> command. For example, you can omit the profile part to use the default profile: <code>@perl:5.26</code>. Remember that you can list the available package modules with the <code>dnf module list</code> command.</p>	dnf module install perl:5.26/minimal

**Important**

Red Hat Enterprise Linux 8 provides several package modules, but Red Hat Enterprise Linux 9.0 did not when it was initially released. Package modules for additional versions of software components are expected to become available in future minor releases of Red Hat Enterprise Linux.

Run the `ansible-navigator doc ansible.builtin.dnf` command for additional parameters and playbook examples.

Optimizing Multiple Package Installation

To operate on several packages, the `name` keyword accepts a list. The following example shows a playbook that installs three packages on `servera.lab.example.com`.

```
---
- name: Install the required packages on the web server
  hosts: servera.lab.example.com
  tasks:
    - name: Install the packages
      ansible.builtin.dnf:
        name:
          - httpd
          - mod_ssl
          - httpd-tools
        state: present
```

With this syntax, Ansible installs the packages in a single DNF transaction. This is equivalent to running the `dnf install httpd mod_ssl httpd-tools` command.

A commonly seen but less efficient and slower version of this task is to use a loop.

```
---
- name: Install the required packages on the web server
  hosts: servera.lab.example.com
  tasks:
    - name: Install the packages
      ansible.builtin.dnf:
        name: "{{ item }}"
        state: present
      loop:
        - httpd
        - mod_ssl
        - httpd-tools
```

Avoid using this method because it requires the module to perform three individual transactions; one for each package.

Gathering Facts about Installed Packages

The `ansible.builtin.package_facts` Ansible module collects the installed package details on managed hosts. It sets the `ansible_facts['packages']` variable with the package details.

The following playbook calls the `ansible.builtin.package_facts` module, and the `ansible.builtin.debug` module to display the content of the `ansible_facts['packages']` variable and the version of the installed NetworkManager package.

```
---
- name: Display installed packages
  hosts: servera.lab.example.com
  tasks:
    - name: Gather info on installed packages
      ansible.builtin.package_facts:
        manager: auto

    - name: List installed packages
      ansible.builtin.debug:
        var: ansible_facts['packages']

    - name: Display NetworkManager version
      ansible.builtin.debug:
        msg: "Version {{ ansible_facts['packages']['NetworkManager'][0]['version'] }}"
        when: "'NetworkManager' in ansible_facts['packages']"
```

When run, the playbook displays the package list and the version of the NetworkManager package:

```
[user@controlnode ~]$ ansible-navigator run -m stdout lspackages.yml

PLAY [Display installed packages] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Gather info on installed packages] ****
ok: [servera.lab.example.com]

TASK [List installed packages] ****
ok: [servera.lab.example.com] => {
  "ansible_facts['packages']": [
    "ModemManager": [
      {
        "arch": "x86_64",
        "epoch": null,
        "name": "ModemManager",
        "release": "3.el9",
        "source": "rpm",
        "version": "1.18.2"
      }
    ],
    ...output omitted...
    "zstd": [
      {
        "arch": "x86_64",
        "epoch": null,
```

```

        "name": "zstd",
        "release": "2.el9",
        "source": "rpm",
        "version": "1.5.1"
    }
]
}
}

TASK [Display NetworkManager version] ****
ok: [servera.lab.example.com] => {
    "msg": "Version 1.36.0"
}

PLAY RECAP ****
servera.lab.example.com : ok=4    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0

```

Reviewing Alternative Modules to Manage Packages

For other package managers, Ansible usually provides a dedicated module. The `ansible.builtin.apt` module uses the APT package tool available on Debian and Ubuntu. The `ansible.windows.win_package` module can install software on Microsoft Windows systems.

The following playbook uses conditionals to select the appropriate module in an environment composed of Red Hat Enterprise Linux systems running major versions 7, 8, and 9.

```

---
- name: Install the required packages on the web servers
  hosts: webservers
  tasks:
    - name: Install httpd on RHEL 8 and 9
      ansible.builtin.dnf:
        name: httpd
        state: present
      when:
        - "ansible_facts['distribution'] == 'RedHat'"
        - "ansible_facts['distribution_major_version'] >= '8'"

    - name: Install httpd on RHEL 7 and earlier
      ansible.builtin.yum:
        name: httpd
        state: present
      when:
        - "ansible_facts['distribution'] == 'RedHat'"
        - "ansible_facts['distribution_major_version'] <= '7'"

```

As an alternative, the generic `ansible.builtin.package` module automatically detects and uses the package manager available on the managed hosts. With the `ansible.builtin.package` module, you can rewrite the previous playbook as follows:

```
---
- name: Install the required packages on the web servers
  hosts: webservers
  tasks:
    - name: Install httpd
      ansible.builtin.package:
        name: httpd
        state: present
```

However, the `ansible.builtin.package` module does not support all the features that the more specialized modules provide.

Also, operating systems often have different names for the packages they provide. For example, the package that installs the Apache HTTP Server is `httpd` on Red Hat Enterprise Linux and `apache2` on Ubuntu.

In that situation, you still need a conditional for selecting the correct package name depending on the operating system of the managed host.

Registering and Managing Systems with Red Hat Subscription Management

To entitle your Red Hat Enterprise Linux systems to product subscriptions, one option is to use the `community.general.redhat_subscription` and `community.general.rhsm_repository` modules. These modules interface with the Red Hat Subscription Management tool on the managed hosts.



Important

The Subscription Management modules previously described are part of the `community.general` Ansible Content Collection, and are not supported by Red Hat at this time.

As an alternative, you could use the `ansible.builtin.command` module to call the command-line Subscription Management commands. To avoid accidental reregistration and allow the playbook to be idempotent, you need to determine what condition to use to skip subscription-related tasks that have already been completed.

Registering and Subscribing New Systems

The first two tasks you usually perform with the Red Hat Subscription Management tool is to register the new system and attach an available subscription.

Without Ansible, you perform these tasks with the `subscription-manager` command:

```
[user@host ~]$ subscription-manager register --username=yourusername \
> --password=yourpassword
[user@host ~]$ subscription-manager attach --pool=poolID
```

Remember that you can list the pools available to your account with the `subscription-manager list --available` command.

The `community.general.redhat_subscription` module performs the registration and the subscription in one task.

```
- name: Register and subscribe the system
  community.general.redhat_subscription:
    username: yourusername
    password: yourpassword
    pool_ids: poolID
    state: present
```

A `state` keyword set to `present` indicates to register and to subscribe the system. When it is set to `absent`, the module unregisters the system.

Enabling Red Hat Software Repositories

The next task after the subscription is to enable Red Hat software repositories on the new system.

Without Ansible, you would execute the `subscription-manager` command for that purpose:

```
[user@host ~]$ subscription-manager repos \
> --enable "rhel-9-for-x86_64-baseos-rpms" \
> --enable "rhel-9-for-x86_64-appstream-rpms"
```

Remember that you can list the available repositories with the `subscription-manager repos --list` command.

With Ansible, you can use the `community.general.rhsm_repository` module:

```
- name: Enable Red Hat repositories
  community.general.rhsm_repository:
    name:
      - rhel-9-for-x86_64-baseos-rpms
      - rhel-9-for-x86_64-appstream-rpms
    state: present
```

Configuring an RPM Package Repository

To enable support for a third-party Yum repository on a managed host, Ansible provides the `ansible.builtin.yum_repository` module.

Declaring an RPM Package Repository

When run, the following playbook declares a new Yum repository on `servera.lab.example.com`.

```
---
- name: Configure the company Yum/DNF repositories
  hosts: servera.lab.example.com
  tasks:
    - name: Ensure Example Repo exists
      ansible.builtin.yum_repository:
        file: example ①
        name: example-internal
```

```
description: Example Inc. Internal YUM/DNF repo
baseurl: http://materials.example.com/yum/repository/
enabled: yes
gpgcheck: yes ②
state: present ③
```

- ① The `file` keyword specifies the name of the file to create under the `/etc/yum.repos.d/` directory. The module automatically adds the `.repo` extension to that name.
- ② Typically, software providers digitally sign RPM packages using GPG keys. By setting the `gpgcheck` keyword to `yes`, the RPM system verifies package integrity by confirming that the package was signed by the appropriate GPG key. It refuses to install a package if the GPG signature does not match. Use the `ansible.builtin.rpm_key` Ansible module, described later in this section, to install the required GPG public key.
- ③ When you set the `state` keyword to `present`, Ansible creates or updates the `.repo` file. When `state` is set to `absent`, Ansible deletes the file.

The resulting `/etc/yum.repos.d/example.repo` file on `servera.lab.example.com` is as follows:

```
[example-internal]
baseurl = http://materials.example.com/yum/repository/
enabled = 1
gpgcheck = 1
name = Example Inc. Internal YUM repo
```

The `ansible.builtin.yum_repository` module exposes most of the repository configuration parameters as keywords. Run the `ansible-navigator doc ansible.builtin.yum_repository` command for additional parameters and playbook examples.



Note

Some third-party repositories provide the configuration file and the GPG public key as part of an RPM package that can be downloaded and installed using the `dnf install` command.

For example, the *Extra Packages for Enterprise Linux (EPEL)* project provides the `https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm` package that deploys the `/etc/yum.repos.d/epel.repo` configuration file.

For this repository, use the `ansible.builtin.dnf` module to install the EPEL package instead of the `ansible.builtin.yum_repository` module.

Importing an RPM GPG Key

When the `gpgcheck` keyword is set to `yes` in the `ansible.builtin.yum_repository` module, you also need to install the GPG key on the managed host. The `ansible.builtin.rpm_key` module in the following example deploys the GPG public key hosted on a remote web server to the `servera.lab.example.com` managed host.

```
---
- name: Configure the company Yum/DNF repositories
  hosts: servera.lab.example.com
  tasks:
    - name: Deploy the GPG public key
      ansible.builtin.rpm_key:
        key: http://materials.example.com/yum/repository/RPM-GPG-KEY-example
        state: present

    - name: Ensure Example Repo exists
      ansible.builtin.yum_repository:
        file: example
        name: example-internal
        description: Example Inc. Internal YUM/DNF repo
        baseurl: http://materials.example.com/yum/repository/
        enabled: yes
        gpgcheck: yes
        state: present
```



References

`dnf(8)`, `yum.conf(5)`, and `subscription-manager(8)` man pages

`ansible.builtin.dnf module – Manages packages with the dnf package`

manager – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/dnf_module.html

`ansible.builtin.package_facts module – Package information as facts – Ansible Documentation`

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/package_facts_module.html

`community.general.redhat_subscription module – Manage registration and subscriptions to RHSM using the subscription-manager command – Ansible Documentation`

https://docs.ansible.com/ansible/latest/collections/community/general/redhat_subscription_module.html

`community.general.rhsm_repository module – Manage RHSM repositories using the subscription-manager command – Ansible Documentation`

https://docs.ansible.com/ansible/latest/collections/community/general/rhsm_repository_module.html

`ansible.builtin.yum_repository module – Add or remove YUM repositories – Ansible Documentation`

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/yum_repository_module.html

`ansible.builtin.rpm_key module – Adds or removes a gpg key from the rpm db – Ansible Documentation`

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/rpm_key_module.html

► Guided Exercise

Managing Software and Subscriptions

In this exercise, you configure a new Yum repository and install packages from it on your managed hosts.

Outcomes

- Configure a Yum repository using the `ansible.builtin.yum_repository` module.
- Manage RPM GPG keys using the `ansible.builtin.rpm_key` module.
- Obtain information about the installed packages on a host using the `ansible.builtin.package_facts` module.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start system-software
```

Instructions

Your organization requires that the `simple-agent` RPM package be installed on all hosts. This package is provided by an internal Yum repository maintained by your organization to host its internally developed software packages.

You need to write a playbook to ensure that the `simple-agent` package is installed on all managed hosts. The playbook must also ensure that all managed hosts are configured to use the internal Yum repository.

The repository is located at `http://materials.example.com/yum/repository`. All RPM packages in the repository are signed with a GPG key pair. The GPG public key for the repository packages is available at `http://materials.example.com/yum/repository/RPM-GPG-KEY-example`.

- 1. Change to the `/home/student/system-software` directory.

```
[student@workstation ~]$ cd ~/system-software  
[student@workstation system-software]$
```

- 2. Begin writing the `repo_playbook.yml` playbook. Define a single play in the playbook that targets all hosts. Add a `vars` clause to the play that defines a single variable, `custom_pkg`, with the value `simple-agent` (the name of the RPM package that needs to be installed everywhere.) Add an empty `tasks` clause to the play.

The playbook should consist of the following content:

```
---
- name: Repository Configuration
  hosts: all
  vars:
    custom_pkg: simple-agent
  tasks:
```

- 3. Add two tasks to the `tasks` clause of the play in the `repo_playbook.yml` file.

Use the `ansible.builtin.package_facts` module in the first task to gather information about installed packages on the managed hosts. This task populates the `ansible_facts.packages` fact.

Use the `ansible.builtin.debug` module in the second task to print the installed version of the package referenced by the `custom_pkg` variable. Only run this task if the `custom` package is found in the `ansible_facts.packages` fact.

Run the `repo_playbook.yml` playbook.

- 3.1. Add the first task to the play. Set the value of the `manager` keyword to `auto` for the `ansible.builtin.package_facts` module.

The first task should consist of the following content:

```
- name: Gather Package Facts
  ansible.builtin.package_facts:
    manager: auto
```

- 3.2. Add a second task to the play that uses the `ansible.builtin.debug` module to display the value of the `ansible_facts.packages[custom_pkg]` variable. Add a `when` clause to the task to verify that the value of the `custom_pkg` variable is contained in the `ansible_facts['packages']` variable.

The second task should consist of the following content:

```
- name: Show Package Facts for the custom package
  ansible.builtin.debug:
    var: ansible_facts['packages'][custom_pkg]
    when: custom_pkg in ansible_facts['packages']
```

- 3.3. Run the playbook:

```
[student@workstation system-software]$ ansible-navigator run \
> -m stdout repo_playbook.yml

PLAY [Repository Configuration] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Gather Package Facts] ****
ok: [servera.lab.example.com]

TASK [Show Package Facts for the custom package] ****
skipping: [servera.lab.example.com]
```

```
PLAY RECAP ****
servera.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

The Show Package Facts for the custom package task is skipped because the simple-agent package is not installed on the managed hosts.

- 4. Add a third task to the play that uses the `ansible.builtin.yum_repository` module to ensure the internal Yum repository is configured on the managed hosts. This task has the following requirements:

- The repository configuration is stored in the file `/etc/yum.repos.d/example.repo`
- The repository ID is `example-internal`
- The repository base URL is `http://materials.example.com/yum/repository`
- The repository is configured to check RPM GPG signatures
- The repository description is Example Inc. Internal YUM repo

The third task should consist of the following content:

```
- name: Ensure Example Repo exists
  ansible.builtin.yum_repository:
    name: example-internal
    description: Example Inc. Internal YUM repo
    file: example
    baseurl: http://materials.example.com/yum/repository/
    gpgcheck: yes
```

- 5. Add a fourth task to the play that uses the `ansible.builtin.rpm_key` module to ensure that the repository's public key is present on the managed hosts. The repository's public key is available at `http://materials.example.com/yum/repository/RPM-GPG-KEY-example`.

The fourth task should consist of the following content:

```
- name: Ensure Repo RPM Key is Installed
  ansible.builtin.rpm_key:
    key: http://materials.example.com/yum/repository/RPM-GPG-KEY-example
    state: present
```

- 6. Add a fifth task to the play that ensures that the package referenced by the `custom_pkg` variable is installed on the managed hosts.

The fifth task should consist of the following content:

```
- name: Install Example package
  ansible.builtin.dnf:
    name: "{{ custom_pkg }}"
    state: present
```

- 7. The `ansible_facts['packages']` fact is not automatically updated when a new package is installed on a managed host. This step demonstrates that this is true.

Copy the second task and add it as the sixth task in the play. Run the playbook and verify that the `ansible_facts['packages']` fact does not contain information about the `simple-agent` package installed on the managed hosts.

7.1. The sixth task contains a copy of the second task:

```
- name: Show Package Facts for the custom package
  ansible.builtin.debug:
    var: ansible_facts['packages'][custom_pkg]
  when: custom_pkg in ansible_facts['packages']
```

The entire `repo_playbook.yml` playbook should now consist of the following content:

```
---
- name: Repository Configuration
  hosts: all
  vars:
    custom_pkg: simple-agent
  tasks:
    - name: Gather Package Facts
      ansible.builtin.package_facts:
        manager: auto

    - name: Show Package Facts for the custom package
      ansible.builtin.debug:
        var: ansible_facts['packages'][custom_pkg]
      when: custom_pkg in ansible_facts['packages']

    - name: Ensure Example Repo exists
      ansible.builtin.yum_repository:
        name: example-internal
        description: Example Inc. Internal YUM repo
        file: example
        baseurl: http://materials.example.com/yum/repository/
        gpgcheck: yes

    - name: Ensure Repo RPM Key is Installed
      ansible.builtin.rpm_key:
        key: http://materials.example.com/yum/repository/RPM-GPG-KEY-example
        state: present

    - name: Install Example package
      ansible.builtin.dnf:
        name: "{{ custom_pkg }}"
        state: present

    - name: Show Package Facts for the custom package
      ansible.builtin.debug:
        var: ansible_facts['packages'][custom_pkg]
      when: custom_pkg in ansible_facts['packages']
```

7.2. Run the playbook.

```
[student@workstation system-software]$ ansible-navigator run \
> -m stdout repo_playbook.yml

PLAY [Repository Configuration] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Gather Package Facts] ****
ok: [servera.lab.example.com] ①

TASK [Show Package Facts for the custom package] ****
skipping: [servera.lab.example.com]

TASK [Ensure Example Repo exists] ****
changed: [servera.lab.example.com]

TASK [Ensure Repo RPM Key is Installed] ****
changed: [servera.lab.example.com]

TASK [Install Example package] ****
changed: [servera.lab.example.com]

TASK [Show Package Facts for the custom package] ****
skipping: [servera.lab.example.com] ②

PLAY RECAP ****
servera.lab.example.com      : ok=5    changed=3    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
```

- ① The Gather Package Facts task determines the data contained in the `ansible_facts['packages']` fact.
 - ② The task is skipped because the `simple-agent` package is installed after the Gather Package Facts task.
- ▶ 8. Update the package facts in your play by inserting a task immediately after the `Install Example package` task. Write the new task so that it runs the `ansible.builtin.package_facts` module. Set the module's `manager` attribute to `auto`.

The complete playbook should consist of the following content:

```
---
- name: Repository Configuration
  hosts: all
  vars:
    custom_pkg: simple-agent
  tasks:
    - name: Gather Package Facts
      ansible.builtin.package_facts:
        manager: auto
```

```

- name: Show Package Facts for the custom package
  ansible.builtin.debug:
    var: ansible_facts['packages'][custom_pkg]
  when: custom_pkg in ansible_facts['packages']

- name: Ensure Example Repo exists
  ansible.builtin.yum_repository:
    name: example-internal
    description: Example Inc. Internal YUM repo
    file: example
    baseurl: http://materials.example.com/yum/repository/
    gpgcheck: yes

- name: Ensure Repo RPM Key is Installed
  ansible.builtin.rpm_key:
    key: http://materials.example.com/yum/repository/RPM-GPG-KEY-example
    state: present

- name: Install Example package
  ansible.builtin.dnf:
    name: "{{ custom_pkg }}"
    state: present

- name: Gather Package Facts
  ansible.builtin.package_facts:
    manager: auto

- name: Show Package Facts for the custom package
  ansible.builtin.debug:
    var: ansible_facts['packages'][custom_pkg]
  when: custom_pkg in ansible_facts['packages']

```

- ▶ 9. Use an Ansible ad hoc command to remove the `simple-agent` package installed during the previous execution of the playbook. Run the playbook with the inserted `ansible.builtin.package_facts` task and use the output to verify the installation of the `simple-agent` package.

- 9.1. To remove the `simple-agent` package from all hosts, use the `ansible all` command with the `-m ansible.builtin.dnf` and `-a 'name=simple-agent state=absent'` options.

```
[student@workstation system-software]$ ansible all -m ansible.builtin.dnf \
> -a 'name=simple-agent state=absent'
servera.lab.example.com | CHANGED => {
...output omitted...
  "changed": true,
  "msg": "",
  "rc": 0,
  "results": [
    "Removed: simple-agent-1.0-1.el9.x86_64"
  ]
...output omitted...
```

- 9.2. Run the `repo_playbook.yml` playbook.

```
[student@workstation system-software]$ ansible-navigator run \
> -m stdout repo_playbook.yml

PLAY [Repository Configuration] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Gather Package Facts] ****
ok: [servera.lab.example.com]

TASK [Show Package Facts for the custom package] ****
skipping: [servera.lab.example.com] ①

...output omitted...

TASK [Install Example package] ****
changed: [servera.lab.example.com] ②

TASK [Gather Package Facts] ****
ok: [servera.lab.example.com] ③

TASK [Show Package Facts for the custom package] ****
ok: [servera.lab.example.com] => {
    "ansible_facts['packages'][custom_pkg)": [④
        {
            "arch": "x86_64",
            "epoch": null,
            "name": "simple-agent",
            "release": "1.el9",
            "source": "rpm",
            "version": "1.0"
        }
    ]
}

PLAY RECAP ****
servera.lab.example.com : ok=7    changed=1    unreachable=0    failed=0
skipped=1    rescued=0    ignored=0
```

- ① No package fact exists for the `simple-agent` package because the package is not installed on the managed hosts.
- ② The `simple-agent` package is installed as a result of this task, as indicated by the `changed` status.
- ③ This task updates the package facts with information about the `simple-agent` package.
- ④ The `simple-agent` package fact exists and indicates only one `simple-agent` package is installed. The installed package is version `1.0`.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-software
```

This concludes the section.

Managing Users and Authentication

Objectives

- Manage Linux users and groups, configure SSH, and modify the Sudo configuration on managed hosts.

The User Module

The Ansible `ansible.builtin.user` module lets you create, configure, and remove user accounts on managed hosts. You can remove or add a user, set a user's home directory, set the UID for system user accounts, manage passwords, and assign a user to supplementary groups. To create a user that can log in to the machine, you need to provide a hashed password for the `password` parameter. See "How do I generate encrypted passwords for the user module?" [https://docs.ansible.com/ansible/latest/reference_appendices/faq.html#how-do-i-generate-encrypted-passwords-for-the-user-module] for information on how to hash a password.

The following example demonstrates the `ansible.builtin.user` module:

```
- name: Create devops_user if missing, make sure is member of correct groups
  ansible.builtin.user:
    name: devops_user ①
    shell: /bin/bash ②
    groups: sys_admins, developers ③
    append: true
```

- The `name` parameter is the only option required by the `ansible.builtin.user` module. Its value is the name of the service account or user account to create, remove, or modify.
- The `shell` parameter sets the user's shell.
- The `groups` parameter, when used with the `append` parameter, tells the machine to append the supplementary groups `sys_admins` and `developers` to this user. If you do not use the `append` parameter then the groups provided overwrite a user's existing supplementary groups. To set the primary group for a user, use the `group` option.



Note

The `ansible.builtin.user` module also provides information in return values, such as the user's home directory and a list of groups that the user is a member of. These return values can be registered into a variable and used in subsequent tasks. More information is available in the documentation for the module.

Commonly Used Parameters for the User Module

Parameter	Comments
<code>comment</code>	Optionally sets the description of a user account.

Parameter	Comments
group	Optionally sets the user's primary group.
groups	Optionally sets a list of supplementary groups for the user. When set to a null value, all groups except the primary group are removed.
home	Optionally sets the user's home directory location.
create_home	Optionally takes a Boolean value of <code>true</code> or <code>false</code> . A home directory is created for the user if the value is set to <code>true</code> .
system	Optionally takes a Boolean value of <code>true</code> or <code>false</code> . When creating an account, this makes the user a system account if the value is set to <code>true</code> . This setting cannot be changed on existing users.
uid	Sets the UID number of the user.
state	If set to <code>present</code> , create the account if it is missing (the default setting). If set to <code>absent</code> , remove the account if it is present.

Use the User Module to Generate an SSH Key

The `ansible.builtin.user` module can generate an SSH key if called with the `generate_ssh_key` parameter.

The following example demonstrates how the `ansible.builtin.user` module generates an SSH key:

```
- name: Create an SSH key for user1
  ansible.builtin.user:
    name: user1
    generate_ssh_key: true ①
    ssh_key_bits: 2048 ②
    ssh_key_file: .ssh/id_my_rsa ③
```

- ① The `generate_ssh_key` parameter accepts a Boolean value that specifies whether to generate an SSH key for the user. This does not overwrite an existing SSH key unless the `force` parameter is provided with the `true` value.
- ② The `ssh_key_bits` parameter sets the number of bits in the new SSH key.
- ③ The `ssh_key_file` parameter specifies the file name for the new SSH private key (the public key adds the `.pub` suffix).

The Group Module

The `ansible.builtin.group` module adds, deletes, and modifies groups on the managed hosts. The managed hosts need to have the `groupadd`, `groupdel`, and `groupmod` commands available, which are provided by the `shadow-utils` package in Red Hat Enterprise Linux 9. For Microsoft Windows managed hosts, use the `win_group` module.

The following example demonstrates how the `ansible.builtin.group` module creates a group:

```
- name: Verify that the auditors group exists
  ansible.builtin.group:
    name: auditors ①
    state: present ②
```

- ① The `name` parameter is the only required option for the `ansible.builtin.group` module. The value is the name of the group to manage.
- ② The `state` parameter accepts the value `absent` or `present`, which removes or creates the group, respectively. The default value for the `state` parameter is `present`.

Parameters for the Group Module

Parameter	Comments
<code>gid</code>	This parameter sets the GID number to for the group. If omitted, the number is automatically selected.
<code>local</code>	This parameter forces the use of local command alternatives (instead of commands that might change central authentication sources) on platforms that implement it.
<code>name</code>	This parameter sets the name of the group to manage.
<code>state</code>	This parameter determines whether the group should be <code>present</code> or <code>absent</code> on the remote host.
<code>system</code>	If this parameter is set to <code>true</code> , then the group is created as a system group (typically, with a GID number below 1000).

The Known Hosts Module

The `ansible.builtin.known_hosts` module manages SSH host keys by adding or removing them on managed hosts. This ensures that managed hosts can automatically establish the authenticity of SSH connections to other managed hosts, ensuring that users are not prompted to verify a remote managed host's SSH fingerprint the first time they connect to it.

The following example demonstrates how the `ansible.builtin.known_hosts` module copies a host key to a managed host:

```
- name: Copy host keys to remote servers
  ansible.builtin.known_hosts:
    path: /etc/ssh/ssh_known_hosts ①
    name: servera.lab.example.com ②
    key: servera.lab.example.com,172.25.250.10 ssh-rsa ASDeararAIUHI324324 ③
```

- ① The `path` parameter specifies the path to the `known_hosts` file to edit. If the file does not exist, then it is created.
- ② The `name` parameter specifies the name of the host to add or remove. The name must match the hostname or IP address of the key being added.
- ③ The `key` parameter specifies the host key to add. It includes the host name, IP address, and the key itself in a specific format.

- ③ The key parameter is the SSH public host key as a string in a specific format. For example, the value for the key parameter must be in the format <hostname[, IP]> ssh-rsa <pubkey> for an RSA public host key (found in a host's /etc/ssh/ssh_host_rsa_key.pub key file), or <hostname[, IP]> ssh-ed25519 <pubkey> for an Ed25519 public host key (found in a host's /etc/ssh/ssh_host_ed25519_key.pub key file).

The following example demonstrates how to use the `lookup` plug-in to populate the key parameter from an existing file in the Ansible project:

```
- name: Copy host keys to remote servers
  ansible.builtin.known_hosts:
    path: /etc/ssh/ssh_known_hosts
    name: serverb
    key: "{{ lookup('ansible.builtin.file', 'pubkeys/serverb') }}" ❶
```

- ❶ This Jinja2 expression uses the `lookup` function with the `ansible.builtin.file` lookup plug-in to load the content of the `pubkeys/serverb.lab.example.com` key file from the Ansible project as the value of the `key` option. You can list available lookup plug-ins using the `ansible-navigator doc -l -t lookup` command.

The following play is an example that uses some advanced techniques to construct an `/etc/ssh/ssh_known_hosts` file for all managed hosts in the inventory. There might be more efficient ways to accomplish this, because it runs a nested loop on all managed hosts.

It uses the `ansible.builtin.slurp` module to get the content of the RSA and Ed25519 SSH public host keys in Base64 format, and then processes the values of the registered variable with the `b64decode` and `trim` filters to convert those values back to plain text.

```
- name: Configure /etc/ssh/ssh_known_hosts files
  hosts: all

  tasks:
    - name: Collect RSA keys
      ansible.builtin.slurp:
        src: /etc/ssh/ssh_host_rsa_key.pub
        register: rsa_host_keys

    - name: Collect Ed25519 keys
      ansible.builtin.slurp:
        src: /etc/ssh/ssh_host_ed25519_key.pub
        register: ed25519_host_keys

    - name: Deploy known_hosts
      ansible.builtin.known_hosts:
        path: /etc/ssh/ssh_known_hosts
        name: "{{ item[0] }}" ❶
        key: "{{ hostvars[ item[0] ]['ansible_facts']['fqdn'] }}"
        {{ hostvars[ item[0] ][ item[1] ]['content'] | b64decode | trim }}" ❷
        state: present
        with_nested:
          - "{{ ansible_play_hosts }}"
            ❸
          - [ 'rsa_host_keys', 'ed25519_host_keys' ] ❹
```

- ➊ `item[0]` is an inventory hostname from the list in the `ansible_play_hosts` variable.
- ➋ `item[1]` is the string `rsa_host_keys` or `ed25519_host_keys`. The `b64decode` filter converts the value stored in the variable from Base64 to plain text, and the `trim` filter removes an unnecessary newline. This is all one line starting with `key`, and there is a single space between the two Jinja2 expressions.
- ➌ `ansible_play_hosts` is a list of the hosts remaining in the play at this point, taken from the inventory and removing hosts with failed tasks. The play must retrieve the RSA and Ed25519 public host keys for each of the other hosts when it constructs the `known_hosts` file on each host in the play.
- ➍ This is a two-item list of the two variables that the play uses to store host keys.

**Note**

Lookup plug-ins and filters are covered in more detail in the course *DO374: Developing Advanced Automation with Red Hat Ansible Automation Platform*.

The Authorized Key Module

The `ansible.posix.authorized_key` module manages SSH authorized keys for user accounts on managed hosts.

The following example demonstrates how to use the `ansible.posix.authorized_key` module to add an SSH key to a managed host:

```
- name: Set authorized key
  ansible.posix.authorized_key:
    user: user1 ➊
    state: present ➋
    key: "{{ lookup('ansible.builtin.file', 'files/user1/id_rsa.pub') }}" ➌
```

- ➊ The `user` parameter specifies the username of the user whose `authorized_keys` file is modified on the managed host.
- ➋ The `state` parameter accepts the `present` or `absent` value with `present` as the default.
- ➌ The `key` parameter specifies the SSH public key to add or remove. In this example, the `lookup` function uses the `ansible.builtin.file` lookup plug-in to load the contents of the `files/user1/id_rsa.pub` file in the Ansible project as the value for `key`. As an alternative, you can provide a URL to a public key file as this value.

Configuring Sudo Access for Users and Groups

In Red Hat Enterprise Linux 9, you can configure access for a user or group to run `sudo` commands without requiring a password prompt.

The following example demonstrates how to use the `ansible.builtin.lineinfile` module to provide a group with sudo access to the `root` account without prompting the group members for a password:

```
- name: Modify sudo to allow the group01 group sudo without a password
  ansible.builtin.lineinfile:
    path: /etc/sudoers.d/group01 ①
    state: present ②
    create: true ③
    mode: 0440 ④
    line: "%group01 ALL=(ALL) NOPASSWD: ALL" ⑤
    validate: /usr/sbin/visudo -cf %s ⑥
```

- ①** The path parameter specifies the file to modify in the /etc/sudoers.d/ directory. It is a good practice to match the file name with the name of the user or group you are providing access to. This makes it easier for future reference.
- ②** The state parameter accepts the present or absent value. The default value is present.
- ③** The create parameter takes a Boolean value and specifies if the file should be created if it does not already exist. The default value for the create parameter is false.
- ④** The mode parameter specifies the permissions on the sudoers file.
- ⑤** The line parameter specifies the line to add to the file. The format is specific, and an example can be found in the /etc/sudoers file under the "Same thing but without a password" comment. If you are configuring sudo access for a group, then you need to add a percent sign (%) to the beginning of the group name. If you are configuring sudo access for a user, then do not add the percent sign.
- ⑥** The validate parameter specifies the command to run to verify that the file is correct. When the validate parameter is present, the file is created in a temporary file path and the provided command validates the temporary file. If the validate command succeeds, then the temporary file is copied to the path specified in the path parameter and the temporary file is removed.

An example of the sudo validation command can be found in the examples section of the output from the `ansible-navigator doc ansible.builtin.lineinfile` command.



References

Users Module Ansible Documentation

http://docs.ansible.com/ansible/latest/modules/user_module.html#user-module

How do I generate encrypted passwords for the user module

https://docs.ansible.com/ansible/latest/reference_appendices/faq.html#how-do-i-generate-encrypted-passwords-for-the-user-module

Group Module Ansible Documentation

https://docs.ansible.com/ansible/latest/modules/group_module.html#group-module

SSH Known Hosts Module Ansible Documentation

https://docs.ansible.com/ansible/latest/modules/known_hosts_module.html#known-hosts-module

Authorized Key Module Ansible Documentation

https://docs.ansible.com/ansible/latest/modules/authorized_key_module.html#authorized-key-module

The Lookup Plugin Ansible Documentation

<https://docs.ansible.com/ansible/latest/plugins/lookup.html?highlight=lookup>

Using Filters to Manipulate Data

https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html

The Line in File Module Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/lineinfile_module.html

► Guided Exercise

Managing Users and Authentication

In this exercise, you manage users and groups, adjust Sudo configuration, and configure SSH on your managed hosts.

Outcomes

- Create a new user group.
- Manage users by using the `ansible.builtin.user` module.
- Populate SSH authorized keys by using the `ansible.posix.authorized_key` module.
- Modify the `/etc/ssh/sshd_config` file and a configuration file in `/etc/sudoers.d` by using the `ansible.builtin.lineinfile` module.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start system-users
```

Instructions

Your organization requires that all hosts have the same local users available. These users should belong to the `webadmin` user group, which can use the `sudo` command without specifying a password. Also, the users' SSH public keys should be distributed in the environment to their `~/.ssh/authorized_keys` files to allow key-based authentication, and the `root` user should not be allowed to log in directly using SSH.

Write a playbook to ensure that the users specified in the `/home/student/system-users/vars/users_vars.yml` file and the `webadmin` user group are present on the managed hosts and meet the preceding criteria.

- 1. Change into the `/home/student/system-users` directory.

```
[student@workstation ~]$ cd ~/system-users  
[student@workstation system-users]$
```

- 2. Examine the existing `vars/users_vars.yml` variable file.

```
---  
users:  
  - username: user1  
    groups: webadmin
```

```

- username: user2
  groups: webadmin
- username: user3
  groups: webadmin
- username: user4
  groups: webadmin
- username: user5
  groups: webadmin

```

It uses the `username` variable name to set the correct username, and the `groups` variable to define supplementary groups that the user should belong to.

- 3. Start writing the `/home/student/system-users/users.yml` playbook. Define a single play in the playbook that targets the `webservers` host group.

Add a `vars_files` clause that loads the variables in the `vars/users_vars.yml` file, which has been created for you, and which contains all the usernames that are required for this exercise.

Add the `tasks` clause to the playbook.

The `users.yml` playbook should consist of the following content:

```

---
- name: Create multiple local users
  hosts: webservers
  vars_files:
    - vars/users_vars.yml
  tasks:

```

- 4. Add two tasks to the play.

Create the `webadmin` user group on the managed hosts by using the `ansible.builtin.group` module in the first task.

Create the users specified in the `vars/users_vars.yml` file by using the `ansible.builtin.user` module to loop over those variables in the second task.

Run the `users.yml` playbook.

- 4.1. Add the first task to the play to ensure that the group `webadmin` exists on your managed hosts.

The first task should consist of the following content:

```

- name: Add webadmin group
  ansible.builtin.group:
    name: webadmin
    state: present

```

- 4.2. Add a second task to the play that uses the `ansible.builtin.user` module to ensure that the users exist on your managed hosts.

Add a `loop: "{{ users }}"` clause to the task to iterate over every user found in the `vars/users_vars.yml` file.

For the `name` parameter for the users, use `item['username']`, not `item`. This allows each list item in the variable file to be structured as a dictionary of multiple variables that includes additional information about each user in the list, specifically which supplementary groups the user should be a member of.

The entire playbook should consist of the following content:

```
---
- name: Create multiple local users
  hosts: webservers
  vars_files:
    - vars/users_vars.yml
  tasks:

    - name: Add webadmin group
      ansible.builtin.group:
        name: webadmin
        state: present

    - name: Create user accounts
      ansible.builtin.user:
        name: "{{ item['username'] }}"
        groups: "{{ item['groups'] }}"
      loop: "{{ users }}"

```

4.3. Run the `users.yml` playbook:

```
[student@workstation system-users]$ ansible-navigator run \
> -m stdout users.yml

PLAY [Create multiple local users] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Add webadmin group] ****
changed: [servera.lab.example.com]

TASK [Create user accounts] ****
changed: [servera.lab.example.com] => (item={'username': 'user1', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user2', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user3', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user4', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user5', 'groups': 'webadmin'})

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=2    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0
```

- ▶ 5. Ensure the SSH public keys have been properly distributed to the managed hosts by adding a third task to the play that uses the `ansible.posix.authorized_key` module.

In the `files` directory, each user has a unique SSH public key file. The module loops through the list of users, finds the appropriate key by using the `username` variable, and pushes the key to the managed host.

For the `key` parameter, use the following Jinja2 expression for its value to evaluate the contents of the appropriate public key file. A lookup function is used with the `file` plug-in to read the file, and its file name is constructed using string operations.

```
"{{ lookup('file', 'files/' + item['username'] + '.key.pub') }}"
```

The third task should consist of the following content:

```
- name: Add authorized keys
ansible.posix.authorized_key:
  user: "{{ item['username'] }}"
  key: "{{ lookup('file', 'files/' + item['username'] + '.key.pub') }}"
  loop: "{{ users }}"
```

- ▶ 6. Add a fourth task to the play that uses the `ansible.builtin.lineinfile` module to modify the `sudo` configuration file and allow the `webadmin` group members to use `sudo` without a password on the managed host. Use the `validate` parameter to validate the new configuration entry.

The fourth task should consist of the following content:

```
- name: Modify sudo config to allow webadmin users sudo without a password
ansible.builtin.lineinfile:
  path: /etc/sudoers.d/webadmin
  state: present
  create: yes
  mode: 0440
  line: "%webadmin ALL=(ALL) NOPASSWD: ALL"
  validate: /usr/sbin/visudo -cf %s
```

- ▶ 7. Add a fifth task to ensure that the `root` user is not permitted to log in using SSH directly. Use `notify: "Restart sshd"` to trigger a handler to restart SSH.

The fifth task should consist of the following content:

```
- name: Disable root login via SSH
ansible.builtin.lineinfile:
  dest: /etc/ssh/sshd_config
  regexp: "^\$PermitRootLogin"
  line: "PermitRootLogin no"
  notify: Restart sshd
```

- ▶ 8. In the first line after the location of the variable file, add a new handler definition named `Restart sshd`.

8.1. Define the `Restart sshd` handler as follows:

```
...output omitted...
- vars/users_vars.yml
handlers:
- name: Restart sshd
  ansible.builtin.service:
    name: sshd
    state: restarted
```

8.2. The `users.yml` playbook should consist of the following content:

```
---
- name: Create multiple local users
hosts: webservers
vars_files:
- vars/users_vars.yml
handlers:
- name: Restart sshd
  ansible.builtin.service:
    name: sshd
    state: restarted

tasks:
- name: Add webadmin group
  ansible.builtin.group:
    name: webadmin
    state: present

- name: Create user accounts
  ansible.builtin.user:
    name: "{{ item['username'] }}"
    groups: "{{ item['groups'] }}"
    loop: "{{ users }}"

- name: Add authorized keys
  ansible.posix.authorized_key:
    user: "{{ item['username'] }}"
    key: "{{ lookup('file', 'files/' + item['username'] + '.key.pub') }}"
    loop: "{{ users }}"

- name: Modify sudo config to allow webadmin users sudo without a password
  ansible.builtin.lineinfile:
    path: /etc/sudoers.d/webadmin
    state: present
    create: yes
    mode: 0440
    line: "%webadmin ALL=(ALL) NOPASSWD: ALL"
    validate: /usr/sbin/visudo -cf %

- name: Disable root login via SSH
  ansible.builtin.lineinfile:
    dest: /etc/ssh/sshd_config
```

```
regexp: '^PermitRootLogin'
line: "PermitRootLogin no"
notify: "Restart sshd"
```

8.3. Run the users.yml playbook.

```
[student@workstation system-users]$ ansible-navigator run \
> -m stdout users.yml

PLAY [Create multiple local users] *****

TASK [Gathering Facts] *****
ok: [servera.lab.example.com]

TASK [Add webadmin group] *****
ok: [servera.lab.example.com]

TASK [Create user accounts] *****
ok: [servera.lab.example.com] => (item={'username': 'user1', 'groups': 'webadmin'})
ok: [servera.lab.example.com] => (item={'username': 'user2', 'groups': 'webadmin'})
ok: [servera.lab.example.com] => (item={'username': 'user3', 'groups': 'webadmin'})
ok: [servera.lab.example.com] => (item={'username': 'user4', 'groups': 'webadmin'})
ok: [servera.lab.example.com] => (item={'username': 'user5', 'groups': 'webadmin'})

TASK [Add authorized keys] *****
changed: [servera.lab.example.com] => (item={'username': 'user1', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user2', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user3', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user4', 'groups': 'webadmin'})
changed: [servera.lab.example.com] => (item={'username': 'user5', 'groups': 'webadmin'})

TASK [Modify sudo config to allow webadmin users sudo without a password] ***
changed: [servera.lab.example.com]

TASK [Disable root login via SSH] *****
changed: [servera.lab.example.com]

RUNNING HANDLER [Restart sshd] *****
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com      : ok=7      changed=4      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=0
```

- 9. Use SSH as the `user1` user and log in to the `servera` server. After logging in, use `sudo -i` command to change to the `root` user.

9.1. Use SSH as the `user1` user and log in to the `servera` server.

```
[student@workstation system-users]$ ssh user1@servera
Register this system with Red Hat Insights: insights-client --register
Create an account or view all your systems at https://red.ht/insights-dashboard

[user1@servera ~]$
```

9.2. Change to the `root` user.

```
[user1@servera ~]$ sudo -i
root@servera ~]#
```

9.3. Log out of `servera`.

```
[root@servera ~]# exit
logout
[user1@servera ~]$ exit
logout
Connection to servera closed.
[student@workstation system-users]$
```

- 10. Try to log in to `servera` as the `root` user directly. This step should fail because the SSH daemon configuration has been modified not to permit direct `root` user logins.

10.1. From the `workstation` machine, use SSH as the `root` user and log in to the `servera` server.

```
[student@workstation system-users]$ ssh root@servera
root@servera's password: redhat
Permission denied, please try again.
root@servera's password:
```

This confirms that the SSH configuration denied direct access to the system for the `root` user.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-users
```

This concludes the section.

Managing the Boot Process and Scheduled Processes

Objectives

- Manage service startup, schedule processes with at, cron, and systemd, reboot managed hosts with reboot, and control the default boot target on managed hosts.

Scheduling Jobs for Future Execution

Red Hat Enterprise Linux provides several mechanisms that you can use to schedule commands to run at some point in the future. The `at` command schedules jobs that run once at a specified time. The Cron subsystem schedules jobs to run on a recurring schedule, either in a user's personal `crontab` file, in the system Cron configuration in `/etc/crontab`, or as a file in `/etc/cron.d`. The `systemd` subsystem also provides timer units that can start service units on a set schedule.

Scheduling Jobs That Run One Time

Quick one-time scheduling is done with the `ansible.posix.at` module. You create the job to run at a future time, and it is held until that time to execute.

Options for the `ansible.posix.at` Module

Option	Comments
<code>command</code>	The command to schedule to run in the future.
<code>count</code>	The integer number of units from now that the job should run. (Must be used with <code>units</code> .)
<code>units</code>	Specifies whether <code>count</code> is measured in minutes, hours, days, or weeks.
<code>script_file</code>	An existing script file to schedule to run in the future.
<code>state</code>	The default value (<code>present</code>) adds a job; <code>absent</code> removes a matching job if present.
<code>unique</code>	If set to <code>yes</code> , then if a matching job is already present, a new job is not added.

In the following example, the task shown uses `at` to schedule the `userdel -r tempuser` command to run in 20 minutes.

```
- name: Remove tempuser
ansible.posix.at:
  command: userdel -r tempuser
  count: 20
  units: minutes
  unique: yes
```

Scheduling Repeating Jobs with Cron

You can configure a command that runs on a repeating schedule by using Cron. To set up Cron jobs, use the `ansible.builtin.cron` module. The `name` option is mandatory, and is inserted in the `crontab` as a description of the repeating job. It is also used by the module to determine if the Cron job already exists, or which Cron job to modify or delete.

Some commonly used parameters for the `ansible.builtin.cron` module include:

Options for the `ansible.builtin.cron` Module

Options	Comments
<code>name</code>	The comment identifying the Cron job.
<code>job</code>	The command to run.
<code>minute, hour, day, month, weekday</code>	The value for the field in the time specification for the job in the <code>crontab</code> entry. If not set, "*" (all values) is assumed.
<code>state</code>	If set to <code>present</code> , it creates the Cron job (the default); <code>absent</code> removes it.
<code>user</code>	The Cron job runs as this user. If <code>cron_file</code> is not specified, the job is set in that user's <code>crontab</code> file.
<code>cron_file</code>	If set, create a system Cron job in <code>cron_file</code> . You must specify <code>user</code> and a time specification. If you use a relative path, then the file is created in <code>/etc/cron.d</code> .

This first example task creates a Cron job in the `testing` user's personal `crontab` file. It runs their personal `backup-home-dir` script at 16:00 every Friday. You could log in as that user and run `crontab -l` after running the playbook to confirm that it worked.

```
- name: Schedule backups for my home directory
ansible.builtin.cron:
  name: Backup my home directory
  user: testing
  job: /home/testing/bin/backup-home-dir
  minute: 0
  hour: 16
  weekday: 5
```

In the following example, the task creates a system Cron job in the `/etc/cron.d/flush_bolt` file that runs a command as `root` to flush the Bolt cache every morning at 11:45.

```
- name: Schedule job to flush the Bolt cache
ansible.builtin.cron:
  name: Flush Bolt cache
  cron_file: flush_bolt
  user: "root"
  minute: 45
  hour: 11
  job: "php ./app/nut cache:clear"
```

**Warning**

Do not use `cron_file` to modify the `/etc/crontab` file. The file you specify must only be maintained by Ansible and should only contain the entry specified by the task.

Controlling Systemd Timer Units

The `ansible.builtin.systemd` module can be used to enable or disable existing `systemd` timer units that run recurring jobs (usually `systemd` service units that eventually exit).

The following example disables and stops the `systemd` timer that automatically populates the `dnf` package cache on Red Hat Enterprise Linux 9.

```
- name: Disable dnf makecache
ansible.builtin.systemd:
  name: dnf-makecache.timer
  state: stopped
  enabled: no
```

Managing Services

You can choose between two modules to manage services or reload daemons: `ansible.builtin.systemd` and `ansible.builtin.service`.

The `ansible.builtin.service` module is intended to work with a number of service-management systems, including `systemd`, Upstart, SysVinit, BSD `init`, and others. Because it provides a generic interface to the initialization system, it offers a basic set of options to start, stop, restart, and enable services and other daemons.

```
- name: Start and enable nginx
ansible.builtin.service:
  name: nginx
  state: started
  enabled: yes
```

The `ansible.builtin.systemd` module is designed to work with `systemd` only, but it offers additional configuration options specific to that system and service manager.

The following example that uses `ansible.builtin.systemd` is equivalent to the preceding example that used `ansible.builtin.service`:

```
- name: Start nginx
ansible.builtin.systemd:
  name: nginx
  state: started
  enabled: yes
```

The next example reloads the `httpd` daemon, but before it does that it runs `systemctl daemon-reload` to reload the entire `systemd` configuration.

```
- name: Reload web server
ansible.builtin.systemd:
  name: httpd
  state: reloaded
  daemon_reload: yes
```

Setting the Default Boot Target

The `ansible.builtin.systemd` module cannot set the default boot target. You can use the `ansible.builtin.command` module to set the default boot target.

```
- name: Change default systemd target
hosts: all
gather_facts: false

vars:
  systemd_target: "multi-user.target" ①

tasks:
  - name: Get current systemd target
    ansible.builtin.command:
      cmd: systemctl get-default ②
      changed_when: false ③
      register: target ④

  - name: Set default systemd target
    ansible.builtin.command:
      cmd: systemctl set-default {{ systemd_target }} ⑤
      when: systemd_target not in target['stdout'] ⑥
      become: true ⑦
```

- ① This variable holds the name of the default target you want.
- ② This gets the current target.
- ③ Because this is just gathering information, the task should never report changed.
- ④ The `target` variable holds the information that was gathered.
- ⑤ This command sets the default target.
- ⑥ Skip this task if the current default target is already the desired default target. This ensures the task is idempotent.
- ⑦ This is the only task in this play that requires root access.

Rebooting Managed Hosts

You can use the dedicated `ansible.builtin.reboot` module to reboot managed hosts during playbook execution. This module reboots the managed host, and waits until the managed host comes back up before continuing with playbook execution. The module determines that a managed host is back up by waiting until Ansible can run a command on the managed host.

The following simple example immediately triggers a reboot:

```
- name: Reboot now
ansible.builtin.reboot:
```

By default, the playbook waits up to 600 seconds before deciding that the reboot failed, and another 600 seconds before deciding that the test command failed. You can adjust this value so that the timeouts are each 180 seconds. For example:

```
- name: Reboot, shorten timeout
ansible.builtin.reboot:
    reboot_timeout: 180
```

Some other useful options to the module include:

Options for the `ansible.builtin.reboot` Module

Options	Comments
<code>pre_reboot_delay</code>	The time to wait before reboot. On Linux, this is measured in minutes, and if less than 60, is rounded down to 0.
<code>msg</code>	The message to display to users before reboot.
<code>test_command</code>	The command used to determine whether the managed host is usable and ready for more Ansible tasks after reboot. The default is <code>whoami</code> .



References

ansible.posix.at module – Schedule the execution of a command or script file via the at command – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/posix/at_module.html

ansible.builtin.cron module – Manage cron.d and crontab entries – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/cron_module.html

ansible.builtin.reboot module – Reboot a machine – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/reboot_module.html

ansible.builtin.service module – Manage services – Ansible Documentation

https://docs.ansible.com/ansible/latest/modules/service_module.html

ansible.builtin.systemd module – Manage systemd units – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/systemd_module.html

► Guided Exercise

Managing the Boot Process and Scheduled Processes

In this exercise, you manage the startup process, schedule recurring jobs, and reboot managed hosts.

Outcomes

- Schedule a Cron job.
- Remove a single, specific Cron job from a `crontab` file.
- Schedule an `at` task.
- Set the default boot target on managed hosts.
- Reboot managed hosts.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available, including the `/home/student/system-process` project directory.

```
[student@workstation ~]$ lab start system-process
```

Instructions

- 1. Change into the `/home/student/system-process` directory.

```
[student@workstation ~]$ cd ~/system-process  
[student@workstation system-process]$
```

- 2. Create the `create_crontab_file.yml` playbook in the working directory.

Configure the playbook to use the `ansible.builtin.cron` module to create a `crontab` file named `/etc/cron.d/add-date-time` that schedules a recurring Cron job. The job should run as the `devops` user every two minutes starting at 09:00 and ending at 16:59 from Monday through Friday. The job should append the current date and time to the `/home/devops/my_datetime_cron_job` file.

- 2.1. Create the `create_crontab_file.yml` playbook and add the lines needed to start the play. It should target the managed hosts in the `webservers` group and enable privilege escalation.

```
---
- name: Recurring cron job
  hosts: webservers
  become: true
```

- 2.2. Define a task that uses the `ansible.builtin.cron` module to schedule a recurring Cron job, the `date >> /home/devops/my_date_time_cron_job` command.



Note

The `ansible.builtin.cron` module provides a `name` option to uniquely describe the `crontab` file entry and to ensure expected results.

The value you use for the `name` option is added to the `crontab` file as a comment. For example, the `name` option is required if you are removing a `crontab` entry using `state: absent`.

Additionally, the `name` option prevents a new `crontab` entry from always being created when the default state, `state: present`, is set.

```
tasks:
- name: Crontab file exists
  ansible.builtin.cron:
    name: Add date and time to a file
    job: date >> /home/devops/my_date_time_cron_job
```

- 2.3. Configure the job to run every two minutes starting at 09:00 and ending at 16:59 on Monday through Friday.

```
minute: "*/2"
hour: 9-16
weekday: 1-5
```

- 2.4. Use the `cron_file` parameter to use the `crontab` file named `/etc/cron.d/add-date-time` instead of an individual user's `crontab` in `/var/spool/cron/`.

A relative path places the file in the `/etc/cron.d` directory.

If the `cron_file` parameter is used, you must also specify the `user` parameter for the system `crontab` file. Use the `devops` user for this job.

```
user: devops
cron_file: add-date-time
state: present
```

- 2.5. When completed, the playbook must appear as follows. Review the playbook for accuracy.

```
---
- name: Recurring cron job
  hosts: webservers
```

```
become: true

tasks:
  - name: Crontab file exists
    ansible.builtin.cron:
      name: Add date and time to a file
      job: date >> /home/devops/my_date_time_cron_job
      minute: "*/2"
      hour: 9-16
      weekday: 1-5
      user: devops
      cron_file: add-date-time
      state: present
```

- 2.6. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout create_crontab_file.yml --syntax-check
playbook: /home/student/system-process/create_crontab_file.yml
```

- 2.7. Run the `create_crontab_file.yml` playbook.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout create_crontab_file.yml

PLAY [Recurring cron job] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Crontab file exists] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0
```

- 2.8. Run the following command to verify that the `/etc/cron.d/add-date-time` cron file exists, and its content is correct.

```
[student@workstation system-process]$ ssh devops@servera \
> "cat /etc/cron.d/add-date-time"
#Ansible: Add date and time to a file
*/2 9-16 * * 1-5 devops date >> /home/devops/my_date_time_cron_job
```

- 3. Create the `remove_cron_job.yml` playbook in the working directory. Configure the playbook to use the `ansible.builtin.cron` module to remove the `Add date and time to a file` Cron job from the `/etc/cron.d/add-date-time` crontab file.

- 3.1. Create the `remove_cron_job.yml` playbook and add the following lines:

```
---
- name: Remove scheduled cron job
  hosts: webservers
  become: true

  tasks:
    - name: Cron job removed
      ansible.builtin.cron:
        name: Add date and time to a file
        user: devops
        cron_file: add-date-time
        state: absent
```

- 3.2. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout remove_cron_job.yml --syntax-check
playbook: /home/student/system-process/remove_cron_job.yml
```

- 3.3. Run the `remove_cron_job.yml` playbook.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout remove_cron_job.yml

PLAY [Remove scheduled cron job] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Cron job removed] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0
```

- 3.4. Run the following command to verify that the `/etc/cron.d/add-date-time` file has been removed.

```
[student@workstation system-process]$ ssh devops@servera \
> "ls -l /etc/cron.d"
total 4
-rw-r--r--. 1 root root 128 Aug  9  2021 @hourly
```

- ▶ 4. Create the `schedule_at_task.yml` playbook in the working directory. Configure the playbook to use the `ansible.posix.at` module to schedule a task that runs one minute in the future.

The task should run the `date` command and redirect its output to the `/home/devops/my_at_date_time` file.

Use the `unique: yes` option to ensure that if the command already exists in the `at` queue, a new task is not added.

4.1. Create the `schedule_at_task.yml` playbook and add the following lines:

```
---
- name: Schedule at task
  hosts: webservers
  become: true
  become_user: devops

  tasks:
    - name: Create date and time file
      ansible.posix.at:
        command: date > ~/my_at_date_time
        count: 1
        units: minutes
        unique: yes
        state: present
```

4.2. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout schedule_at_task.yml --syntax-check
playbook: /home/student/system-process/schedule_at_task.yml
```

4.3. Run the `schedule_at_task.yml` playbook.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout schedule_at_task.yml

PLAY [Schedule at task] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Create date and time file] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
                           skipped=0   rescued=0   ignored=0
```

4.4. After waiting a minute or two for the `at` command to complete, run the following commands to verify that the `/home/devops/my_at_date_time` file exists and has the correct contents.

```
[student@workstation system-process]$ ssh devops@servera \
> "ls -l my_at_date_time"
-rw-rw-r--. 1 devops devops 32 Aug 15 00:00 my_at_date_time

[student@workstation system-process]$ ssh devops@servera \
> "cat my_at_date_time"
Mon Aug 15 12:00:00 AM EDT 2022
```

- 5. Create the `set_default_boot_target_graphical.yml` playbook in the working directory. Write a play in the playbook to set the default `systemd` target to `graphical.target`.

- 5.1. Create the `set_default_boot_target_graphical.yml` playbook and add the following lines:

```
---
- name: Change default boot target
  hosts: webservers
  become: true
  gather_facts: false
  vars:
    default_target: "graphical.target"

  tasks:
    - name: Get current boot target
      ansible.builtin.command:
        cmd: systemctl get-default
      changed_when: false
      register: target

    - name: Set default boot target
      ansible.builtin.command:
        cmd: systemctl set-default {{ default_target }}
      when: default_target not in target['stdout']
```

- 5.2. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout set_default_boot_target_graphical.yml --syntax-check
playbook: /home/student/system-process/set_default_boot_target_graphical.yml
```

- 5.3. Before running the playbook, run the following command to verify that the current default boot target is `multi-user.target`.

```
[student@workstation system-process]$ ssh devops@servera \
> "systemctl get-default"
multi-user.target
```

- 5.4. Run the `set_default_boot_target_graphical.yml` playbook.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout set_default_boot_target_graphical.yml

PLAY [Change default boot target] ****
TASK [Get current boot target] ****
ok: [servera.lab.example.com]

TASK [Set default boot target] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

- 5.5. Run the following command to verify that the default boot target is now `graphical.target`.

```
[student@workstation system-process]$ ssh devops@servera \
> "systemctl get-default"
graphical.target
```

- ▶ 6. Create the `reboot_hosts.yml` playbook in the working directory to reboot the managed hosts.

- 6.1. Create the `reboot_hosts.yml` playbook and add the following lines:

```
---
- name: Reboot hosts
hosts: webservers
become: true

tasks:
- name: Hosts are rebooted
  ansible.builtin.reboot:
```

- 6.2. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout reboot_hosts.yml --syntax-check
playbook: /home/student/system-process/reboot_hosts.yml
```

- 6.3. Before running the playbook, run the following command to determine the time stamp of the last system reboot:

```
[student@workstation system-process]$ ssh devops@servera \
> "who -b"
system boot 2022-08-15 00:07
```

- 6.4. Run the `reboot_hosts.yml` playbook.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout reboot_hosts.yml

PLAY [Reboot hosts] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Hosts are rebooted] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

- 6.5. Run the following command to determine the time stamp of the last system boot. The time stamp displayed after the playbook runs should be later.

```
[student@workstation system-process]$ ssh devops@servera \
> "who -b"
system boot 2022-08-15 00:44
```

- 6.6. Run this next command to determine that the `graphical.target` boot target is still the default after the reboot.

```
[student@workstation system-process]$ ssh devops@servera \
> "systemctl get-default"
graphical.target
```

- 7. To maintain consistency throughout the remaining exercises, change the default boot target back to its former setting, `multi-user.target`.

Copy your `set_default_boot_target_graphical.yml` playbook to `set_default_boot_target_multi-user.yml` in the Ansible project directory. Edit the `default_target` variable to set `multi-user.target` as the default.

- 7.1. Copy your `set_default_boot_target_graphical.yml` playbook to `set_default_boot_target_multi-user.yml`.

```
[student@workstation system-process]$ cp set_default_boot_target_graphical.yml \
> set_default_boot_target_multi-user.yml
```

- 7.2. Edit the play in the `set_default_boot_target_multi-user.yml` playbook to change the `default_target` variable to `multi-user.target`.

```
---
- name: Change default boot target
  hosts: webservers
  become: true
  gather_facts: false
  vars:
```

```

default_target: "multi-user.target"

tasks:
  - name: Get current boot target
    ansible.builtin.command:
      cmd: systemctl get-default
    changed_when: false
    register: target

  - name: Set default boot target
    ansible.builtin.command:
      cmd: systemctl set-default {{ default_target }}
    when: default_target not in target['stdout']

```

- 7.3. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout set_default_boot_target_multi-user.yml --syntax-check
playbook: /home/student/system-process/set_default_boot_target_multi-user.yml
```

- 7.4. Run the `set_default_boot_target_multi-user.yml` playbook.

```
[student@workstation system-process]$ ansible-navigator run \
> -m stdout set_default_boot_target_multi-user.yml

PLAY [Change default boot target] ****
TASK [Get current boot target] ****
ok: [servera.lab.example.com]

TASK [Set default boot target] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
                          skipped=0   rescued=0   ignored=0
```

- 7.5. Run the following command to verify that the default boot target is now `multi-user.target`.

```
[student@workstation system-process]$ ssh devops@servera \
> "systemctl get-default"
multi-user.target
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-process
```

This concludes the section.

Managing Storage

Objectives

- Partition storage devices, configure LVM, format partitions or logical volumes, mount file systems, and add swap spaces.

Mounting Existing File Systems

Use the `ansible.posix.mount` module to mount an existing file system. The most common parameters for the `ansible.posix.mount` module are the `path` parameter, which specifies the path to mount the file system to, the `src` parameter, which specifies the device (this could be a device name, UUID, or NFS volume), the `fstype` parameter, which specifies the file system type, and the `state` parameter, which accepts the `absent`, `mounted`, `present`, `unmounted`, or `remounted` values.

The following example task mounts the NFS share available at `172.25.250.100:/share` on the `/nfsshare` directory on the managed hosts.

```
- name: Mount NFS share
  ansible.posix.mount:
    path: /nfsshare
    src: 172.25.250.100:/share
    fstype: nfs
    opts: defaults
    dump: '0'
    passno: '0'
    state: mounted
```

Configuring Storage with the Storage System Role

Red Hat Ansible Automation Platform provides the `redhat.rhel_system_roles.storage` system role to configure local storage devices on your managed hosts. It can manage file systems on unpartitioned block devices, and format and create logical volumes on LVM physical volumes based on unpartitioned block devices.



Important

The `redhat.rhel_system_roles.storage` role formally supports managing file systems and mount entries for two use cases:

- Unpartitioned devices (whole-device file systems)
- LVM on unpartitioned whole-device physical volumes

If you have other use cases, then you might need to use other modules and roles to implement them.

Managing a File System on an Unpartitioned Device

To create a file system on an unpartitioned block device with the `redhat.rhel_system_roles.storage` role, define the `storage_volumes` variable. The `storage_volumes` variable contains a list of storage devices to manage.

The following dictionary items are available in the `storage_volumes` variable:

Parameters for the `storage_volumes` Variable

Parameter	Comments
<code>name</code>	The name of the volume.
<code>type</code>	This value must be <code>disk</code> .
<code>disks</code>	Must be a list of exactly one item; the unpartitioned block device.
<code>mount_point</code>	The directory on which the file system is mounted.
<code>fstype</code>	The file system type to use. (<code>xfs</code> , <code>ext4</code> , or <code>swap</code> .)
<code>mount_options</code>	Custom mount options, such as <code>ro</code> or <code>rw</code> .

The following example play creates an XFS file system on the `/dev/vdg` device, and mounts it on `/opt/extra`.

```
- name: Example of a simple storage device
  hosts: all

  roles:
    - name: redhat.rhel_system_roles.storage
      storage_volumes:
        - name: extra
          type: disk
          disks:
            - /dev/vdg
          fs_type: xfs
          mount_point: /opt/extra
```

Managing LVM with the Storage Role

To create an LVM volume group with the `redhat.rhel_system_roles.storage` role, define the `storage_pools` variable. The `storage_pools` variable contains a list of pools (LVM volume groups) to manage.

The dictionary items inside the `storage_pools` variable are used as follows:

- The `name` variable is the name of the volume group.
- The `type` variable must have the value `lvm`.
- The `disks` variable is the list of block devices that the volume group uses for its storage.
- The `volumes` variable is the list of logical volumes in the volume group.

The following entry creates the volume group `vg01` with the `type` key set to the value `lvm`. The volume group's physical volume is the `/dev/vdb` disk.

```

---
- name: Configure storage on webservers
  hosts: webservers

  roles:
    - name: redhat.rhel_system_roles.storage
      storage_pools:
        - name: vg01
          type: lvm
          disks:
            - /dev/vdb

```

**Important**

The disks option only supports unpartitioned block devices for your LVM physical volumes.

To create logical volumes, populate the `volumes` variable, nested under the `storage_pools` variable, with a list of logical volume names and their parameters. Each item in the list is a dictionary that represents a single logical volume within the `storage_pools` variable.

Each logical volume list item has the following dictionary variables:

- `name`: The name of the logical volume.
- `size`: The size of the logical volume.
- `mount_point`: The directory used as the mount point for the logical volume's file system.
- `fs_type`: The logical volume's file system type.
- `state`: Whether the logical volume should exist using the `present` or `absent` values.

The following example creates two logical volumes, named `lvol01` and `lvol02`. The `lvol01` logical volume is 128 MB in size, formatted with the `xfs` file system, and is mounted at `/data`. The `lvol02` logical volume is 256 MB in size, formatted with the `xfs` file system, and is mounted at `/backup`.

```

---
- name: Configure storage on webservers
  hosts: webservers

  roles:
    - name: redhat.rhel_system_roles.storage
      storage_pools:
        - name: vg01
          type: lvm
          disks:
            - /dev/vdb
      volumes:
        - name: lvol01
          size: 128m
          mount_point: "/data"
          fs_type: xfs
          state: present
        - name: lvol02
          size: 256m

```

```
mount_point: "/backup"
fs_type: xfs
state: present
```

In the following example entry, if the `lvol01` logical volume is already created with a size of 128 MB, then the logical volume and file system are enlarged to 256 MB, assuming that the space is available within the volume group.

```
volumes:
  - name: lvol01
    size: 256m
    mount_point: "/data"
    fs_type: xfs
    state: present
```

Configuring Swap Space

You can use the `redhat.rhel_system_roles.storage` role to create logical volumes that are formatted as swap spaces. The role creates the logical volume, the swap file system type, adds the swap volume to the `/etc/fstab` file, and enables the swap volume immediately.

The following playbook example creates the `lvswap` logical volume in the `vgswap` volume group, adds the swap volume to the `/etc/fstab` file, and enables the swap space.

```
---
- name: Configure a swap volume
  hosts: all

  roles:
    - name: redhat.rhel_system_roles.storage
      storage_pools:
        - name: vgswap
          type: lvm
          disks:
            - /dev/vdb
      volumes:
        - name: lvswap
          size: 512m
          fs_type: swap
          state: present
```

Managing Partitions and File Systems with Tasks

You can manage partitions and file systems on your storage devices without using the `system` role. However, the most convenient modules for doing this are currently unsupported by Red Hat, which can make this more complicated.



Warning

Be careful when you use Ansible to partition and format file systems, especially if you use `ansible.builtin.command` tasks. Mistakes in your code on important systems can result in lost data.

Managing Partitions

If you want to partition your storage devices without using the system role, your options are a bit more complex.

- The unsupported `community.general.parted` module in the `community.general` Ansible Content Collection can perform this task.
- You can use the `ansible.builtin.command` module to run the partitioning commands on the managed hosts. However, you need to take special care to make sure the commands are idempotent and do not inadvertently destroy data on your existing storage devices.

For example, the following task creates a GPT disk label and a `/dev/sda1` partition on the `/dev/sda` storage device only if `/dev/sda1` does not already exist:

```
- name: Ensure that /dev/sda1 exists
  ansible.builtin.command:
    cmd: parted --script mklabel gpt mkpart primary 1MiB 100%
    creates: /dev/sda1
```

This depends on the fact that if the `/dev/sda1` partition exists, then a Linux system creates a `/dev/sda1` device file for it automatically.

Managing File Systems

The easiest way to manage file systems without using the system role might be the `community.general.filesystem` module. However, Red Hat does not support this module, so you use it at your own risk.

As an alternative, you can use the `ansible.builtin.command` module to run commands to format file systems. However, you should use some mechanism to make sure that the device you are formatting does not already contain a file system, to ensure idempotency of your play, and to avoid accidental data loss. One way to do that might be to review storage-related facts gathered by Ansible to determine if a device appears to be formatted with a file system.

Ansible Facts for Storage Configuration

Ansible facts gathered by `ansible.builtin.setup` contain useful information about the storage devices on your managed hosts.

Facts about Block Devices

The `ansible_facts['devices']` fact includes information about all the storage devices available on the managed host. This includes additional information such as the partitions on each device, or each device's total size.

The following playbook gathers and displays the `ansible_facts['devices']` fact for each managed host.

```

---
- name: Display storage facts
  hosts: all

  tasks:
    - name: Display device facts
      ansible.builtin.debug:
        var: ansible_facts['devices']

```

This fact contains a dictionary of variables named for the devices on the system. Each named device variable itself has a dictionary of variables for its value, which represent information about the device. For example, if you have the /dev/sda device on your system, you can use the following Jinja2 expression (all on one line) to determine its size in bytes:

```

{{ ansible_facts['devices']['sda']['sectors'] * ansible_facts['devices']['sda']['sectorsize'] }}

```

Selected Facts from a Device Variable Dictionary

Fact	Comments
host	A string that identifies the controller to which the block device is connected.
model	A string that identifies the model of the storage device, if applicable.
partitions	A dictionary of block devices that are partitions on this device. Each dictionary variable has as its value a dictionary structured like any other device (including values for sectors, size, and so on).
sectors	The number of storage sectors the device contains.
sectorsize	The size of each sector in bytes.
size	A human-readable rough calculation of the device size.

For example, you could find the size of /dev/sda1 from the following fact:

```

ansible_facts['devices']['sda']['partitions']['sda1']['size']

```

Facts about Device Links

The `ansible_facts['device_links']` fact includes all the links available for each storage device. If you have multipath devices, you can use this to help determine which devices are alternative paths to the same storage device, or are multipath devices.

The following playbook gathers and displays the `ansible_['device_links']` fact for all managed hosts.

```
---
- name: Gather device link facts
  hosts: all

  tasks:
    - name: Display device link facts
      ansible.builtin.debug:
        var: ansible_facts['device_links']
```

Facts about Mounted File Systems

The `ansible_facts['mounts']` fact provides information about the currently mounted devices on the managed host. For each device, this includes the mounted block device, its file system's mount point, mount options, and so on.

The following playbook gathers and displays the `ansible_facts['mounts']` fact for managed hosts.

```
---
- name: Gather mounts
  hosts: all

  tasks:
    - name: Display mounts facts
      ansible.builtin.debug:
        var: ansible_facts['mounts']
```

The fact contains a list of dictionaries for each mounted file system on the managed host.

Selected Variables from the Dictionary in a Mounted File System List Item

Variable	Comments
<code>mount</code>	The directory on which this file system is mounted.
<code>device</code>	The name of the block device that is mounted.
<code>fstype</code>	The type of file system the device is formatted with (such as <code>xfs</code>).
<code>options</code>	The current mount options in effect.
<code>size_total</code>	The total size of the device.
<code>size_available</code>	How much space is free on the device.
<code>block_size</code>	The size of blocks on the file system.
<code>block_total</code>	How many blocks are in the file system.
<code>block_available</code>	How many blocks are free in the file system.
<code>inode_available</code>	How many inodes are free in the file system.

For example, you can determine the free space on the root (/) file system on each managed host with the following play:

```
- name: Print free space on / file system
hosts: all
gather_facts: true ①

tasks:
  - name: Display free space
    ansible.builtin.debug:
      msg: >
        The root file system on {{ ansible_facts['fqdn'] }} has
        {{ item['block_available'] * item['block_size'] / 1000000 }} 
        megabytes free. ②
    loop: "{{ ansible_facts['mounts'] }}"
    when: item['mount'] == '/' ④
      
```

- ① Gather the facts automatically.
- ② The math inside the second Jinja2 expression computes decimal megabytes of free space.
- ③ The loop iterates over every mounted file system in the list.
- ④ Conditionals are checked on **every** iteration of the loop. The loop is unrolled but the module is only run by the task when the conditional matches.



References

mount - Control active and configured mount points – Ansible Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/posix/mount_module.html

Roles – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html

Managing local storage using RHEL System Roles – Red Hat Documentation

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html/administration_and_configuration_tasks_using_system_roles_in_rhel/managing-local-storage-using-rhel-system-roles_assembly_updating-packages-to-enable-automation-for-the-rhel-system-roles

► Guided Exercise

Managing Storage

In this exercise, you assign a new disk as an LVM physical volume to a volume group, create logical volumes and format them with XFS file systems, and mount them immediately and automatically at boot time on your managed hosts.

Outcomes

- Use the `redhat.rhel_system_roles.storage` role to manage LVM volume groups and volumes.
- Use the `redhat.rhel_system_roles.storage` role to create file systems.
- Use the `redhat.rhel_system_roles.storage` role to control and configure mount points in `/etc/fstab`.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start system-storage
```

Instructions

You are responsible for managing a set of web servers. You need to configure them to store website data on a separate file system from the configuration and log files for the web servers. This is a recommended practice.

Write a playbook to:

- Use the `/dev/vdb` device as an LVM physical volume, contributing space to the volume group `apache-vg`.
 - Create two logical volumes named `content-lv` (64 MB in size) and `logs-lv` (128 MB in size), both backed by the `apache-vg` volume group.
 - Create an XFS file system on both logical volumes.
 - Mount the `content-lv` logical volume on the `/var/www` directory.
 - Mount the `logs-lv` logical volume on the `/var/log/httpd` directory.
- 1. Change into the `/home/student/system-storage` project directory.

```
[student@workstation ~]$ cd ~/system-storage  
[student@workstation system-storage]$
```

- 2. Install the `rhel_system_roles` collection from the `redhat-rhel_system_roles-1.19.3.tar.gz` file into the `collections` directory in the project directory.
- 2.1. Use the `ansible-galaxy` command to install the `rhel_system_roles` collection from the `redhat-rhel_system_roles-1.19.3.tar.gz` file into the `collections` directory.

```
[student@workstation system-storage]$ ansible-galaxy collection install \
> ./redhat-rhel_system_roles-1.19.3.tar.gz -p collections
...output omitted...
redhat.rhel_system_roles:1.19.3 was installed successfully
```

- 3. Create the `storage.yml` playbook. Write a play in that playbook that runs the `redhat.rhel_system_roles.storage` system role on the managed hosts in the `webservers` group. That play must configure the LVM physical volume, volume group, and logical volumes for the web servers.
- 3.1. Create the `storage.yml` playbook that targets the `webservers` group and applies the `redhat.rhel_system_roles.storage` role.

```
---
- name: Configure storage on webservers
  hosts: webservers

  roles:
    - name: redhat.rhel_system_roles.storage
```

- 3.2. Define the `storage_pools` role variable for the `redhat.rhel_system_roles.storage` role. Use it to set the volume group name to `apache-vg`, the type to `lvm`, and specify that the disks use the `/dev/vdb` device.

```
storage_pools:
  - name: apache-vg
    type: lvm
    disks:
      - /dev/vdb
```

- 3.3. Define the `volumes` variable within the `storage_pools` role variable.

```
volumes:
```

- 3.4. Create a logical volume within `volumes` with the name `content-lv`, a size of 64 MB, a file system type of `xfs`, and a mount point of `/var/www`.

```
- name: content-lv
  size: 64m
  mount_point: "/var/www"
  fs_type: xfs
  state: present
```

- 3.5. Create another logical volume within `volumes` with the name `logs-lv`, a size of 128 MB, a file system type of `xfs`, and a mount point of `/var/log/httpd`.

```
- name: logs-lv
  size: 128m
  mount_point: "/var/log/httpd"
  fs_type: xfs
  state: present
```

The final playbook should consist of the following content:

```
---
- name: Configure storage on webservers
  hosts: webservers

  roles:
    - name: redhat.rhel_system_roles.storage
      storage_pools:
        - name: apache-vg
          type: lvm
          disks:
            - /dev/vdb
      volumes:
        - name: content-lv
          size: 64m
          mount_point: "/var/www"
          fs_type: xfs
          state: present
        - name: logs-lv
          size: 128m
          mount_point: "/var/log/httpd"
          fs_type: xfs
          state: present
```

► 4. Run the `storage.yml` playbook.

```
[student@workstation system-storage]$ ansible-navigator run \
> -m stdout storage.yml

PLAY [Configure storage on webservers] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

...output omitted...

TASK [rhel-system-roles.storage : make sure blivet is available] ****
changed: [servera.lab.example.com]

...output omitted...

TASK [rhel-system-roles.storage : set up new/current mounts] ****
```

```

changed: [servera.lab.example.com] => (item={'src': '/dev/mapper/apache--vg-
content--lv', 'path': '/var/www', 'fstype': 'xfs', 'opts': 'defaults', 'dump': 0,
'passno': 0, 'state': 'mounted'})
changed: [servera.lab.example.com] => (item={'src': '/dev/mapper/apache--vg-logs--lv',
'path': '/var/log/httpd', 'fstype': 'xfs', 'opts': 'defaults', 'dump': 0,
'passno': 0, 'state': 'mounted'})

...output omitted...

PLAY RECAP ****
servera.lab.example.com : ok=21    changed=3    unreachable=0    failed=0
skipped=12   rescued=0   ignored=0

```

- ▶ 5. Run the `get-storage.yml` playbook provided in the project directory to verify that the storage has been properly configured on the managed hosts in the `webservers` group. If it has, then information about the storage appears in the output of the playbook as highlighted in the following example:

```

[student@workstation system-storage]$ ansible-navigator run \
> -m stdout get-storage.yml

PLAY [View storage configuration] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Retrieve physical volumes] ****
changed: [servera.lab.example.com]

TASK [Display physical volumes] ****
ok: [servera.lab.example.com] => {
    "msg": [
        " PV          VG          Fmt  Attr PSize    PFree  ",
        " /dev/vdb    apache-vg  lvm2  a--  1020.00m  828.00m"
    ]
}

TASK [Retrieve volume groups] ****
changed: [servera.lab.example.com]

TASK [Display volume groups] ****
ok: [servera.lab.example.com] => {
    "msg": [
        " VG          #PV #LV #SN Attr  VSize    VFree  ",
        " apache-vg   1   2   0 wz--n- 1020.00m 828.00m"
    ]
}

TASK [Retrieve logical volumes] ****
changed: [servera.lab.example.com]

TASK [Display logical volumes] ****
ok: [servera.lab.example.com] => {
    "msg": [

```

```

    " LV      VG      Attr     LSize   Pool Origin Data%  Meta%  Move
Log Cpy%Sync Convert",
    " content-lv apache-vg -wi-ao---- 64.00m
    ",
    " logs-lv   apache-vg -wi-ao---- 128.00m
    "
]

}

TASK [Retrieve mounted logical volumes] *****
changed: [servera.lab.example.com]

TASK [Display mounted logical volumes] *****
ok: [servera.lab.example.com] => {
  "msg": [
    "/dev/mapper/apache--vg-content--lv on /var/www type xfs
(rw,relatime,seclabel,attr2,inode64,logbufs=8,logbsize=32k,noquota)",
    "/dev/mapper/apache--vg-logs--lv on /var/log/httpd type xfs
(rw,relatime,seclabel,attr2,inode64,logbufs=8,logbsize=32k,noquota)"
  ]
}

TASK [Retrieve /etc/fstab contents] *****
changed: [servera.lab.example.com]

TASK [Display /etc/fstab contents] *****
ok: [servera.lab.example.com] => {
  "msg": [
    "UUID=5e75a2b9-1367-4cc8-bb38-4d6abc3964b8\t/boot\txfs\tdefaults\t0\t0",
    "UUID=fb535add-9799-4a27-b8bc-e8259f39a767\t\txfs\tdefaults\t0\t0",
    "UUID=7B77-95E7\t/boot/efi\tvfat
\tdefaults,uid=0,gid=0,umask=077,shortname=winnt\t0\t2",
    "/dev/mapper/apache--vg-content--lv /var/www xfs defaults 0 0",
    "/dev/mapper/apache--vg-logs--lv /var/log/httpd xfs defaults 0 0"
  ]
}

PLAY RECAP *****
servera.lab.example.com : ok=11    changed=5      unreachable=0    failed=0
skipped=0    rescued=0    ignored=0

```

Finish

On the workstation machine, change to the student user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-storage
```

This concludes the section.

Managing Network Configuration

Objectives

- Configure network settings and name resolution on managed hosts, and collect network-related Ansible facts.

Configuring Networking with the Network System Role

The `redhat.rhel_system_roles.network` system role provides a way to automate the configuration of network interfaces and network-related settings on Red Hat Enterprise Linux managed hosts.

This role supports the configuration of Ethernet interfaces, bridge interfaces, bonded interfaces, VLAN interfaces, MACVLAN interfaces, InfiniBand interfaces, and wireless interfaces.

The role is configured by using two variables: `network_provider` and `network_connections`.

```
---
network_provider: nm
network_connections:
  - name: ens4
    type: ethernet
    ip:
      address:
        - 172.25.250.30/24
```

The `network_provider` variable configures the back-end provider, either `nm` (NetworkManager) or `initscripts`. On Red Hat Enterprise Linux 7 and later, use `nm` as the default networking provider.

If you still have Red Hat Enterprise Linux 6 systems in Extended Lifecycle Support (ELS), you must use the `initscripts` provider for those managed hosts. That provider requires that the legacy `network` service is available on the managed hosts.

The `network_connections` variable configures the different connections. It takes as a value a list of dictionaries, each of which represents settings for a specific connection. Use the interface name as the connection name.

The following table lists the options for the `network_connections` variable.

Selected Options for the `network_connections` Variable

Option name	Description
<code>name</code>	For NetworkManager, identifies the connection profile (the <code>connection.id</code> option). For <code>initscripts</code> , identifies the configuration file name (<code>/etc/sysconfig/network-scripts/ifcfg-name</code>).

Option name	Description
state	The runtime state of a connection profile. Either up, if the connection profile is active, or down if it is not.
persistent_state	Identifies if a connection profile is persistent. Either present if the connection profile is persistent (the default), or absent if it is not.
type	Identifies the connection type. Valid values are ethernet, bridge, bond, team, vlan, macvlan, infiniband, and wireless.
autoconnect	Determines if the connection automatically starts. Set to yes by default.
mac	Restricts the connection to be used on devices with this specific MAC address.
interface_name	Restricts the connection profile to be used by a specific interface.
zone	Configures the firewalld zone for the interface.
ip	Determines the IP configuration for the connection. Supports the options address to specify a list of static IPv4 or IPv6 addresses on the interface, gateway4 or gateway6 to specify the IPv4 or IPv6 default router, and dns to configure a list of DNS servers.

The ip variable in turn takes a dictionary of variables for its settings. Not all of these need to be used. A connection might just have an address setting with a single IPv4 address, or it might skip the address setting and have dhcp4: yes set to enable DHCPv4 addressing.

Selected Options for the ip Variable

Option name	Description
address	A list of static IPv4 or IPv6 addresses and netmask prefixes for the connection.
gateway4	Sets a static address of the default IPv4 router.
gateway6	Sets a static address of the default IPv6 router.
dns	A list of DNS name servers for the connection.
dhcp4	Use DHCPv4 to configure the interface.
auto6	Use IPv6 autoconfiguration to configure the interface.

This is a minimal example network_connections variable to configure and immediately activate a static IPv4 address for the enp1s0 interface:

```
network_connections:
- name: enp1s0
  type: ethernet
  ip:
    address:
      - 192.0.2.25/24
  state: up
```

If you were dynamically configuring the interface using DHCP and SLAAC, you might use the following settings instead:

```
network_connections:
- name: enp1s0
  type: ethernet
  ip:
    dhcp4: true
    auto6: true
  state: up
```

The next example temporarily deactivates an existing network interface:

```
network_connections:
- name: enp1s0
  type: ethernet
  state: down
```

To delete the configuration for enp1s0 entirely, you would write the variable as follows:

```
network_connections:
- name: enp1s0
  type: ethernet
  state: down
  persistent_state: absent
```

The following example uses some of these options to set up the interface `eth0` with a static IPv4 address, set a static DNS name server, and place the interface in the `external` zone for `firewalld`:

```
network_connections:
- name: eth0 1
  persistent_state: present 2
  type: ethernet 3
  autoconnect: yes 4
  mac: 00:00:5e:00:53:5d 5
  ip:
    address:
      - 172.25.250.40/24 6
    dns:
      - 8.8.8.8 7
  zone: external 8
```

- ➊ Use `eth0` as the connection name.
- ➋ Make the connection persistent. This is the default value.
- ➌ Set the connection type to `ether`.
- ➍ Automatically start the connection at boot. This is the default value.
- ➎ Restrict the connection so that it can only be used on a device with that MAC address.
- ➏ Configure the `172.25.250.40/24` IP address for the connection.
- ➐ Configure `8.8.8.8` as the DNS name server for the connection.
- ➑ Configure the `external` zone as the `firewalld` zone of the connection.

The following example play sets `network_connections` as a play variable and then calls the `redhat.rhel_system_roles.network` role:

```
- name: NIC Configuration
hosts: webservers
vars:
  network_connections:
    - name: ens4
      type: ether
      ip:
        address:
          - 172.25.250.30/24
roles:
  - redhat.rhel_system_roles.network
```

You can specify variables for the network role with the `vars` clause, as in the previous example, as role variables. Alternatively, you can create a YAML file with those variables under the `group_vars` or `host_vars` directories, depending on your use case.

You can use this role to set up 802.11 wireless connections, VLANs, bridges, and other more complex network configurations. See the role's documentation for more details and examples.

Configuring Networking with Modules

In addition to the `redhat.rhel_system_roles.network` system role, Ansible includes modules that support the configuration of the hostname and firewall on a system.

The `ansible.builtin.hostname` module sets the hostname for a managed host without modifying the `/etc/hosts` file. This module uses the `name` parameter to specify the new hostname, as in the following task:

```
- name: Change hostname
ansible.builtin.hostname:
  name: managedhost1
```

The `ansible.posix.firewalld` module supports the management of `firewalld` on managed hosts.

This module supports the configuration of `firewalld` rules for services and ports. It also supports the zone management, including the association or network interfaces and rules to a specific zone.

The following task shows how to create a `firewalld` rule for the `http` service on the default zone (`public`). The following task configures the rule as permanent, and makes sure it is active:

```
- name: Enabling http rule
ansible.posix.firewalld:
  service: http
  permanent: yes
  state: enabled
```

This following task configures the `eth0` in the external `firewalld` zone:

```
- name: Moving eth0 to external
ansible.posix.firewalld:
  zone: external
  interface: eth0
  permanent: yes
  state: enabled
```

The following table lists some parameters for the `ansible.posix.firewalld` module.

Parameter name	Description
<code>interface</code>	Interface name to manage with <code>firewalld</code> .
<code>port</code>	Port or port range. Uses the port/protocol or port-port/protocol format.
<code>rich_rule</code>	Rich rule for <code>firewalld</code> .
<code>service</code>	Service name to manage with <code>firewalld</code> .
<code>source</code>	Source network to manage with <code>firewalld</code> .
<code>zone</code>	<code>firewalld</code> zone.
<code>state</code>	Enable or disable a <code>firewalld</code> configuration.
<code>type</code>	Type of device or network connection.
<code>permanent</code>	Change persists across reboots.
<code>immediate</code>	If the changes are set to <code>permanent</code> , then apply them immediately.

Ansible Facts for Network Configuration

Ansible collects a number of facts that are related to each managed host's network configuration. For example, a list of the network interfaces on a managed host are available in the `ansible_facts['interfaces']` fact.

The following playbook gathers and displays the available interfaces for a host:

```
---
- name: Obtain interface facts
  hosts: host.lab.example.com

  tasks:
    - name: Display interface facts
      ansible.builtin.debug:
        var: ansible_facts['interfaces']
```

The preceding playbook produces the following list of the network interfaces:

```
PLAY [Obtain interface facts] ****
TASK [Gathering Facts] ****
ok: [host.lab.example.com]

TASK [Display interface facts] ****
ok: [host.lab.example.com] => {
  "ansible_facts['interfaces']": [
    "eth2",
    "eth1",
    "eth0",
    "lo"
  ]
}

PLAY RECAP ****
host.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
                      skipped=0   rescued=0   ignored=0
```

The output in the previous example shows that four network interfaces are available on the `host.lab.example.com` managed host: `lo`, `eth2`, `eth1`, and `eth0`.

You can retrieve additional information about the configuration for a specific network interface from the `ansible_facts['NIC_name']` fact. For example, the following play displays the configuration for the `eth0` network interface by printing the value of the `ansible_facts['eth0']` fact.

```
- name: Obtain eth0 facts
  hosts: host.lab.example.com

  tasks:
    - name: Display eth0 facts
      ansible.builtin.debug:
        var: ansible_facts['eth0']
```

The preceding playbook produces the following output:

```
PLAY [Obtain eth0 facts] ****
TASK [Gathering Facts] ****
ok: [host.lab.example.com]
```

```

TASK [Display eth0 facts] ****
ok: [host.lab.example.com] => {
    "ansible_facts['eth0']": {
        "active": true,
        "device": "eth0",
        "features": {
            ...output omitted...
        },
        "hw_timestamp_filters": [],
        "ipv4": {
            "address": "172.25.250.10",
            "broadcast": "172.25.250.255",
            "netmask": "255.255.255.0",
            "network": "172.25.250.0",
            "prefix": "24"
        },
        "ipv6": [
            {
                "address": "fe80::82a0:2335:d88a:d08f",
                "prefix": "64",
                "scope": "link"
            }
        ],
        "macaddress": "52:54:00:00:fa:0a",
        "module": "virtio_net",
        "mtu": 1500,
        "pciid": "virtio0",
        "promisc": false,
        "speed": -1,
        "timestamping": [],
        "type": "ether"
    }
}

PLAY RECAP ****
host.lab.example.com      : ok=2      changed=0      unreachable=0      failed=0
                           skipped=0     rescued=0     ignored=0

```

The preceding output displays additional configuration details, such as the IP address configuration both for IPv4 and IPv6, the associated device, and the type of interface.



Important

Different managed hosts might have different network interface names, depending on their hardware and software configuration. Ansible facts can help you identify differences between systems so that you can address or mitigate them in your automation.

The following table lists some other useful network-related facts.

Fact name	Description
<code>ansible_facts['dns']</code>	A list of the DNS name server IP addresses and the search domains.
<code>ansible_facts['domain']</code>	The subdomain for the managed host.
<code>ansible_facts['all_ipv4_addresses']</code>	All the IPv4 addresses configured on the managed host.
<code>ansible_facts['all_ipv6_addresses']</code>	All the IPv6 addresses configured on the managed host.
<code>ansible_facts['fqdn']</code>	The fully qualified domain name (FQDN) of the managed host.
<code>ansible_facts['hostname']</code>	The unqualified hostname (the part of the hostname before the first period in the FQDN).
<code>ansible_facts['nodename']</code>	The hostname of the managed host as reported by the system.



Note

Ansible also provides the `inventory_hostname` "magic variable" which includes the hostname as configured in the Ansible inventory file.



References

Knowledgebase: Red Hat Enterprise Linux (RHEL) System Roles

<https://access.redhat.com/articles/3050101>

Linux System Roles

<https://linux-system-roles.github.io/>

linux-system-roles/network at GitHub

<https://github.com/linux-system-roles/network>

ansible.builtin.hostname Module Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/hostname_module.html

ansible.posix.firewalld Module Documentation

https://docs.ansible.com/ansible/latest/collections/ansible/posix/firewalld_module.html

► Guided Exercise

Managing Network Configuration

In this exercise, you adjust the network configuration of a managed host and collect information about it in a file created by a template.

Outcomes

- You should be able to configure network settings on managed hosts, and collect network-related Ansible facts.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start system-network
```

Instructions

- 1. Review the inventory file in the `/home/student/system-network` project directory.

- 1.1. Change into the `/home/student/system-network` directory.

```
[student@workstation ~]$ cd ~/system-network  
[student@workstation system-network]$
```

- 1.2. In the `inventory` file, verify that `servera.lab.example.com` is part of the `webservers` host group. That server has a spare network interface.

```
[student@workstation system-network]$ cat inventory  
[webservers]  
servera.lab.example.com
```

- 2. Install the `redhat.rhel_system_roles` Ansible Content Collection from the `redhat-rhel_system_roles-1.19.3.tar.gz` file to the `collections` directory in the project directory.

- 2.1. Use the `ansible-galaxy` command to install the `rhel_system_roles` Ansible Content Collection from the `redhat-rhel_system_roles-1.19.3.tar.gz` file into the project's `collections` directory.

```
[student@workstation system-network]$ ansible-galaxy collection install \
> ./redhat-rhel_system_roles-1.19.3.tar.gz -p collections
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Installing 'redhat.rhel_system_roles:1.19.3' to '/home/student/system-network/
collections/ansible_collections/redhat/rhel_system_roles'
redhat.rhel_system_roles:1.19.3 was installed successfully
```

- ▶ 3. Create a playbook that uses the `redhat.rhel_system_roles.network` role to configure the network interface `eth1` on `servera.lab.example.com` with the `172.25.250.30/24` IP address and network prefix.
- 3.1. Create a playbook named `playbook.yml`, and add one play that targets the `webservers` host group. Include the `rhel_system_roles.network` role in the `roles` section of the play.

```
---
- name: NIC Configuration
  hosts: webservers

  roles:
    - redhat.rhel_system_roles.network
```

- 3.2. The documentation for the `redhat.rhel_system_roles.network` system role lists the variables and options that are available to configure its operation.

Run `ansible-navigator collections`, select the `redhat.rhel_system_roles` collection, and then select the `network` role to see its documentation.

Review the `Setting the IP configuration:` section (near line 1219 of the documentation) and determine which variable is required to configure the `eth1` network interface with a static IP address and network prefix.

For your convenience, that part of the documentation reads as follows:

```
...output omitted...

Setting the IP configuration:

```yaml
network_connections:
 - name: eth0
 type: ethernet
 ip:
 route_metric4: 100
 dhcp4: no
 #dhcp4_send_hostname: no
 gateway4: 192.0.2.1

 dns:
 - 192.0.2.2
 - 198.51.100.5
```
}
```

```

dns_search:
  - example.com
  - subdomain.example.com
dns_options:
  - rotate
  - timeout:1

route_metric6: -1
auto6: no
gateway6: 2001:db8::1

address:
  - 192.0.2.3/24
  - 198.51.100.3/26
  - 2001:db8::80/7

...output omitted...

```

3.3. Create the group_vars/webservers subdirectory.

```
[student@workstation system-network]$ mkdir -pv group_vars/webservers
mkdir: created directory 'group_vars'
mkdir: created directory 'group_vars/webservers'
```

- 3.4. Create a new variable file named `network.yml` in the `group_vars/webservers` directory to define the `network_connections` role variable for the `webservers` group.

The value of that variable must configure a network connection for the `eth1` network interface that assigns it the static IP address and network prefix `172.25.250.30/24`.

When completed, the file contains the following content:

```
[student@workstation system-network]$ cat group_vars/webservers/network.yml
---
network_connections:
  - name: eth1
    type: ethernet
    ip:
      address:
        - 172.25.250.30/24
```

3.5. Run the playbook to configure the secondary network interface on servera.

```
[student@workstation system-network]$ ansible-navigator run \
> -m stdout playbook.yml

PLAY [NIC Configuration] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Ensure ansible_facts used by role] ****
```

```
included: /home/student/system-network/collections/ansible_collections/redhat/
rhel_system_roles/roles/network/tasks/set_facts.yml for servera.lab.example.com

TASK [redhat.rhel_system_roles.network : Ensure ansible_facts used by role are
present] ***
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Check which services are running] *****
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Check which packages are installed] ***
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Print network provider] ****
ok: [servera.lab.example.com] => {
    "msg": "Using network provider: nm"
}

TASK [redhat.rhel_system_roles.network : Abort applying the network state
configuration if using the network_state variable with the initscripts provider]
***  

skipping: [servera.lab.example.com]

...output omitted...

TASK [redhat.rhel_system_roles.network : Configure networking connection profiles]
***  

changed: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Configure networking state] ****
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Show stderr messages for the
network_connections] ***
ok: [servera.lab.example.com] => {
    "__network_connections_result.stderr_lines": [
        "[002] <info> #0, state:None persistent_state:present, 'eth1': add
connection eth1, c0177289-f461-4042-b9d4-86fd1b235be1"
    ]
}

TASK [redhat.rhel_system_roles.network : Show debug messages for the
network_connections] ***
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Show debug messages for the
network_state] ***
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Re-test connectivity] ****
```

```
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=10    changed=1    unreachable=0    failed=0
skipped=12   rescued=0   ignored=0
```

- ▶ 4. Run the `get-eth1.yml` playbook provided in the project directory to verify that the `eth1` network interface configuration on `servera` is correct.

Verify that the `eth1` network interface uses the `172.25.250.30` IP address with the `/24` network prefix (equivalent to the `255.255.255.0` subnet mask). It might take up to a minute to configure the IP address.

```
[student@workstation system-network]$ ansible-navigator run \
> -m stdout get-eth1.yml

PLAY [Obtain network info for webservers] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Display eth1 info] ****
ok: [servera.lab.example.com] => {
    "ansible_facts['eth1']['ipv4']": {
        "address": "172.25.250.30",
        "broadcast": "172.25.250.255",
        "netmask": "255.255.255.0",
        "network": "172.25.250.0",
        "prefix": "24"
    }
}

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
skipped=0   rescued=0   ignored=0
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-network
```

This concludes the section.

▶ Lab

Automating Linux Administration Tasks

In this lab, you perform common Linux administrative tasks on your managed hosts, using techniques that were covered in this chapter.

Outcomes

- Create playbooks for configuring a software repository, users and groups, logical volumes, cron jobs, and additional network interfaces on a managed host.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start system-review
```

Instructions

1. Create a playbook named `repo_playbook.yml` to run on the `webservers` host group that configures those managed hosts to use the Yum internal repository located at `http://materials.example.com/yum/repository`, and then installs the `rhelver` package available from that repository on those managed hosts.

The repository configuration must satisfy the following requirements:

- The configuration is in the `/etc/yum.repos.d/example.repo` file.
- The repository ID is `example-internal`.
- The base URL is `http://materials.example.com/yum/repository`.
- The repository is configured to check RPM GPG signatures.
- The repository description is `Example Inc. Internal YUM repo`.

All RPM packages in that repository are signed with an organizational GPG key pair. That GPG public key is available at `http://materials.example.com/yum/repository/RPM-GPG-KEY-example`. You need to make sure that the key is configured on all managed hosts in the `webservers` host group.

Run the playbook. You should confirm that the playbook installed the package on your managed hosts after it runs.

2. Create a playbook named `users.yml` to run on the `webservers` host group that creates the `webadmin` user group, adds the `ops1` and `ops2` users, and ensures that both users have `webadmin` as a supplementary group on those managed hosts.

Run the playbook. You should confirm that the users exist on the managed hosts and have `webadmin` as a supplementary group after the playbook runs.

3. Install the `redhat-rhel_system_roles-1.19.3.tar.gz` Ansible Content Collection provided in the project directory to the `collections` directory.

Create a playbook named `storage.yml` to run on the `webservers` host group and configure those managed hosts by using the `redhat.rhel_system_roles.storage` system role, as follows:

- Uses the `/dev/vdb` device as an LVM physical volume for the `apache-vg` volume group.
- Creates the `content-lv` logical volume, 64 MB in size, in the `apache-vg` volume group.
- Creates the `logs-lv` logical volume, 128 MB in size, in the `apache-vg` volume group.
- Formats each logical volume with an XFS file system.
- Mounts the `content-lv` logical volume on the `/var/www` directory.
- Mounts the `logs-lv` logical volume on the `/var/log/httpd` directory.

Run the playbook. Confirm that the playbook ran correctly after you run it.

4. Create a playbook named `create_crontab_file.yml` to run on the `webservers` host group that uses the `ansible.builtin.cron` module to create a system crontab file that schedules a recurring Cron job. Create the `/etc/cron.d/disk_usage` file as the system crontab file. Configure that file's system Cron job as follows:

- It must run as the `devops` user.
- It must run every two minutes from 9:00 to 16:59 on Monday through Friday.
- It must run the command `df >> /home/devops/disk_usage`.

Run the playbook. You should confirm that it set up the Cron job correctly after running the playbook. You could look to see if the file deployed correctly, or inspect the `/var/log/cron` log file as `root` to see if the job is running.

5. Create a playbook named `network_playbook.yml` to run on the `webservers` host group that uses the `redhat.rhel_system_roles.network` role to configure the network interface `eth1` with the `172.25.250.40/24` IP address.

Run the playbook. Confirm that the playbook ran correctly.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade system-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-review
```

This concludes the section.

► Solution

Automating Linux Administration Tasks

In this lab, you perform common Linux administrative tasks on your managed hosts, using techniques that were covered in this chapter.

Outcomes

- Create playbooks for configuring a software repository, users and groups, logical volumes, cron jobs, and additional network interfaces on a managed host.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start system-review
```

Instructions

1. Create a playbook named `repo_playbook.yml` to run on the `webservers` host group that configures those managed hosts to use the Yum internal repository located at `http://materials.example.com/yum/repository`, and then installs the `rhelver` package available from that repository on those managed hosts.

The repository configuration must satisfy the following requirements:

- The configuration is in the `/etc/yum.repos.d/example.repo` file.
- The repository ID is `example-internal`.
- The base URL is `http://materials.example.com/yum/repository`.
- The repository is configured to check RPM GPG signatures.
- The repository description is `Example Inc. Internal YUM repo`.

All RPM packages in that repository are signed with an organizational GPG key pair. That GPG public key is available at `http://materials.example.com/yum/repository/RPM-GPG-KEY-example`. You need to make sure that the key is configured on all managed hosts in the `webservers` host group.

Run the playbook. You should confirm that the playbook installed the package on your managed hosts after it runs.

- 1.1. Change into the `/home/student/system-review` directory.

```
[student@workstation ~]$ cd ~/system-review  
[student@workstation system-review]$
```

- 1.2. Create the `repo_playbook.yml` playbook, which runs on the managed hosts in the `webservers` host group.

Add a task that uses the `ansible.builtin.yum_repository` module to ensure the correct configuration of the internal Yum repository on the remote host.

The configuration must satisfy the following requirements:

- The configuration is stored in the `/etc/yum.repos.d/example.repo` file.
- The repository ID is `example-internal`.
- The base URL is `http://materials.example.com/yum/repository`.
- The repository is configured to check RPM GPG signatures.
- The repository description is `Example Inc. Internal YUM repo`.

The playbook contains the following content:

```
---
- name: Repository Configuration
  hosts: webservers
  tasks:
    - name: Ensure Example Repo exists
      ansible.builtin.yum_repository:
        name: example-internal
        description: Example Inc. Internal YUM repo
        file: example
        baseurl: http://materials.example.com/yum/repository/
        gpgcheck: yes
```

- 1.3. Add a second task to the play that uses the `ansible.builtin.rpm_key` module to ensure that the repository public key is present on the remote host. The repository public key URL is `http://materials.example.com/yum/repository/RPM-GPG-KEY-example`.

The second task contains the following content:

```
- name: Ensure Repo RPM Key is Installed
  ansible.builtin.rpm_key:
    key: http://materials.example.com/yum/repository/RPM-GPG-KEY-example
    state: present
```

- 1.4. Add a third task to install the `rhelver` package available in the Yum internal repository.

The third task contains the following content:

```
- name: Install rhelver package
  ansible.builtin.dnf:
    name: rhelver
    state: present
```

- 1.5. Run the `repo_playbook.yml` playbook:

```
[student@workstation system-review]$ ansible-navigator run \
> -m stdout repo_playbook.yml

PLAY [Repository Configuration] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
```

```

TASK [Ensure Example Repo exists] ****
changed: [serverb.lab.example.com]

TASK [Ensure Repo RPM Key is Installed] ****
changed: [serverb.lab.example.com]

TASK [Install rhelver package] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com      : ok=4      changed=3      unreachable=0      failed=0
skipped=0      rescued=0      ignored=0

```

2. Create a playbook named `users.yml` to run on the `webservers` host group that creates the `webadmin` user group, adds the `ops1` and `ops2` users, and ensures that both users have `webadmin` as a supplementary group on those managed hosts.

Run the playbook. You should confirm that the users exist on the managed hosts and have `webadmin` as a supplementary group after the playbook runs.

- 2.1. Create a `vars/users_vars.yml` variable file, which defines two users, `ops1` and `ops2`, which belong to the `webadmin` user group.

You might need to create the `vars` subdirectory.

```

[student@workstation system-review]$ mkdir vars
[student@workstation system-review]$ vim vars/users_vars.yml
---
users:
  - username: ops1
    groups: webadmin
  - username: ops2
    groups: webadmin

```

- 2.2. Create the `users.yml` playbook. Define a single play in the playbook that targets the `webservers` host group.

Add a `vars_files` clause that defines the location of the `vars/users_vars.yml` file.

Add a task that uses the `group` module to create the `webadmin` user group on the remote host.

The playbook contains the following content:

```

---
- name: Create multiple local users
hosts: webservers
vars_files:
  - vars/users_vars.yml

tasks:
  - name: Add webadmin group
    ansible.builtin.group:
      name: webadmin
      state: present

```

- 2.3. Add a second task to the playbook that uses the `ansible.builtin.user` module to create the users.

Add a `loop: "{{ users }}"` clause to the task to loop through the variable file for every username found in the `vars/users_vars.yml` file.

Use the `item['username']` variable for the `name:` value. The variable file might contain additional information useful for creating the users, such as the groups that the users should belong to.

The second task contains the following content:

```
- name: Create user accounts
  ansible.builtin.user:
    name: "{{ item['username'] }}"
    groups: webadmin
  loop: "{{ users }}"
```

- 2.4. Run the `users.yml` playbook:

```
[student@workstation system-review]$ ansible-navigator run \
> -m stdout users.yml

PLAY [Create multiple local users] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Add webadmin group] ****
changed: [serverb.lab.example.com]

TASK [Create user accounts] ****
changed: [serverb.lab.example.com] => (item={'username': 'ops1', 'groups': 'webadmin'})
changed: [serverb.lab.example.com] => (item={'username': 'ops2', 'groups': 'webadmin'})

PLAY RECAP ****
serverb.lab.example.com      : ok=3      changed=2      unreachable=0      failed=0
                               skipped=0     rescued=0     ignored=0
```

3. Install the `redhat-rhel_system_roles-1.19.3.tar.gz` Ansible Content Collection provided in the project directory to the `collections` directory.

Create a playbook named `storage.yml` to run on the `webservers` host group and configure those managed hosts by using the `redhat.rhel_system_roles.storage` system role, as follows:

- Uses the `/dev/vdb` device as an LVM physical volume for the `apache-vg` volume group.
- Creates the `content-lv` logical volume, 64 MB in size, in the `apache-vg` volume group.
- Creates the `logs-lv` logical volume, 128 MB in size, in the `apache-vg` volume group.
- Formats each logical volume with an XFS file system.
- Mounts the `content-lv` logical volume on the `/var/www` directory.
- Mounts the `logs-lv` logical volume on the `/var/log/httpd` directory.

Run the playbook. Confirm that the playbook ran correctly after you run it.

- 3.1. Install the `redhat-rhel_system_roles` collection from the `~/system-review/redhat-rhel_system_roles-1.19.3.tar.gz` file into the `~/system-review/collections` directory.

```
[student@workstation system-review]$ ansible-galaxy collection \
> install ./redhat-rhel_system_roles-1.19.3.tar.gz -p collections
```

- 3.2. Create the `group_vars/webservers` subdirectory.

```
[student@workstation system-review]$ mkdir -pv group_vars/webservers
mkdir: created directory 'group_vars'
mkdir: created directory 'group_vars/webservers'
```

- 3.3. Create the `~/system-review/group_vars/webservers/storage_vars.yml` variables file.

In the variable file, define a `storage_pool` variable with the pool name `apache-vg` for the volume group on the `/dev/vdb` device, and with the type set to `lvm`.

Within the `apache-vg` pool define two logical volumes:

- Define the `content-lv` logical volume with a size of 64 MB formatted with the XFS file system, mounted at `/var/www`.
- Define the `logs-lv` logical volume with a size of 128 MB formatted with the XFS file system, mounted at `/var/log/httpd`.

When completed, the `~/system-review/group_vars/webservers/storage_vars.yml` variables file should contain the following content:

```
---
storage_pools:
  - name: apache-vg
    type: lvm
    disks:
      - /dev/vdb
    volumes:
      - name: content-lv
        size: 64m
        mount_point: "/var/www"
        fs_type: xfs
        state: present
      - name: logs-lv
        size: 128m
        mount_point: "/var/log/httpd"
        fs_type: xfs
        state: present
```

- 3.4. Create the `storage.yml` playbook to apply the `redhat.rhel_system_roles.storage` role to the `webservers` host group.

When completed, the playbook should contain the following content:

```
---
- name: Configure storage on webservers
  hosts: webservers

  roles:
    - name: redhat.rhel_system_roles.storage
```

3.5. Run the `storage.yml` playbook.

```
[student@workstation system-review]$ ansible-navigator run \
> -m stdout storage.yml

PLAY [Configure storage on webservers] ****
...output omitted...

TASK [redhat.rhel_system_roles.storage : make sure blivet is available] ****
changed: [serverb.lab.example.com]

...output omitted...

TASK [redhat.rhel_system_roles.storage : manage the pools and volumes to match the
specified state] ***
changed: [serverb.lab.example.com]

...output omitted...

TASK [redhat.rhel_system_roles.storage : set up new/current mounts] ****
changed: [serverb.lab.example.com] => (item={'src': '/dev/mapper/apache--vg-
content--lv', 'path': '/var/www', 'fstype': 'xfs', 'opts': 'defaults', 'dump': 0,
'passno': 0, 'state': 'mounted'})
changed: [serverb.lab.example.com] => (item={'src': '/dev/mapper/apache--vg-logs--lv',
'path': '/var/log/httpd', 'fstype': 'xfs', 'opts': 'defaults', 'dump': 0,
'passno': 0, 'state': 'mounted'})

...output omitted...

PLAY RECAP ****
serverb.lab.example.com      : ok=21    changed=3     unreachable=0    failed=0
skipped=12    rescued=0    ignored=0
```

- Create a playbook named `create_crontab_file.yml` to run on the `webservers` host group that uses the `ansible.builtin.cron` module to create a system crontab file that schedules a recurring Cron job. Create the `/etc/cron.d/disk_usage` file as the system crontab file. Configure that file's system Cron job as follows:
 - It must run as the `devops` user.
 - It must run every two minutes from 9:00 to 16:59 on Monday through Friday.
 - It must run the command `df >> /home/devops/disk_usage`.

Run the playbook. You should confirm that it set up the Cron job correctly after running the playbook. You could look to see if the file deployed correctly, or inspect the `/var/log/cron` log file as `root` to see if the job is running.

- 4.1. Create a new playbook named `create_crontab_file.yml`, and add the lines needed to start the play. The play should target the managed hosts in the `webservers` group and enable privilege escalation.

```
---
- name: Recurring cron job
  hosts: webservers
```

- 4.2. Define a task that uses the `ansible.builtin.cron` module to schedule a recurring Cron job.

```
tasks:
  - name: Crontab file exists
    ansible.builtin.cron:
      name: Add date and time to a file
```

- 4.3. Configure the job to run every two minutes from 09:00 through 16:59 on Monday through Friday.

```
  minute: "*/2"
  hour: 9-16
  weekday: 1-5
```

- 4.4. Use the `cron_file` parameter to use the `/etc/cron.d/disk_usage` system crontab file instead of an individual user's crontab file in `/var/spool/cron/`. Use a relative path to place the file in the `/etc/cron.d` directory. If the `cron_file` parameter is used, you must also specify the `user` parameter.

```
  user: devops
  job: df >> /home/devops/disk_usage
  cron_file: disk_usage
  state: present
```

- 4.5. When completed, the playbook should contain the following content. Review the playbook for accuracy.

```
---
- name: Recurring cron job
  hosts: webservers

  tasks:
    - name: Crontab file exists
      ansible.builtin.cron:
        name: Add date and time to a file
        minute: "*/2"
        hour: 9-16
        weekday: 1-5
        user: devops
        job: df >> /home/devops/disk_usage
        cron_file: disk_usage
        state: present
```

- 4.6. Run the `create_crontab_file.yml` playbook.

```
[student@workstation system-review]$ ansible-navigator run \
> -m stdout create_crontab_file.yml

PLAY [Recurring cron job] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Crontab file exists] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com      : ok=2      changed=1      unreachable=0      failed=0
                               skipped=0    rescued=0    ignored=0
```

5. Create a playbook named `network_playbook.yml` to run on the `webservers` host group that uses the `redhat.rhel_system_roles.network` role to configure the network interface `eth1` with the `172.25.250.40/24` IP address.

Run the playbook. Confirm that the playbook ran correctly.

- 5.1. Create a playbook named `network_playbook.yml`, with one play that targets the `webservers` host group.
Include the `redhat.rhel_system_roles.network` role in the `roles` section of the play.

```
---
- name: NIC Configuration
  hosts: webservers

  roles:
    - redhat.rhel_system_roles.network
```

- 5.2. Create a new file named `network.yml` to define role variables.

Because these variable values apply to the hosts on the `webservers` host group, you need to create that file in the `group_vars/webservers` directory.

Add variable definitions to support the configuration of the `eth1` network interface.

The variable file contains the following content:

```
[student@workstation system-review]$ vim group_vars/webservers/network.yml
---
network_connections:
  - name: eth1
    type: ethernet
    ip:
      address:
        - 172.25.250.40/24
```

- 5.3. Run the `network_playbook.yml` playbook to configure the `eth1` network interface.

```
[student@workstation system-review]$ ansible-navigator run \
> -m stdout network_playbook.yml

PLAY [NIC Configuration] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

...output omitted...

TASK [redhat.rhel_system_roles.network : Configure networking connection profiles]
*** changed: [serverb.lab.example.com]

TASK [redhat.rhel_system_roles.network : Show stderr messages] ****
ok: [serverb.lab.example.com] => {
    "__network_connections_result.stderr_lines": [
        "[002] <info> #0, state:None persistent_state:present, 'eth1': add
connection eth1, b2332ada-021d-4f1b-a228-0e342034f95e"
    ]
}

...output omitted...

PLAY RECAP ****
serverb.lab.example.com      : ok=10    changed=1    unreachable=0    failed=0
skipped=12    rescued=0    ignored=0
```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade system-review
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish system-review
```

This concludes the section.

Summary

- The `ansible.builtin.yum_repository` module configures a Yum repository on a managed host. For repositories that use public keys, you can verify that the key is available with the `ansible.builtin.rpm_key` module.
- The `ansible.builtin.user` and `ansible.builtin.group` modules create users and groups respectively on a managed host.
- The `ansible.builtin.known_hosts` module configures SSH known hosts for a server and the `ansible.posix.authorized_key` modules configures authorized keys for user authentication.
- The `ansible.builtin.cron` module configures system or user Cron jobs on managed hosts.
- The `ansible.posix.at` module configures One-off at jobs on managed hosts.
- The `redhat.rhel_system_roles` Red Hat Certified Ansible Content Collection includes two particularly useful system roles: `storage`, which supports the configuration of LVM logical volumes, and `network`, which enables the configuration of network interfaces and connections.

Chapter 10

Comprehensive Review: Red Hat Enterprise Linux Automation with Ansible

Goal

Demonstrate skills learned in this course by installing, optimizing, and configuring Ansible for the management of managed hosts.

Sections

- Comprehensive Review

Labs

- Deploying and Configuring Ansible
- Creating Playbooks
- Managing Linux Hosts and Using System Roles
- Creating Roles

Comprehensive Review

Objectives

After completing this section, you should have reviewed and refreshed the knowledge and skills learned in *Red Hat Enterprise Linux Automation with Ansible*.

Reviewing Red Hat Enterprise Linux Automation with Ansible

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, Introducing Ansible

Describe the fundamental concepts of Ansible and how it is used, and install development tools from Red Hat Ansible Automation Platform.

- Describe the motivation for automating Linux administration tasks with Ansible, fundamental Ansible concepts, and the basic architecture of Ansible.
- Install Ansible on a control node and describe the distinction between community Ansible and Red Hat Ansible Automation Platform.

Chapter 2, Implementing an Ansible Playbook

Create an inventory of managed hosts, write a simple Ansible Playbook, and run the playbook to automate tasks on those hosts.

- Describe Ansible inventory concepts and manage a static inventory file.
- Describe where Ansible configuration files are located, how Ansible selects them, and edit them to apply changes to default settings.
- Write a basic Ansible Playbook and run it using the automation content navigator.
- Write a playbook that uses multiple plays with per-play privilege escalation, and effectively use automation content navigator to find new modules in available Ansible Content Collections and use them to implement tasks for a play.

Chapter 3, Managing Variables and Facts

Write playbooks that use variables to simplify management of the playbook and facts to reference information about managed hosts.

- Create and reference variables that affect particular hosts or host groups, the play, or the global environment, and describe how variable precedence works.
- Encrypt sensitive variables using Ansible Vault, and run playbooks that reference Vault-encrypted variable files.

- Reference data about managed hosts using Ansible facts, and configure custom facts on managed hosts.

Chapter 4, Implementing Task Control

Manage task control, handlers, and task errors in Ansible Playbooks.

- Use loops to write efficient tasks and use conditions to control when to run tasks.
- Implement a task that runs only when another task changes the managed host.
- Control what happens when a task fails, and what conditions cause a task to fail.

Chapter 5, Deploying Files to Managed Hosts

Deploy, manage, and adjust files on hosts managed by Ansible.

- Create, install, edit, and remove files on managed hosts, and manage the permissions, ownership, SELinux context, and other characteristics of those files.
- Deploy files to managed hosts that are customized by using Jinja2 templates.

Chapter 6, Managing Complex Plays and Playbooks

Write playbooks for larger, more complex plays and playbooks.

- Write sophisticated host patterns to efficiently select hosts for a play.
- Manage large playbooks by importing or including other playbooks or tasks from external files, either unconditionally or based on a conditional test.

Chapter 7, Simplifying Playbooks with Roles and Ansible Content Collections

Use Ansible Roles and Ansible Content Collections to develop playbooks more quickly and to reuse Ansible code.

- Describe the purpose of an Ansible Role, its structure, and how roles are used in playbooks.
- Create a role in a playbook's project directory and run it as part of one of the plays in the playbook.
- Select and retrieve roles from external sources such as Git repositories or Ansible Galaxy, and use them in your playbooks.
- Obtain a set of related roles, supplementary modules, and other content from an Ansible Content Collection and use them in a playbook.
- Write playbooks that take advantage of system roles for Red Hat Enterprise Linux to perform standard operations.

Chapter 8, Troubleshooting Ansible

Troubleshoot playbooks and managed hosts.

- Troubleshoot generic issues with a new playbook and repair them.
- Troubleshoot failures on managed hosts when running a playbook.

Chapter 9, Automating Linux Administration Tasks

Automate common Linux system administration tasks with Ansible.

- Subscribe systems, configure software channels and repositories, enable module streams, and manage RPM packages on managed hosts.
- Manage Linux users and groups, configure SSH, and modify Sudo configuration on managed hosts.
- Manage service startup, schedule processes with at, cron, and systemd, reboot managed hosts with reboot, and control the default boot target on managed hosts.
- Partition storage devices, configure LVM, format partitions or logical volumes, mount file systems, and add swap spaces.
- Configure network settings and name resolution on managed hosts, and collect network-related Ansible facts.

▶ Lab

Deploying Ansible

In this review, you install Ansible on `workstation`, configure it as a control node, and configure an inventory for connections to the `servera.lab.example.com` and `serverb.lab.example.com` managed hosts. You create, modify, and troubleshoot simple playbooks that use variables and facts.

Outcomes

- Install and configure Ansible.
- Create, modify, and troubleshoot playbooks.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr1
```

Specifications

- Install the automation content navigator on `workstation` so that it can serve as the control node. The Yum repository containing the package has been configured on `workstation` for you.
- Your Ansible project directory is `/home/student/review-cr1`.
- On the control node, create the `/home/student/review-cr1/inventory` inventory file. The inventory must contain a group called `dev` that consists of the `servera.lab.example.com` and `serverb.lab.example.com` managed hosts.
- Create an Ansible configuration file named `/home/student/review-cr1/ansible.cfg`. This configuration file must use the `/home/student/review-cr1/inventory` file as the project inventory file.
- Log in to your private automation hub at `utility.lab.example.com` from the command line before attempting to run automation content navigator, so that you can pull automation execution environment images from its container registry. Your username is `admin` and your password is `redhat`.
- Create a configuration file for automation content navigator named `/home/student/review-cr1/ansible-navigator.yml`. This configuration file must set the default automation execution environment image to `utility.lab.example.com/ee-supported-rhel8:latest`, and automation content navigator must only pull this image from the container repository if the image is missing on your control node.

- Create a playbook named `users.yml` in the project directory. It must contain one play that runs on managed hosts in the `dev` group. Its play must use one task to add the users `joe` and `sam` to all managed hosts in the `dev` group. Run the `users.yml` playbook and confirm that it works.
- Inspect the existing `packages.yml` playbook. In the play in that playbook, define a play variable named `packages` with a list of two packages as its value: `httpd` and `mariadb-server`. Run the `packages.yml` playbook and confirm that both of those packages are installed on the managed hosts on which the playbook ran.
- Add a task to the `packages.yml` playbook that installs the `redis` package if the total swap space on the managed host is greater than 10 MB. Run the `packages.yml` playbook again after adding this task.
- Troubleshoot the existing `verify_user.yml` playbook. It is supposed to verify that the `sam` user was created successfully, and it is not supposed to create the `sam` user if it is missing. Run the playbook with the `--check` option and resolve any errors. Repeat this process until you can run the playbook with the `--check` option and it passes, and then run the `verify_user.yml` playbook normally.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr1
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr1
```

This concludes the section.

► Solution

Deploying Ansible

In this review, you install Ansible on `workstation`, configure it as a control node, and configure an inventory for connections to the `servera.lab.example.com` and `serverb.lab.example.com` managed hosts. You create, modify, and troubleshoot simple playbooks that use variables and facts.

Outcomes

- Install and configure Ansible.
- Create, modify, and troubleshoot playbooks.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr1
```

1. Install automation content navigator on `workstation` so that it can serve as the control node.

```
[student@workstation ~]$ sudo dnf install ansible-navigator
[sudo] password for student: student
Last metadata expiration check: 1:51:10 ago on Fri 26 Aug 2022 02:42:43 PM EDT.
Dependencies resolved.
...output omitted...
Is this ok [y/N]: y
...output omitted...
Complete!
```

2. On the control node, create the `/home/student/review-cr1/inventory` inventory file. It must contain a group called `dev` that consists of the `servera.lab.example.com` and `serverb.lab.example.com` managed hosts.
 - 2.1. Change into the `/home/student/review-cr1` directory.

```
[student@workstation ~]$ cd ~/review-cr1
[student@workstation review-cr1]$
```

- 2.2. Create the `inventory` file with the following content:

```
[dev]
servera.lab.example.com
serverb.lab.example.com
```

3. Create an Ansible configuration file named /home/student/review-cr1/ansible.cfg. The configuration file must use the /home/student/review-cr1/inventory file as the project inventory file.

Add the following entries to configure the ./inventory inventory file as the inventory source. Save and close the file.

```
[defaults]
inventory=./inventory
```

4. Create an automation content navigator configuration file named /home/student/review-cr1/ansible-navigator.yml. This configuration file should set the default execution environment image to utility.lab.example.com/ee-supported-rhel8:latest, and ansible-navigator should only pull this image from the container registry if the image is missing.

Make sure to log in to your private automation hub on utility.lab.example.com with the podman login command. Your username is admin and your password is redhat.

Run ansible-navigator to download the execution environment image.

- 4.1. Create the /home/student/review-cr1/ansible-navigator.yml file with the following content:

```
---
ansible-navigator:
  execution-environment:
    image: utility.lab.example.com/ee-supported-rhel8:latest
    pull:
      policy: missing
```

- 4.2. Run the podman login utility.lab.example.com command.

```
[student@workstation review-cr1]$ podman login utility.lab.example.com
Username: admin
Password: redhat
Login Succeeded!
```

- 4.3. Run the ansible-navigator command.

```
[student@workstation review-cr1]$ ansible-navigator
-----
Execution environment image and pull policy overview
-----
Execution environment image name:      utility.lab.example.com/ee-supported-rhel8:latest
Execution environment image tag:       latest
Execution environment pull arguments: None
Execution environment pull policy:    missing
```

```
Execution environment pull needed: True
-----
Updating the execution environment
-----
Running the command: podman pull utility.lab.example.com/ee-supported-rhel8:latest
Trying to pull utility.lab.example.com/ee-supported-rhel8:latest...
...output omitted...
```

5. In the project directory, create and run the `users.yml` playbook to add the users `joe` and `sam` to the inventory hosts in the `dev` group. Only use a single task in this playbook.

- 5.1. Create the `users.yml` playbook with the following content:

```
---
- name: Add users
  hosts: dev

  tasks:

    - name: Add the users joe and sam
      ansible.builtin.user:
        name: "{{ item }}"
      loop:
        - joe
        - sam
```

- 5.2. Run the `users.yml` playbook.

```
[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout users.yml

PLAY [Add users] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Add the users joe and sam] ****
changed: [serverb.lab.example.com] => (item=joe)
changed: [servera.lab.example.com] => (item=joe)
changed: [serverb.lab.example.com] => (item=sam)
changed: [servera.lab.example.com] => (item=sam)

PLAY RECAP ****
servera.lab.example.com    : ok=2    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
serverb.lab.example.com    : ok=2    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

6. Inspect the `packages.yml` playbook. In the play in that playbook, define a play variable named `packages` with a list of two packages as its value: `httpd` and `mariadb-server`. Run the `packages.yml` playbook.

- 6.1. Define the `packages` variable.

```

---
- name: Install packages
  hosts: dev
  vars:
    packages:
      - httpd
      - mariadb-server

  tasks:

    - name: Install the required packages
      ansible.builtin.dnf:
        name: "{{ packages }}"
        state: latest

```

6.2. Run the packages.yml playbook.

```

[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout packages.yml

PLAY [Install packages] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Install the required packages] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=2    changed=1    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0
serverb.lab.example.com      : ok=2    changed=1    unreachable=0    failed=0
                               skipped=0   rescued=0   ignored=0

```

7. Add a task to the packages.yml playbook that installs the redis package if the available swap space on the managed host is greater than 10 MB. Run the packages.yml playbook again after adding this task.

7.1. Add the new task to the packages.yml playbook.

```

---
- name: Install packages
  hosts: dev
  vars:
    packages:
      - httpd
      - mariadb-server

  tasks:

    - name: Install the required packages

```

```

ansible.builtin.dnf:
  name: "{{ packages }}"
  state: latest

- name: Install redis
  ansible.builtin.dnf:
    name: redis
    state: latest
  when: ansible_facts['swaptotal_mb'] > 10

```

- 7.2. Run the packages.yml playbook again.

```

[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout packages.yml

PLAY [Install packages] ****

TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [Install the required packages] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Install redis] ****
skipping: [servera.lab.example.com]
changed: [serverb.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=2    changed=0    unreachable=0    failed=0
  skipped=1    rescued=0    ignored=0
serverb.lab.example.com    : ok=3    changed=1    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0

```

8. Troubleshoot the existing verify_user.yml playbook. It is supposed to verify that the sam user was created successfully, and it is not supposed to create the sam user if it is missing. Run the playbook with the --check option and resolve any errors. Repeat this process until you can run the playbook with the --check option and it passes, and then run the verify_user.yml playbook normally.

- 8.1. Run the verify_user.yml playbook with the --check option.

```

[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout verify_user.yml --check
ERROR! couldn't resolve module/action 'ansible.builtin.user'. This often indicates
a misspelling, missing collection, or incorrect module path.

The error appears to be in '/home/student/review-cr1/verify_user.yml': line 7,
column 7, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

```

```
- name: Verify the sam user exists
  ^ here
Please review the log for errors.
```

8.2. Correct the spelling error in the verify_user.yml playbook.

```
---
- name: Verify the sam user was created
  hosts: dev

  tasks:

    - name: Verify the sam user exists
      ansible.builtin.user:
        name: sam
        check_mode: yes
        register: sam_check

    - name: Sam was created
      ansible.builtin.debug:
        msg: "Sam was created"
      when: sam_check['changed'] == false

    - name: Output sam user status to file
      ansible.builtin.lineinfile:
        path: /home/student/verify.txt
        line: "Sam was created"
        create: yes
      when: sam_check['changed'] == false
```

8.3. Run the verify_user.yml playbook with the --check option again.

```
[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout verify_user.yml --check

...output omitted...

TASK [Output sam user status to file] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "msg": "
Unsupported parameters for (ansible.builtin.lineinfile) module: when. Supported
parameters include: backup, group, firstmatch, setype, create, path (dest,
destfile, name), selevel, serole, backrefs, regexp (regex), mode, seuser,
attributes (attr), insertafter, line (value), insertbefore, search_string, state,
owner, unsafe_writes, validate."}
fatal: [serverb.lab.example.com]: FAILED! => {"changed": false, "msg": "
Unsupported parameters for (ansible.builtin.lineinfile) module: when. Supported
parameters include: path (dest, destfile, name), search_string, backrefs, serole,
validate, unsafe_writes, regexp (regex), state, setype, firstmatch, backup,
selevel, create, mode, insertbefore, insertafter, group, owner, attributes
(attr), seuser, line (value)."}  

PLAY RECAP ****
```

```
servera.lab.example.com    : ok=3      changed=0      unreachable=0      failed=1
  skipped=0    rescued=0    ignored=0
serverb.lab.example.com    : ok=3      changed=0      unreachable=0      failed=1
  skipped=0    rescued=0    ignored=0
Please review the log for errors.
```

8.4. Correct the indentation error in the `verify_user.yml` playbook.

```
---
- name: Verify the sam user was created
  hosts: dev

  tasks:

    - name: Verify the sam user exists
      ansible.builtin.user:
        name: sam
      check_mode: yes
      register: sam_check

    - name: Sam was created
      ansible.builtin.debug:
        msg: "Sam was created"
      when: sam_check['changed'] == false

    - name: Output sam user status to file
      ansible.builtin.lineinfile:
        path: /home/student/verify.txt
        line: "Sam was created"
        create: yes
      when: sam_check['changed'] == false
```

8.5. Run the `verify_user.yml` playbook with the `--check` option again.

```
[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout verify_user.yml --check

...output omitted...

PLAY RECAP ****
servera.lab.example.com    : ok=4      changed=1      unreachable=0      failed=0
  skipped=0    rescued=0    ignored=0
serverb.lab.example.com    : ok=4      changed=1      unreachable=0      failed=0
  skipped=0    rescued=0    ignored=0
```

8.6. Run the `verify_user.yml` playbook.

```
[student@workstation review-cr1]$ ansible-navigator run \
> -m stdout verify_user.yml

PLAY [Verify the sam user was created] ****
TASK [Gathering Facts] ****
```

```
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Verify the sam user exists] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Sam was created] ****
ok: [servera.lab.example.com] => {
    "msg": "Sam was created"
}
ok: [serverb.lab.example.com] => {
    "msg": "Sam was created"
}

TASK [Output sam user status to file] ****
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=4    changed=1    unreachable=0    failed=0
    skipped=0   rescued=0   ignored=0
serverb.lab.example.com      : ok=4    changed=1    unreachable=0    failed=0
    skipped=0   rescued=0   ignored=0
```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr1
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr1
```

This concludes the section.

▶ Lab

Creating Playbooks

In this review, you create three playbooks. The first playbook, `dev_deploy.yml`, installs and starts the web server. The second playbook, `get_web_content.yml`, ensures that the web server is serving content. The third playbook, `site.yml`, runs the other two playbooks.

Outcomes

- Create and execute playbooks to perform tasks on managed hosts.
- Use Jinja2 templates, blocks, and handlers in playbooks.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr2
```

Specifications

- Create the playbooks specified by this activity in the `/home/student/review-cr2` project directory.
- Create a playbook named `dev_deploy.yml` with one play that runs on the `webservers` host group (which contains the `servera.lab.example.com` and `serverb.lab.example.com` managed hosts). Enable privilege escalation for the play. Add the following tasks to the play:
 - Install the `httpd` package.
 - Start the `httpd` service and enable it to start on boot.
 - Deploy the `template/vhost.conf.j2` template to `/etc/httpd/conf.d/vhost.conf` on the managed hosts. This task should notify the `Restart httpd` handler.
 - Copy the `files/index.html` file to the `/var/www/vhosts/hostname` directory on the managed hosts. Ensure that the destination directory is created if it does not already exist.
 - Configure the firewall to allow the `httpd` service.
 - Add a `Restart httpd` handler to the play that restarts the `httpd` service.
- Create a playbook named `get_web_content.yml` with one play named `Test web content` that runs on the `workstation` managed host. This playbook tests whether the `dev_deploy.yml` playbook was run successfully and ensures that the web server is serving content. Enable privilege escalation for the play. Structure the play as follows:

- Create a block and rescue task named `Retrieve web content` and write to error log on failure.
- Inside the block, create a task named `Retrieve web content` that uses the `ansible.builtin.uri` module to return content from `http://servera.lab.example.com`. Register the results in a variable named `content`.
- Inside the rescue clause, create a task named `Write to error file` that writes the value of the `content` variable to the `/home/student/review-cr2/error.log` file if the block fails. The task must create the `error.log` file if it does not already exist.
- Create a new `site.yml` playbook that imports the plays from both the `dev_deploy.yml` and the `get_web_content.yml` playbooks.
- After you have completed the rest of the specifications, run the `site.yml` playbook. Make sure that all three playbooks run successfully.

Evaluation

As the student user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr2
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr2
```

This concludes the section.

► Solution

Creating Playbooks

In this review, you create three playbooks. The first playbook, `dev_deploy.yml`, installs and starts the web server. The second playbook, `get_web_content.yml`, ensures that the web server is serving content. The third playbook, `site.yml`, runs the other two playbooks.

Outcomes

- Create and execute playbooks to perform tasks on managed hosts.
- Use Jinja2 templates, blocks, and handlers in playbooks.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr2
```

1. Create a playbook named `dev_deploy.yml` that contains one play that runs on the `webservers` host group. Enable privilege escalation for the play. Add a task that installs the `httpd` package.

- 1.1. Change into the `/home/student/review-cr2` directory.

```
[student@workstation ~]$ cd ~/review-cr2
[student@workstation review-cr2]$
```

- 1.2. Create a playbook named `dev_deploy.yml` with one play that runs on the `webservers` host group. Enable privilege escalation for the play.

```
---
- name: Install and configure web servers
  hosts: webservers
  become: true

  tasks:
```

- 1.3. Add a task that installs the `httpd` package.

```
- name: Install httpd package
  ansible.builtin.dnf:
    name: httpd
    state: present
```

2. Add a task to the `dev_deploy.yml` playbook that starts the `httpd` service and enables it to start on boot.

```
- name: Start httpd service
ansible.builtin.service:
  name: httpd
  state: started
```

3. Add a task to the `dev_deploy.yml` playbook that deploys the `template/vhost.conf.j2` template to `/etc/httpd/conf.d/vhost.conf` on the managed hosts. This task should notify the `Restart httpd` handler.

```
- name: Deploy configuration template
ansible.builtin.template:
  src: templates/vhost.conf.j2
  dest: /etc/httpd/conf.d/vhost.conf
  owner: root
  group: root
  mode: '0644'
  notify: Restart httpd
```

4. Add a task to the `dev_deploy.yml` playbook that copies the `files/index.html` file to the `/var/www/vhosts/{{ ansible_facts['hostname'] }}` directory on the managed hosts.

Ensure that the destination directory is created if it does not already exist.

```
- name: Copy index.html
ansible.builtin.copy:
  src: files/
  dest: "/var/www/vhosts/{{ ansible_facts['hostname'] }}/"
  owner: root
  group: root
  mode: '0644'
```

5. Add a task to the `dev_deploy.yml` playbook that configures the firewall to allow the `httpd` service.

```
- name: Ensure web server port is open
ansible.posix.firewalld:
  state: enabled
  permanent: true
  immediate: true
  service: http
```

6. Add the `Restart httpd` handler to the `dev_deploy.yml` playbook that restarts the `httpd` service.

The completed playbook contains the following content:

```
---
- name: Install and configure web servers
  hosts: webservers
  become: true
```

```

tasks:
  - name: Install httpd package
    ansible.builtin.dnf:
      name: httpd
      state: present

  - name: Start httpd service
    ansible.builtin.service:
      name: httpd
      state: started

  - name: Deploy configuration template
    ansible.builtin.template:
      src: templates/vhost.conf.j2
      dest: /etc/httpd/conf.d/vhost.conf
      owner: root
      group: root
      mode: '0644'
    notify: Restart httpd

  - name: Copy index.html
    ansible.builtin.copy:
      src: files/
      dest: "/var/www/vhosts/{{ ansible_facts['hostname'] }}/"
      owner: root
      group: root
      mode: '0644'

  - name: Ensure web server port is open
    ansible.posix.firewalld:
      state: enabled
      permanent: true
      immediate: true
      service: http

handlers:
  - name: Restart httpd
    service:
      name: httpd
      state: restarted

```

7. Create a playbook named `get_web_content.yml`. Add a play named `Test web content` that runs on the `workstation` managed host. Enable privilege escalation for the play.

```

---
- name: Test web content
  hosts: workstation
  become: true

  tasks:

```

8. Add a task named `Retrieve web content and write to error log on failure` to the play in the `get_web_content.yml` playbook. Make that task a block that contains

a single task named `Retrieve web content`. The `Retrieve web content` task must use the `ansible.builtin.uri` module to return content from the URL `http://servera.lab.example.com`. Register the retrieved content in a variable named `content`.

```
---
- name: Test web content
  hosts: workstation
  become: true

  tasks:
    - name: Retrieve web content and write to error log on failure
      block:
        - name: Retrieve web content
          ansible.builtin.uri:
            url: http://servera.lab.example.com
            return_content: yes
          register: content
```

- In the `get_web_content.yml` playbook, add a `rescue` clause to the `block` task. Add a task to that `rescue` clause, named `Write to error file`, that writes the `content` variable to the `/home/student/review-cr2/error.log` file when the `Retrieve web content` task fails. Create the `error.log` file if it does not already exist.

The `get_web_content.yml` playbook now contains the following content:

```
---
- name: Test web content
  hosts: workstation
  become: true

  tasks:
    - name: Retrieve web content and write to error log on failure
      block:
        - name: Retrieve web content
          ansible.builtin.uri:
            url: http://servera.lab.example.com
            return_content: yes
          register: content
    rescue:
      - name: Write to error file
        ansible.builtin.lineinfile:
          path: /home/student/review-cr2/error.log
          line: "{{ content }}"
          create: true
```

- Create a new `site.yml` playbook that imports the plays from both the `dev_deploy.yml` and the `get_web_content.yml` playbooks.

```
---
# Deploy web servers
- import_playbook: dev_deploy.yml

# Retrieve web content
- import_playbook: get_web_content.yml
```

11. Run the `site.yml` playbook. You might see some tasks report as changed if you have not yet run the individual playbooks for testing. A second run of the playbook should succeed with no further changes.

```
[student@workstation review-cr2]$ ansible-navigator run \
> -m stdout site.yml

PLAY ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [Install httpd package] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Start httpd service] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Deploy configuration template] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Copy index.html] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Ensure web server port is open] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

RUNNING HANDLER [Restart httpd] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

PLAY [Test web content] ****
TASK [Gathering Facts] ****
ok: [workstation]

TASK [Retrieve web content] ****
ok: [workstation]

PLAY RECAP ****
servera.lab.example.com : ok=7    changed=6    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
serverb.lab.example.com : ok=7    changed=6    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
workstation      : ok=2    changed=0    unreachable=0    failed=0
  skipped=0    rescued=0    ignored=0
```

Evaluation

As the student user on the workstation machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr2
```

Finish

On the workstation machine, change to the student user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr2
```

This concludes the section.

▶ Lab

Managing Linux Hosts and Using System Roles

In this review, on managed hosts running Red Hat Enterprise Linux, you write playbooks that use two system roles to set up new storage with a logical volume and configure network interfaces, and use modules to set up a Cron job and a user with Sudo privileges.

Outcomes

- Use the `redhat.rhel_system_roles.storage` role to create, format, and persistently mount an LVM volume on a managed host.
- Create a user with sudo access.
- Configure network settings on managed hosts, and collect network-related Ansible facts.
- Schedule a Cron job.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr3
```

Specifications

- Use the `/home/student/review-cr3` project directory to perform this activity.
- Install the `redhat.rhel_system_roles` Ansible Content Collection in the `collections` subdirectory of your project directory.
- Write a playbook named `storage.yml` that uses the `redhat.rhel_system_roles.storage` system role to configure logical volumes for the managed hosts in the `webservers` group specified by the inventory file in your project directory. The playbook must set up the logical volumes as follows:
 - Create a volume group named `vg_web` on the `/dev/vdb` storage device.
 - Create a logical volume named `lv_content`, 128 MB in size, from the `vg_web` volume group, format it with an XFS file system, and mount it on the `/var/www/html/content` directory.
 - Create a logical volume named `lv_uploads`, 256 MB in size, from the `vg_web` volume group, format it with an XFS file system, and mount it on the `/var/www/html/uploads` directory.
- Run the `storage.yml` playbook to configure the storage.

- Write a playbook named `dev-users.yml` that creates the `webdev` user on managed hosts in the `webservers` inventory group. It must do so as follows:
 - The `webdev` password must be set by using the `pwhash` variable provided in the `pass-vault.yml` file, which is encrypted with Ansible Vault. The Ansible Vault password for the `pass-vault.yml` file is `redhat`.
 - The `webdev` user must also be a member of the `webdev` group.
 - Members of the `webdev` group must be able to run `sudo` commands without a password prompt. Create or modify the `sudoers` file in `/etc/sudoers.d/webdev` by using the `ansible.builtin.lineinfile` module. Any edits to the `sudoers` file should be validated before changes are applied.
- Run the `dev-users.yml` playbook. Verify that the `webdev` user can log in to a managed host in the `webservers` group, and can execute commands as `root` on that host by using `sudo` without a password.
- Write a playbook named `network.yml` that uses the `redhat.rhel_system_roles.network` system role to configure the `eth1` network interface on the managed hosts in the `webservers` inventory group with the `172.25.250.45/24` IP address.
- Run the `network.yml` playbook.
- Write a playbook named `log-rotate.yml` to set up a system Cron job as follows:
 - Use the `ansible.builtin.cron` module to create the `/etc/cron.d/rotate_web` system Cron job on managed hosts in the `webservers` inventory group.
 - The job must run as the `devops` user every night at midnight.
 - The job must run the `logrotate -f /etc/logrotate.d/httpd` command to rotate the logs in the `/var/log/httpd/` directory.
- Run the `log-rotate.yml` playbook.
- Write a playbook named `site.yml` that imports the four playbooks that you wrote in this activity, in the following order:
 - `storage.yml`
 - `dev-users.yml`
 - `network.yml`
 - `log-rotate.yml`
- Run the `site.yml` playbook, and ensure that there are no errors.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr3
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr3
```

This concludes the section.

► Solution

Managing Linux Hosts and Using System Roles

In this review, on managed hosts running Red Hat Enterprise Linux, you write playbooks that use two system roles to set up new storage with a logical volume and configure network interfaces, and use modules to set up a Cron job and a user with Sudo privileges.

Outcomes

- Use the `redhat.rhel_system_roles.storage` role to create, format, and persistently mount an LVM volume on a managed host.
- Create a user with sudo access.
- Configure network settings on managed hosts, and collect network-related Ansible facts.
- Schedule a Cron job.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr3
```

1. Install the `rhel_system_roles` collection from the `redhat-rhel_system_roles-1.19.3.tar.gz` file into the `collections` directory in the project directory.
 - 1.1. Change into the `review-cr3` directory.

```
[student@workstation ~]$ cd ~/review-cr3
[student@workstation review-cr3]$
```

- 1.2. Use the `ansible-galaxy` command to install the `rhel_system_roles` collection from the `redhat-rhel_system_roles-1.19.3.tar.gz` file into the `collections` directory.

```
[student@workstation review-cr3]$ ansible-galaxy collection install \
> ./redhat-rhel_system_roles-1.19.3.tar.gz -p collections
...output omitted...
redhat.rhel_system_roles:1.19.3 was installed successfully
```

2. Write a playbook named `storage.yml` that uses the `redhat.rhel_system_roles.storage` system role to configure logical volumes for

the managed hosts in the `webservers` group specified by the inventory file in your project directory. The playbook must set up the logical volumes as follows:

- Create a volume group named `vg_web` on the `/dev/vdb` storage device.
 - Create a logical volume named `lv_content`, 128 MB in size, from the `vg_web` volume group, format it with an XFS file system, and mount it on the `/var/www/html/content` directory.
 - Create a logical volume named `lv_uploads`, 256 MB in size, from the `vg_web` volume group, format it with an XFS file system, and mount it on the `/var/www/html/uploads` directory.
- 2.1. Create the `storage.yml` playbook that targets the `webservers` group and applies the `redhat.rhel_system_roles.storage` role.

```
---
- name: Configure storage on webservers
hosts: webservers

roles:
  - name: redhat.rhel_system_roles.storage
```

- 2.2. Define the `storage_pools` variable. Set the volume group name to `vg_web`, the type to `lvm`, and the disks to use the `/dev/vdb` device.

```
storage_pools:
  - name: vg_web
    type: lvm
    disks:
      - /dev/vdb
```

- 2.3. Define the `volumes` variable within `storage_pools`.

```
volumes:
```

- 2.4. Create a logical volume within `volumes` with the name `lv_content`, a size of 128 MB, a file system type of `xfs`, and a mount point of `/var/www/html/content`.

```
- name: lv_content
size: 128m
mount_point: "/var/www/html/content"
fs_type: xfs
state: present
```

- 2.5. Create another logical volume within `volumes` with the name `lv_uploads`, a size of 256 MB, a file system type of `xfs`, and a mount point of `/var/www/html/uploads`.

```
- name: lv_uploads
size: 256m
mount_point: "/var/www/html/uploads"
fs_type: xfs
state: present
```

The final playbook should consist of the following content:

```
---
- name: Configure storage on webservers
  hosts: webservers

  roles:
    - name: redhat.rhel_system_roles.storage
      storage_pools:
        - name: vg_web
          type: lvm
          disks:
            - /dev/vdb
      volumes:
        - name: lv_content
          size: 128m
          mount_point: "/var/www/html/content"
          fs_type: xfs
          state: present
        - name: lv_uploads
          size: 256m
          mount_point: "/var/www/html/uploads"
          fs_type: xfs
          state: present
```

2.6. Run the `storage.yml` playbook to configure the storage.

```
[student@workstation review-cr3]$ ansible-navigator run \
> -m stdout storage.yml

PLAY [Configure storage on webservers]
 ****

TASK [Gathering Facts]
 ****

ok: [servera.lab.example.com]
...output omitted...
TASK [redhat.rhel_system_roles.storage : make sure blivet is available]
 ****

changed: [servera.lab.example.com]
...output omitted...
TASK [redhat.rhel_system_roles.storage : set up new/current mounts]
 ****

changed: [servera.lab.example.com] => (item={'src': '/dev/mapper/vg_web-
lv_content', 'path': '/var/www/html/content', 'fstype': 'xfs', 'opts': 'defaults',
'dump': 0, 'passno': 0, 'state': 'mounted'})
changed: [servera.lab.example.com] => (item={'src': '/dev/mapper/vg_web-
lv_uploads', 'path': '/var/www/html/uploads', 'fstype': 'xfs', 'opts': 'defaults',
'dump': 0, 'passno': 0, 'state': 'mounted'})
...output omitted...
PLAY RECAP ****
servera.lab.example.com    : ok=21    changed=3    unreachable=0    failed=0
                           skipped=12   rescued=0   ignored=0
```

3. Write a playbook named `dev-users.yml` that creates the user `webdev` on managed hosts in the `webservers` inventory group. It must do so as follows:

- The `webdev` password must be set by using the `pwhash` variable provided in the `pass-vault.yml` file, which is encrypted with Ansible Vault. The Ansible Vault password for the `pass-vault.yml` file is `redhat`.
- The `webdev` user must be a member of the `webdev` group.
- Members of the `webdev` group must be able to run `sudo` commands without a password prompt. Create or modify the `sudoers` file in `/etc/sudoers.d/webdev` by using the `ansible.builtin.lineinfile` module. Any edits to the `sudoers` file should be validated before changes are applied.

- 3.1. Start writing the `dev-users.yml` playbook. Define a single play in the playbook that targets the `webservers` host group.

Add a `vars_files` key to access the `pass-vault.yml` file.

Add the `tasks` key to the playbook.

```
---
- name: Create local users
  hosts: webservers
  vars_files:
    - pass-vault.yml
  tasks:
```

- 3.2. Add the first task to the playbook. Use the `ansible.builtin.group` module to create the `webdev` group on the managed host.

The task should consist of the following content:

```
- name: Add webdev group
  ansible.builtin.group:
    name: webdev
    state: present
```

- 3.3. Add a second task to the playbook that uses the `ansible.builtin.user` module. Use the `ansible.builtin.user` module to create the `webdev` user on the managed host.

The task should consist of the following content:

```
- name: Create user accounts
  ansible.builtin.user:
    name: webdev
    groups: webdev
    password: "{{ pwhash }}"
```

- 3.4. Add a third task to the play that uses the `ansible.builtin.lineinfile` module to modify the `sudo` configuration file and allow the `webdev` group members to use `sudo` without a password on the managed host. Use the `validate` parameter to validate the new `sudoers` entry.

The task should consist of the following content:

```

- name: Modify sudo config to allow webdev members sudo without a password
  ansible.builtin.lineinfile:
    path: /etc/sudoers.d/webdev
    state: present
    create: yes
    mode: 0440
    line: "%webdev ALL=(ALL) NOPASSWD: ALL"
    validate: /usr/sbin/visudo -cf %
  
```

The completed playbook should consist of the following content:

```

---
- name: Create local users
  hosts: webservers
  vars_files:
    - pass-vault.yml
  tasks:
    - name: Add webdev group
      ansible.builtin.group:
        name: webdev
        state: present

    - name: Create user accounts
      ansible.builtin.user:
        name: webdev
        groups: webdev
        password: "{{ pwhash }}"

    - name: Modify sudo config to allow webdev members sudo without a password
      ansible.builtin.lineinfile:
        path: /etc/sudoers.d/webdev
        state: present
        create: yes
        mode: 0440
        line: "%webdev ALL=(ALL) NOPASSWD: ALL"
        validate: /usr/sbin/visudo -cf %
  
```

4. Run the `dev-users.yml` playbook. Verify that the `webdev` user can log in to a managed host, and execute commands using `sudo` without a password.

4.1. Run the `dev-users.yml` playbook:

```

[student@workstation review-cr3]$ ansible-navigator run \
> -m stdout --playbook-artifact-enable false dev-users.yml --vault-id @prompt
Vault password (default): redhat

PLAY [Create local users] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Add webdev group] ****
changed: [servera.lab.example.com]
  
```

```

TASK [Create user accounts] ****
changed: [servera.lab.example.com]

TASK [Modify sudo config to allow webdev members sudo without a password] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=4      changed=3      unreachable=0      failed=0
skipped=0      rescued=0      ignored=0

```

- 4.2. Use SSH as the `webdev` user and log in to the `servera.lab.example.com` server.

```

[student@workstation review-cr3]$ ssh webdev@servera
...output omitted...
[webdev@servera ~]$

```

- 4.3. Change to the `root` user and confirm that no password is required.

```

[webdev@servera ~]$ sudo -i
[root@servera ~]#

```

- 4.4. Log out from `servera.lab.example.com`.

```

[root@servera ~]# exit
logout
[webdev@servera ~]$ exit
logout
Connection to servera closed.
[student@workstation review-cr3]$

```

5. Write a playbook named `network.yml` that uses the `redhat.rhel_system_roles.network` system role to configure the `eth1` network interface on managed hosts in the `webservers` inventory group with the `172.25.250.45/24` IP address.

- 5.1. Create the `network.yml` playbook with one play that targets the `webservers` inventory group. Include the `redhat.rhel_system_roles.network` role in the `roles` section of the play.

```

---
- name: NIC Configuration
  hosts: webservers

  roles:
    - redhat.rhel_system_roles.network

```

- 5.2. The `/usr/share/doc/rhel-system-roles/network/README.md` file lists all available variables and options for the `rhel-system-roles.network` role.

Review the `Setting the IP configuration:` section in the `README.md` file and determine which variable is required to configure the `eth1` network interface with the `172.25.250.45` IP address.

```
[student@workstation review-cr3]$ cat > collections/ansible_collections/redhat/rhel_system_roles/roles/network/README.md  
...output omitted...  
Setting the IP configuration:  
  
```yaml  
network_connections:
 - name: eth0
 type: ethernet
 ip:
 route_metric4: 100
 dhcp4: no
 #dhcp4_send_hostname: no
 gateway4: 192.0.2.1

 dns:
 - 192.0.2.2
 - 198.51.100.5
 dns_search:
 - example.com
 - subdomain.example.com
 dns_options:
 - rotate
 - timeout:1

 route_metric6: -1
 auto6: no
 gateway6: 2001:db8::1

 address:
 - 192.0.2.3/24
 - 198.51.100.3/26
 - 2001:db8::80/7

...output omitted...
```

### 5.3. Create the group\_vars/webservers subdirectory.

```
[student@workstation review-cr3]$ mkdir -pv group_vars/webservers
mkdir: created directory 'group_vars'
mkdir: created directory 'group_vars/webservers'
```

### 5.4. Create a network.yml file to define the required role variable.

Because the variable value applies to the hosts on the `webservers` host group, you need to create the file in the `group_vars/webservers` directory.

Add the variable definition to support the configuration of the `eth1` network interface.

When completed, the file contains the following content:

```

network_connections:
 - name: eth1
 type: ethernet
 ip:
 address:
 - 172.25.250.45/24
```

- 5.5. Run the `network.yml` playbook to configure the `eth1` network interface on managed hosts in the `webservers` inventory group.

```
[student@workstation review-cr3]$ ansible-navigator run \
> -m stdout network.yml

PLAY [NIC Configuration] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Check which services are running] ****
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Check which packages are installed] ***
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Print network provider] ****
ok: [servera.lab.example.com] => {
 "msg": "Using network provider: nm"
}

TASK [redhat.rhel_system_roles.network : Install packages] ****
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Restart NetworkManager due to wireless or
team interfaces] ***
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Enable and start NetworkManager] ****
ok: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Enable and start wpa_supplicant] ****
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Enable network service] ****
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Ensure initscripts network file
dependency is present] ***
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Configure networking connection profiles]

```

changed: [servera.lab.example.com]

```

TASK [redhat.rhel_system_roles.network : Show stderr messages] ****
ok: [servera.lab.example.com] => {
 "__network_connections_result.stderr_lines": [
 "[002] <info> #0, state:None persistent_state:present, 'eth1': add
connection eth1, 3ddd5197-7a96-4d05-abec-c586089145ff"
]
}

TASK [redhat.rhel_system_roles.network : Show debug messages] ****
skipping: [servera.lab.example.com]

TASK [redhat.rhel_system_roles.network : Re-test connectivity] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=10 changed=1 unreachable=0 failed=0
 skipped=12 rescued=0 ignored=0

```

6. Write a playbook named `log-rotate.yml` to set up a system Cron job as follows:

- Use the `ansible.builtin.cron` module to create the `/etc/cron.d/rotate_web` system Cron job on managed hosts in the `webservers` inventory group.
- The job must run as the `devops` user every night at midnight.
- The job must run the `logrotate -f /etc/logrotate.d/httpd` command to rotate the logs in the `/var/log/httpd/` directory.

Run the `log-rotate.yml` playbook.

- 6.1. Create the `log-rotate.yml` playbook and add the lines needed to start the play. It must target the managed hosts in the `webservers` group and enable privilege escalation.

```

- name: Recurring cron job
 hosts: webservers
 become: true

```

- 6.2. Define a task that uses the `ansible.builtin.cron` module to schedule a recurring Cron job.

```

tasks:
 - name: Crontab file exists
 ansible.builtin.cron:
 name: Rotate HTTPD logs

```

- 6.3. Configure the job to run every night at midnight.

```

minute: "0"
hour: "0"
weekday: "*"

```

- 6.4. Use the `cron_file` parameter to create the job in the `/etc/cron.d/rotate_web` system Cron job file instead of an individual user's `crontab` in `/var/spool/cron/`. Using a relative path places the file in the `/etc/cron.d` directory.
- If the `cron_file` parameter is used, you must also specify the `user` parameter to fill in the field that specifies which user runs the system Cron job.

```
user: devops
job: "logrotate -f /etc/logrotate.d/httpd"
cron_file: rotate_web
state: present
```

- 6.5. When completed, the playbook must contain the following content. Review the playbook for accuracy.

```

- name: Recurring cron job
hosts: webservers
become: true

tasks:
- name: Crontab file exists
 ansible.builtin.cron:
 name: Rotate HTTPD logs
 minute: "0"
 hour: "0"
 weekday: "*"
 user: devops
 job: "logrotate -f /etc/logrotate.d/httpd"
 cron_file: rotate_web
 state: present
```

- 6.6. Run the `ansible-navigator run --syntax-check` command to verify the playbook syntax. Correct any errors before moving to the next step.

```
[student@workstation review-cr3]$ ansible-navigator run \
> -m stdout log-rotate.yml --syntax-check
playbook: /home/student/review-cr3/log-rotate.yml
```

- 6.7. Run the `log-rotate.yml` playbook.

```
[student@workstation review-cr3]$ ansible-navigator run \
> -m stdout log-rotate.yml

PLAY [Recurring cron job] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Crontab file exists] ****
changed: [servera.lab.example.com]
```

```
PLAY RECAP ****
servera.lab.example.com : ok=2 changed=1 unreachable=0 failed=0
 skipped=0 rescued=0 ignored=0
```

- 6.8. Run the following command to verify that the /etc/cron.d/rotate\_web Cron file exists, and its content is correct. Exit servera.lab.example.com when done.

```
[student@workstation review-cr3]$ ssh devops@servera \
> "cat /etc/cron.d/rotate_web"
#Ansible: Rotate HTTPD logs
0 0 * * * devops logrotate -f /etc/logrotate.d/httpd
[student@workstation review-cr3]$
```

7. Write a playbook named site.yml that imports the four playbooks that you wrote in the preceding steps, in the order in which they were created:

- storage.yml
- dev-users.yml
- network.yml
- log-rotate.yml

Run the site.yml playbook, and ensure that there are no errors.

- 7.1. Start writing the site.yml playbook. Add an import\_playbook statement for each playbook created in the previous steps.

```

- import_playbook: storage.yml
- import_playbook: dev-users.yml
- import_playbook: network.yml
- import_playbook: log-rotate.yml
```

- 7.2. Run the site.yml playbook.

```
[student@workstation review-cr3]$ ansible-navigator run \
> -m stdout --playbook-artifact-enable false site.yml --vault-id @prompt
Vault password (default): redhat

...output omitted...

TASK [Crontab file exists] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=37 changed=0 unreachable=0 failed=0
 skipped=24 rescued=0 ignored=0
```

## Evaluation

As the student user on the workstation machine, use the lab command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr3
```

## Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr3
```

This concludes the section.

## ▶ Lab

# Creating Roles

In this review, you create an Ansible role from an existing Ansible Playbook on `workstation`, and then create a playbook to apply the role to `serverb.lab.example.com` and `serverc.lab.example.com`.

## Outcomes

- Create a role from an existing playbook.
- Create a playbook to apply the role to managed hosts.

## Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr4
```

## Specifications

- The `review-cr4` directory contains your Ansible project for this activity.
- Convert the `ansible-htpd.yml` playbook in the project directory into a new Ansible role named `ansible-htpd`. The new role must be created in the `/home/student/review-cr4/roles/ansible-htpd` directory.
- Move any variables, tasks, templates, files, and handlers that were used in or by the playbook into the appropriate files or directories in the new role.
- Update the `meta/main.yml` file in the role with the following content:

Variable	Value
author	Red Hat Training
description	example role for RH294
company	Red Hat
license	BSD

- Edit the `roles/ansible-htpd/README.md` file so that it provides the following information about the role:

```
ansible-httdp
=====
Example ansible-httdp role
from "Red Hat Enterprise Linux Automation with Ansible" (RH294)

Role Variables

* defaults/main.yml contains variables used to configure the httpd.conf template
* vars/main.yml contains the name of the httpd service, the name of the RPM
 package, the location of the service's configuration file, and the name of the
 firewall service.

Dependencies

None.

Example Playbook

- hosts: servers
 roles:
 - ansible-httdp

License

BSD

Author Information

Red Hat (training@redhat.com)
```

- Remove any unused directories and files within the role.
- In the project directory, write a `site.yml` playbook that runs the new `ansible-httdp` role on the managed hosts in the `webdev` inventory group.
- Run the `site.yml` playbook.

## Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr4
```

## Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr4
```

This concludes the section.

## ► Solution

# Creating Roles

In this review, you create an Ansible role from an existing Ansible Playbook on `workstation`, and then create a playbook to apply the role to `serverb.lab.example.com` and `serverc.lab.example.com`.

## Outcomes

- Create a role from an existing playbook.
- Create a playbook to apply the role to managed hosts.

## Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start review-cr4
```

1. Use the `ansible-httpd.yml` playbook to create a new Ansible role named `ansible-httpd`.

- 1.1. Change into the `/home/student/review-cr4` directory.

```
[student@workstation ~]$ cd ~/review-cr4
[student@workstation review-cr4]$
```

- 1.2. Create the `roles` subdirectory.

```
[student@workstation review-cr4]$ mkdir -v roles
mkdir: created directory 'roles'
```

- 1.3. Using the `ansible-galaxy` command, create the directory structure for the new `ansible-httpd` role in the `roles` subdirectory.

```
[student@workstation review-cr4]$ cd roles
[student@workstation roles]$ ansible-galaxy init ansible-httpd
- Role ansible-httpd was created successfully
[student@workstation roles]$ cd ..
[student@workstation review-cr4]$
```

- 1.4. Use the `tree` command to verify the directory structure created for the new role.

```
[student@workstation review-cr4]$ tree roles
roles
└── ansible-htpd
 ├── 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
```

2. Move any variables, tasks, templates, files, and handlers into the appropriate files inside the new role.
  - 2.1. Copy the variables from the `ansible-htpd.yml` file into the `roles/ansible-htpd/vars/main.yml` file. The `roles/ansible-htpd/vars/main.yml` file should contain the following content:

```

vars file for ansible-htpd
web_package: httpd
web_service: httpd
web_config_file: /etc/httpd/conf/httpd.conf
web_root: /var/www/html/index.html
web_fw_service: http
```

- 2.2. Copy the `httpd` configuration file template from `templates/httpd.conf.j2` into the `roles/ansible-htpd/templates/` directory.

```
[student@workstation review-cr4]$ cp \
> -v templates/httpd.conf.j2 roles/ansible-htpd/templates/
'templates/httpd.conf.j2' -> 'roles/ansible-htpd/templates/httpd.conf.j2'
```

- 2.3. Copy the tasks from the `ansible-htpd.yml` file into the `roles/ansible-htpd/tasks/main.yml` file. The `roles/ansible-htpd/tasks/main.yml` file should contain the following content:

```

tasks file for ansible-htpd
- name: Packages are installed
 ansible.builtin.dnf:
```

```

name: "{{ web_package }}"
state: present

- name: Ensure service is started
 ansible.builtin.service:
 name: "{{ web_service }}"
 state: started
 enabled: yes

- name: Deploy configuration file
 ansible.builtin.template:
 src: templates/httpd.conf.j2
 dest: "{{ web_config_file }}"
 owner: root
 group: root
 mode: '0644'
 setype: httpd_config_t
 notify: restart httpd

- name: Deploy index.html file
 ansible.builtin.copy:
 src: files/index.html
 dest: "{{ web_root }}"
 owner: root
 group: root
 mode: '0644'

- name: Web port is open
 ansible.builtin.firewalld:
 service: "{{ web_fw_service }}"
 permanent: yes
 state: enabled
 immediate: yes

```

- 2.4. Copy the `files/index.html` file into the `roles/ansible-htpd/files/` directory.

```
[student@workstation review-cr4]$ cp \
> -v files/index.html roles/ansible-htpd/files/
'files/index.html' -> 'roles/ansible-htpd/files/index.html'
```

- 2.5. Copy the handlers from the `ansible-htpd.yml` file into the `roles/ansible-htpd/handlers/main.yml` file. The `roles/ansible-htpd/handlers/main.yml` file should contain the following content:

```

handlers file for ansible-htpd
- name: restart httpd
 ansible.builtin.service:
 name: "{{ web_service }}"
 state: restarted

```

3. Update the `roles/ansible-httdp/meta/main.yml` file in the role according to the specifications.

- 3.1. Change the value of the `author` entry to `Red Hat Training`.

```
author: Red Hat Training
```

- 3.2. Change the value of the `description` entry to `example role for RH294`.

```
description: example role for RH294
```

- 3.3. Change the value of the `company` entry to `Red Hat`.

```
company: Red Hat
```

- 3.4. Change the value of the `license` entry to `BSD`.

```
license: BSD
```

4. Edit the `roles/ansible-httdp/README.md` file so that it provides pertinent information regarding the role. The file should consist of the following content:

```
ansible-httdp
=====
Example ansible-httdp role
from "Red Hat Enterprise Linux Automation with Ansible" (RH294)

Role Variables

* defaults/main.yml contains variables used to configure the httpd.conf template
* vars/main.yml contains the name of the httpd service, the name of the RPM
package, the location of the service's configuration file, and the name of the
firewall service.

Dependencies

None.

Example Playbook

- hosts: servers
 roles:
 - ansible-httdp

License

BSD
```

**Author Information**

Red Hat (training@redhat.com)

5. Remove the unused directories from the new role.

- 5.1. Remove the `roles/ansible-httdp/defaults/` directory.

```
[student@workstation review-cr4]$ rm -rfv roles/ansible-httdp/defaults/
removed 'roles/ansible-httdp/defaults/main.yml'
removed directory 'roles/ansible-httdp/defaults/'
```

- 5.2. Remove the `roles/ansible-httdp/tests/` directory.

```
[student@workstation review-cr4]$ rm -rfv roles/ansible-httdp/tests/
removed 'roles/ansible-httdp/tests/inventory'
removed 'roles/ansible-httdp/tests/test.yml'
removed directory 'roles/ansible-httdp/tests/'
```

6. In the project directory, write a `site.yml` playbook that runs the new `ansible-httdp` role on the managed hosts in the `webdev` inventory group. The `site.yml` playbook should contain content similar to the following example:

```

- name: Apply the ansible-httdp role
 hosts: webdev

 roles:
 - ansible-httdp
```

7. Run the `site.yml` playbook.

```
[student@workstation review-cr4]$ ansible-navigator run -m stdout site.yml

PLAY [Apply the ansible-httdp role] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [serverc.lab.example.com]

TASK [ansible-httdp : Packages are installed] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

TASK [ansible-httdp : Ensure service is started] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

TASK [ansible-httdp : Deploy configuration file] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]
```

```
TASK [ansible-httd : Deploy index.html file] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

TASK [ansible-httd : Web port is open] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

RUNNING HANDLER [ansible-httd : restart httpd] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=7 changed=6 unreachable=0 failed=0
 skipped=0 rescued=0 ignored=0
serverc.lab.example.com : ok=7 changed=6 unreachable=0 failed=0
 skipped=0 rescued=0 ignored=0
```

## Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab grade review-cr4
```

## Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish review-cr4
```

This concludes the section.