

Join the explorers, builders, and individuals who boldly offer new solutions to old problems. For open source, innovation is only possible because of the people behind it.

RED HAT® TRAINING+ CERTIFICATION

STUDENT WORKBOOK (ROLE)

Red Hat Ansible Tower 3.3 DO410

AUTOMATION WITH ANSIBLE AND ANSIBLE TOWER

Edition 1



AUTOMATION WITH ANSIBLE AND ANSIBLE TOWER



Red Hat Ansible Tower 3.3 DO410
Automation with Ansible and Ansible Tower
Edition 1 20190110
Publication date 20190110

Authors: Chen Chang, Fabien Cambi, Artur Glogowski, Ricardo da Costa,
George Hacker, Razique Mahroua, Adolfo Vazquez,
Snehangshu Karmakar
Editor: Steven Bonneville

Copyright © 2019 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2019 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 e-mail training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

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

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

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

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

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Our thanks for feedback from Timothy Appnel, Graham Mainwaring, Bill Nottingham, David Federlein, and Nikhil Jain.

Document Conventions	ix
Introduction	xi
Automation with Ansible and Ansible Tower	xi
Orientation to the Classroom Environment	xii
Internationalization	xv
1. Introducing Ansible	1
Overview of Ansible	2
Quiz: Overview of Ansible	7
Installing Ansible	9
Guided Exercise: Installing Ansible	13
Summary	14
2. Deploying Ansible	15
Building an Ansible Inventory	16
Guided Exercise: Building an Ansible Inventory	20
Managing Ansible Configuration Files	24
Guided Exercise: Managing Ansible Configuration Files	32
Running Ad Hoc Commands	36
Guided Exercise: Running Ad Hoc Commands	42
Lab: Deploying Ansible	47
Summary	56
3. Implementing Playbooks	57
Writing and Running Playbooks	58
Guided Exercise: Writing and Running Playbooks	64
Implementing Multiple Plays	69
Guided Exercise: Implementing Multiple Plays	79
Lab: Implementing Playbooks	86
Summary	93
4. Managing Variables and Facts	95
Managing Variables	96
Guided Exercise: Managing Variables	104
Managing Secrets	109
Guided Exercise: Managing Secrets	114
Managing Facts	117
Guided Exercise: Managing Facts	126
Lab: Managing Variables and Facts	131
Summary	142
5. Implementing Task Control	143
Writing Loops and Conditional Tasks	144
Guided Exercise: Writing Loops and Conditional Tasks	154
Implementing Handlers	163
Guided Exercise: Implementing Handlers	166
Handling Task Failure	171
Guided Exercise: Handling Task Failure	175
Lab: Implementing Task Control	183
Summary	191
6. Deploying Files to Managed Hosts	193
Modifying and Copying Files to Hosts	194
Guided Exercise: Modifying and Copying Files to Hosts	200
Deploying Custom Files with Jinja2 Templates	209
Guided Exercise: Deploying Custom Files with Jinja2 Templates	214
Lab: Deploying Files to Managed Hosts	217
Summary	222

7. Managing Large Projects	223
Selecting Hosts with Host Patterns	224
Guided Exercise: Selecting Hosts with Host Patterns	232
Managing Dynamic Inventories	239
Guided Exercise: Managing Dynamic Inventories	244
Configuring Parallelism	248
Guided Exercise: Configuring Parallelism	251
Including and Importing Files	256
Guided Exercise: Including and Importing Files	261
Lab: Managing Large Projects	266
Summary	274
8. Simplifying Playbooks with Roles	275
Describing Role Structure	276
Quiz: Describing Role Structure	281
Creating Roles	283
Guided Exercise: Creating Roles	289
Deploying Roles with Ansible Galaxy	298
Guided Exercise: Deploying Roles with Ansible Galaxy	305
Reusing Content with System Roles	312
Guided Exercise: Reusing Content with System Roles	319
Lab: Implementing Roles	325
Summary	337
9. Troubleshooting Ansible	339
Troubleshooting Playbooks	340
Guided Exercise: Troubleshooting Playbooks	343
Troubleshooting Ansible Managed Hosts	350
Guided Exercise: Troubleshooting Ansible Managed Hosts	355
Lab: Troubleshooting Ansible	359
Summary	368
10. Installing and Accessing Ansible Tower	369
Explaining the Red Hat Ansible Tower Architecture	370
Quiz: Explaining the Red Hat Ansible Tower Architecture	374
Installing Red Hat Ansible Tower	376
Guided Exercise: Installing Red Hat Ansible Tower	380
Navigating Red Hat Ansible Tower	382
Guided Exercise: Accessing Red Hat Ansible Tower	391
Quiz: Installing and Accessing Red Hat Ansible Tower	393
Summary	395
11. Managing Access with Users and Teams	397
Creating and Managing Ansible Tower Users	398
Guided Exercise: Creating and Managing Ansible Tower Users	404
Managing Users Efficiently with Teams	408
Guided Exercise: Managing Users Efficiently with Teams	411
Lab: Managing Access with Users and Teams	416
Summary	420
12. Managing Inventories and Credentials	421
Creating a Static Inventory	422
Guided Exercise: Creating a Static Inventory	430
Creating Machine Credentials for Access to Inventory Hosts	434
Guided Exercise: Creating Machine Credentials for Access to Inventory Hosts	440
Lab: Creating and Managing Inventories and Credentials	443
Summary	448
13. Managing Projects and Launching Ansible Jobs	449

Managing Ansible Project Materials Using Git	450
Guided Exercise: Managing Ansible Project Materials Using Git	458
Creating a Project for Ansible Playbooks	463
Guided Exercise: Creating a Project for Ansible Playbooks	470
Creating Job Templates and Launching Jobs	473
Guided Exercise: Creating Job Templates and Launching Jobs	480
Lab: Managing Projects and Launching Ansible Jobs	485
Summary	493
14. Constructing Advanced Job Workflows	495
Improving Performance with Fact Caching	496
Guided Exercise: Improving Performance with Fact Caching	499
Creating Job Template Surveys to Set Variables for Jobs	504
Guided Exercise: Creating Job Template Surveys to Set Variables for Jobs	510
Creating Workflow Job Templates and Launching Workflow Jobs	513
Guided Exercise: Creating Workflow Job Templates and Launching Workflow Jobs	519
Scheduling Jobs and Configuring Notifications	522
Guided Exercise: Scheduling Jobs and Configuring Notifications	526
Launching Jobs with the Ansible Tower API	531
Guided Exercise: Launching Jobs with the Ansible Tower API	544
Lab: Constructing Advanced Job Workflows	549
Summary	556
15. Managing Advanced Inventories	557
Importing External Static Inventories	558
Guided Exercise: Importing External Static Inventories	562
Creating and Updating Dynamic Inventories	566
Guided Exercise: Creating and Updating Dynamic Inventories	574
Filtering Hosts with Smart Inventories	576
Guided Exercise: Filtering Hosts with Smart Inventories	580
Lab: Managing Advanced Inventories	583
Summary	590
16. Performing Maintenance and Routine Administration of Ansible Tower	591
Performing Basic Troubleshooting of Ansible Tower	592
Guided Exercise: Performing Basic Troubleshooting of Ansible Tower	598
Configuring TLS/SSL for Ansible Tower	605
Guided Exercise: Configuring TLS/SSL for Ansible Tower	608
Backing Up and Restoring Ansible Tower	610
Guided Exercise: Backing Up and Restoring Ansible Tower	614
Quiz: Performing Maintenance and Routine Administration of Ansible Tower	617
Summary	619
17. Comprehensive Review: Automation with Ansible and Ansible Tower	621
Lab: Deploying Ansible	622
Lab: Creating Playbooks	627
Lab: Creating Roles and Using Dynamic Inventory	637
Lab: Restoring Ansible Tower from Backup	649
Lab: Adding Users and Teams	652
Lab: Creating a Custom Dynamic Inventory	657
Lab: Configuring Job Templates	663
Lab: Configuring Workflow Job Templates, Surveys, and Notifications	673
Lab: Testing the Prepared Environment	682
A. Ansible Lightbulb Licensing	687
Ansible Lightbulb License	688

DOCUMENT CONVENTIONS



REFERENCES

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



NOTE

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



IMPORTANT

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



WARNING

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

INTRODUCTION

AUTOMATION WITH ANSIBLE AND ANSIBLE TOWER

Automation with Ansible and Ansible Tower (DO410) is designed for IT professionals who want to develop standardized automation of the enterprise IT environment in order to improve operational efficiency by using Ansible. Students will learn how to use Ansible for automation, configuration, provisioning, and management, and Red Hat Ansible Tower to centrally manage Ansible at an enterprise scale.

DO410 combines the content from Automation with Ansible (DO407) and Automation with Ansible II: Ansible Tower (DO409).

Figure 0.1: Automation with Ansible and Ansible Tower introduction

OBJECTIVES

- Installing and troubleshooting Ansible on central nodes and managed hosts
- Automating administration tasks with Ansible playbooks and ad hoc commands
- Writing effective Ansible playbooks
- Protecting sensitive data used by tasks with Ansible Vault
- Installing and configuring Ansible Tower for enterprise Ansible management
- Using Ansible Tower to control access to inventories and machine credentials by users and teams
- Creating job templates in Ansible Tower to standardize playbook execution
- Centrally launching playbooks and monitoring and reviewing job results with Ansible Tower

AUDIENCE

- Linux system administrators, cloud administrators, and network administrators needing to automate configuration management, application deployment, and intra-service orchestration at an enterprise scale.

PREREQUISITES

- Red Hat Certified System Administrator (RHCSA) in Red Hat Enterprise Linux certification or equivalent Linux system administration skills.

ORIENTATION TO THE CLASSROOM ENVIRONMENT

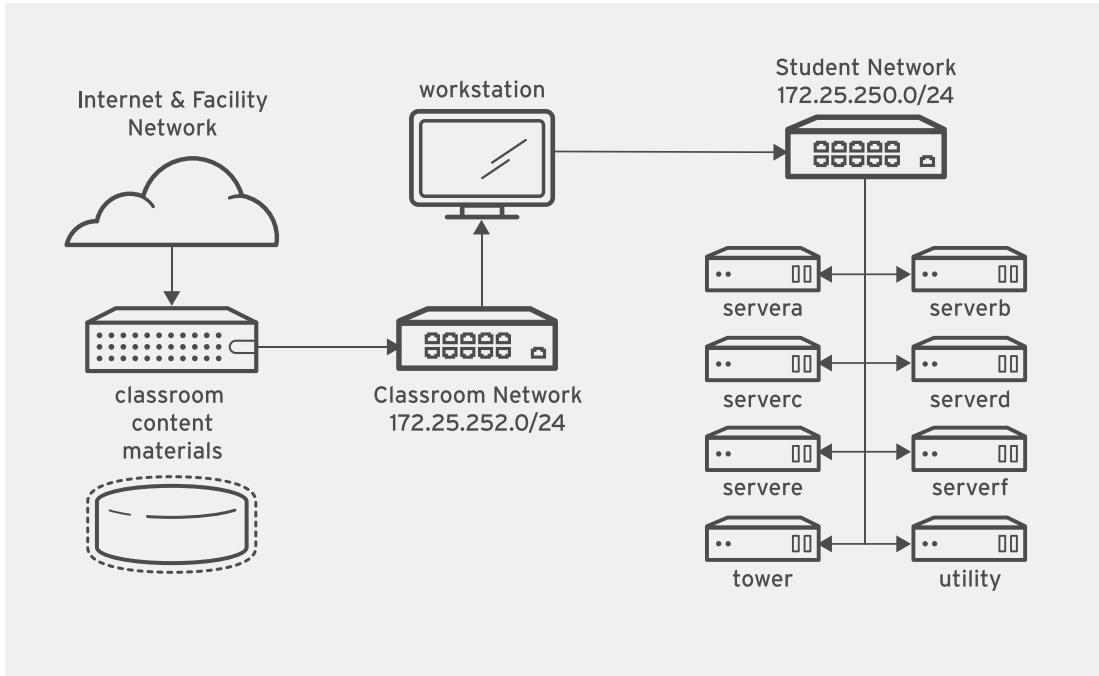


Figure 0.2: Classroom Environment

In this course, the main computer system used for hands-on learning activities is **workstation**. Eight other machines will also be used by students for these activities. These are **tower**, **utility**, **servera**, **serverb**, **serverc**, **serverd**, **servere**, **serverf**. All nine of these 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
workstation.lab.example.com	172.25.250.254	Graphical workstation used for administration
utility.lab.example.com	172.25.250.8	Host used for IPA server and Git repository
tower.lab.example.com	172.25.250.9	Host used for Ansible Tower server
servera.lab.example.com	172.25.250.10	Host managed with Ansible
serverb.lab.example.com	172.25.250.11	Host managed with Ansible
serverc.lab.example.com	172.25.250.12	Host managed with Ansible

MACHINE NAME	IP ADDRESSES	ROLE
serverd.lab.example.com	172.25.250.13	Host managed with Ansible
servere.lab.example.com	172.25.250.14	Host managed with Ansible
serverf.lab.example.com	172.25.250.15	Host managed with Ansible

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

There are several systems in the classroom that 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 will be provided in the instructions for those activities.

Controlling your station

The top of the console describes the state of your machine.

Machine States

STATE	DESCRIPTION
none	Your machine has not yet been started. When started, your machine will boot into a newly initialized state (the desk will have been reset).
starting	Your machine is in the process of booting.
running	Your machine is running and available (or, when booting, soon will be.)
stopping	Your machine is in the process of shutting down.
stopped	Your machine is completely shut down. Upon starting, your machine will boot into the same state as when it was shut down (the disk will have been preserved).
impaired	A network connection to your machine cannot be made. Typically this state is reached when a student has corrupted networking or firewall rules. If the condition persists after a machine reset, or is intermittent, please open a support case.

Depending on the state of your machine, a selection of the following actions will be available to you.

Machine Actions

ACTION	DESCRIPTION
Start Station	Start ("power on") the machine.
Stop Station	Stop ("power off") the machine, preserving the contents of its disk.
Reset Station	Stop ("power off") the machine, resetting the disk to its initial state. Caution: Any work generated on the disk will be lost.

ACTION	DESCRIPTION
Refresh	Refresh the page will re-probe the machine state.
Increase Timer	Adds 15 minutes to the timer for each click.

The Station Timer

Your Red Hat Online Learning enrollment entitles you to a certain amount of computer time. In order to help you conserve your time, the machines have an associated timer, which is initialized to 60 minutes when your machine is started.

The timer operates as a "dead man's switch," which decrements as your machine is running. If the timer is winding down to 0, you may choose to increase the timer.

INTERNATIONALIZATION

LANGUAGE SUPPORT

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

PER-USER LANGUAGE SELECTION

Users may prefer to use a different language for their desktop environment than the system-wide default. They may also want to set their account to use a different keyboard layout or input method.

Language Settings

In the GNOME desktop environment, the user may be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the Region & Language application. Run the command **gnome-control-center region**, or from the top bar, select (User) → Settings. In the window that opens, select Region & Language. Click the Language box and select the preferred language from the list that appears. This also updates the Formats setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including **gnome-terminal**, started inside it. However, they do not apply to that account if accessed through an **ssh** login from a remote system or a local text console (such as **tty2**).



NOTE

A user can make their shell environment use the same **LANG** setting as their graphical environment, even when they log in through a text console or over **ssh**. One way to do this is to place code similar to the following in the user's **~/.bashrc** file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
    | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, and other languages with a non-Latin character set may not display properly on local text consoles.

Individual commands can be made to use another language by setting the **LANG** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date
```

jeu. avril 24 17:55:01 CDT 2014

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to determine the current value of LANG and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 automatically uses the IBus input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The Region & Language application can also be used to enable alternative input methods. In the Region & Language application window, the Input Sources box shows what input methods are currently available. By default, English (US) may be the only available method. Highlight English (US) and click the keyboard icon to see the current keyboard layout.

To add another input method, click the + button at the bottom left of the Input Sources window. An Add an Input Source window will open. Select your language, and then your preferred input method or keyboard layout.

When more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese Japanese (Kana Kanji) input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may also find this useful. For example, under English (United States) is the keyboard layout English (international AltGr dead keys), which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



NOTE

Any Unicode character can be entered in the GNOME desktop environment if the user knows the character's Unicode code point, by typing **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl+Shift+U**, then **03bb**, then **Enter**.

SYSTEM-WIDE DEFAULT LANGUAGE SETTINGS

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, the **root** user can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it displays the current system-wide locale settings.

Introduction

To set the system-wide language, run the command **localectl set-locale LANG=locale**, where *locale* is the appropriate \$LANG from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in **/etc/locale.conf**.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from Region & Language and clicking the Login Screen button at the upper-right corner of the window. Changing the Language of the login screen will also adjust the system-wide default language setting stored in the **/etc/locale.conf** configuration file.



IMPORTANT

Local text consoles such as **tty2** are more limited in the fonts that they can display than **gnome-terminal** and **ssh** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through **localectl** for both local text virtual consoles and the X11 graphical environment. See the **localectl(1)**, **kbd(4)**, and **vconsole.conf(5)** man pages for more information.

LANGUAGE PACKS

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available language packs, run **yum langavailable**. To view the list of language packs currently installed on the system, run **yum langlist**. To add an additional language pack to the system, run **yum langinstall code**, where *code* is the code in square brackets after the language name in the output of **yum langavailable**.



REFERENCES

locale(7), **localectl(1)**, **kbd(4)**, **locale.conf(5)**, **vconsole.conf(5)**, **unicode(7)**, **utf-8(7)**, and **yum-langpacks(8)** man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in **localectl** can be found in the file **/usr/share/X11/xkb/rules/base.lst**.

LANGUAGE CODES REFERENCE

Language Codes

LANGUAGE	\$LANG VALUE
English (US)	en_US.utf8
Assamese	as_IN.utf8

LANGUAGE	\$LANG VALUE
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8

CHAPTER 1

INTRODUCING ANSIBLE

GOAL

Describe Ansible concepts and install Red Hat Ansible Engine.

OBJECTIVES

- Describe Ansible concepts, architecture, and common use cases.
- Install Ansible on a control node and describe the distinction between community Ansible and Red Hat Ansible Engine.

SECTIONS

- Overview of Ansible (and Quiz)
- Installing Ansible (and Guided Exercise)

OVERVIEW OF ANSIBLE

OBJECTIVE

After completing this section, students should be able to describe Ansible concepts, its architecture, and common use cases.

WHAT IS ANSIBLE?

Ansible is an open source automation platform. It is a *simple automation language* that can perfectly 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 different 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. Ansible can be used to orchestrate the entire application life cycle.

Ansible Is Agentless

Ansible is built around an agentless architecture. Typically, Ansible connects to the hosts it manages 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 is finished with 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 will do.
- *Perfect description of applications*: 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 in order 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 life cycle

Communication is the key to DevOps. Ansible is the first automation language that can be read and written across IT. It is also the only automation engine that can automate the application life cycle and continuous delivery pipeline from start to finish.

ANSIBLE CONCEPTS AND ARCHITECTURE

There are two types of machines in the Ansible architecture: *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. A control node could be an administrator's laptop, a system shared by a number of administrators, or a server running Red Hat Ansible Tower.

Managed hosts are listed in an *inventory*, which also organizes those systems into groups for easier collective management. The inventory can be defined in a static text file, or dynamically determined by scripts that get information from external sources.

Instead of writing complex scripts, Ansible users create high-level *plays* to ensure a host or group of hosts are 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 may ensure that a file exists and has particular permissions and contents, while a task using a different module may make certain that a particular file system is mounted. If the system is not in that state, the task should put it in that

state. If the system is already in that state, it does nothing. If a task fails, Ansible's default behavior is to abort the rest of the playbook for the hosts that had a failure.

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. This means that you should be able to run a playbook on the same hosts multiple times safely. When your systems are in the correct state the playbook should make no changes when you run it. There are a handful of modules that you can use to run arbitrary commands, but 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 or an ad hoc command, the control node connects to the managed host using SSH (by default) or WinRM. This means that clients do not need to have an Ansible-specific agent installed on managed hosts, and do not need to permit special network traffic to some nonstandard port.

Red Hat Ansible Tower is an enterprise framework to help you control, secure, and manage your Ansible automation at scale. You can use it to control who has access to run playbooks on which hosts, share the use of SSH credentials without allowing users to transfer or see their contents, log all of your Ansible jobs, and manage inventory, among many other things. It provides a web-based user interface (web UI) and a RESTful API. It is not a core part of Ansible, but a separate product that helps you use Ansible more effectively with a team or at a large scale.

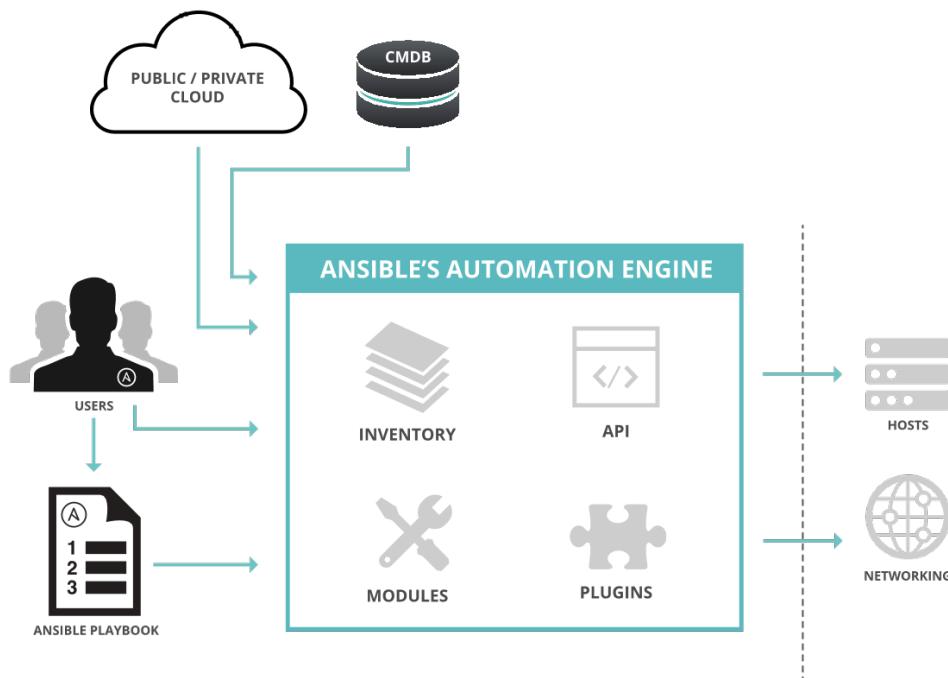


Figure 1.2: Ansible architecture

THE ANSIBLE WAY

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. Ansible's goal 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.

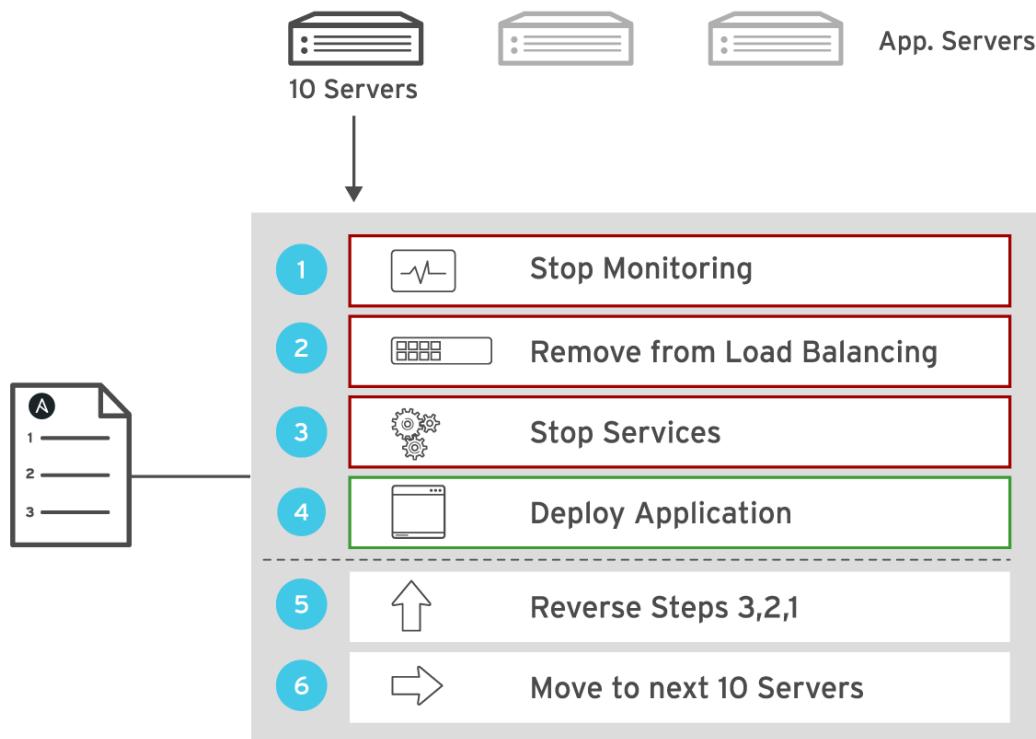


Figure 1.3: Ansible provides complete automation

USE CASES

Unlike some other tools, Ansible combines and unites 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 Red Hat Ansible Tower, teams can effectively manage the entire application life cycle from development to production.

Provisioning

Applications have to be deployed or installed on systems. Ansible and Red Hat Ansible Tower 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. Applications have to be deployed or installed on systems.

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 life cycle.

Security and Compliance

When your security policy is defined in Ansible Playbooks, scanning and remediation of site-wide security policies 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 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>

► QUIZ

OVERVIEW OF ANSIBLE

Choose the correct answer to the following questions:

- ▶ **1. Which of the following terms 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 of the following files defines the actions that Ansible performs on managed nodes?**
 - a. Host inventory
 - b. Manifest
 - c. Playbook
 - d. Script

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

► SOLUTION

OVERVIEW OF ANSIBLE

Choose the correct answer to the following questions:

► 1. Which of the following terms 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 of the following files defines the actions that Ansible performs on managed nodes?

- a. Host inventory
- b. Manifest
- c. Playbook
- d. Script

► 4. What syntax is used to define Ansible Playbooks?

- a. Bash
- b. Perl
- c. Python
- d. YAML

INSTALLING ANSIBLE

OBJECTIVES

After completing this section, students should be able to install Ansible on a control node and describe the distinction between community Ansible and Red Hat Ansible Engine.

Figure 1.3: Installing Ansible

ANSIBLE OR RED HAT ANSIBLE ENGINE?

Red Hat provides Ansible software in special channels as a convenience to Red Hat Enterprise Linux subscribers, and you can use these software packages normally.

However, if you want formal support for Ansible and its modules, Red Hat offers a special subscription for this, Red Hat Ansible Engine. This adds formal technical support with SLAs and a published scope of coverage for Ansible and its core modules. More information on the scope of this support is available at *Red Hat Ansible Engine Life Cycle* [<https://access.redhat.com/support/policy/updates/ansible-engine>].

CONTROL NODES

Ansible is simple to install. The Ansible software only needs to be installed on the control node (or nodes) from which Ansible will be run. Hosts that are managed by Ansible do not need to have Ansible installed. This installation involves relatively few steps and has minimal requirements.

The control node should be a Linux or UNIX system. Microsoft Windows is not supported as a control node, although Windows systems can be managed hosts.

Python 2 (version 2.7 or later) or Python 3 (version 3.5 or later) needs to be installed on the control node. To see whether the appropriate version of Python is installed on a Red Hat Enterprise Linux system, use the **yum** command.

```
[root@controlnode ~]# yum list installed python
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
python.x86_64      2.7.5-48.el7      installed
```

Information on how to install the *ansible* software package on a Red Hat Enterprise Linux system is available in the Knowledgebase article *How Do I Download and Install Red Hat Ansible Engine?* [<https://access.redhat.com/articles/3174981>].

Ansible is under rapid upstream development, and therefore Red Hat Ansible Engine has a rapid life cycle. More information on the current life cycle is available at <https://access.redhat.com/support/policy/updates/ansible-engine>.

Red Hat currently provides several channels for Red Hat Enterprise Linux 7 Server, as listed in the following table.

CHANNEL NAME	DESCRIPTION
rhel-7-server-ansible-2-rpms	The latest update of the Red Hat Ansible Engine 2 major release for RHEL 7. The minor release can change in this channel.
rhel-7-server-ansible-2.7-rpms	Red Hat Ansible Engine 2.7 for RHEL 7.
rhel-7-server-ansible-2.6-rpms	Red Hat Ansible Engine 2.6 for RHEL 7.
rhel-7-server-ansible-2.5-rpms	Red Hat Ansible Engine 2.5 for RHEL 7.
rhel-7-server-ansible-2.4-rpms	Red Hat Ansible Engine 2.4 for RHEL 7.

If you have standard Red Hat Enterprise Linux subscriptions, you can use these channels to install Ansible with limited support. If you need more comprehensive Ansible support, you can purchase full Red Hat Ansible Engine subscriptions and associate them with your systems before enabling the channels, as discussed in the Knowledgebase article.

If you have a Red Hat Ansible Engine subscription, the installation procedure for Red Hat Ansible Engine 2 is as follows:

1. Register your system to Red Hat Subscription Manager.

```
[root@host ~]# subscription-manager register
```

2. Attach your Red Hat Ansible Engine subscription. This command helps you find your Red Hat Ansible Engine subscription:

```
[root@host ~]# subscription-manager list --available
```

3. Use the pool ID of the subscription to attach the pool to the system.

```
[root@host ~]# subscription-manager attach --pool=<engine-subscription-pool>
```

4. Enable the Red Hat Ansible Engine repository.

```
[root@host ~]# subscription-manager repos \
> --enable rhel-7-server-ansible-2-rpms
```

5. Install Red Hat Ansible Engine.

```
[root@host ~]# yum install ansible
```

If you are using the version with limited support provided with your Red Hat Enterprise Linux subscription, use the following procedure:

1. Enable the Red Hat Ansible Engine repository.

```
[root@host ~]# subscription-manager refresh
[root@host ~]# subscription-manager repos \
> --enable rhel-7-server-ansible-2-rpms
```

2. Install Red Hat Ansible Engine.

```
[root@host ~]# yum install ansible
```

**IMPORTANT**

If you are already using Ansible but you installed it from the Red Hat Enterprise Linux 7 Extras channel, be aware that Ansible and its dependencies will no longer be updated through the Extras channel. The official channels discussed in this section replace that distribution method.

For more information, see the Knowledgebase article *Ansible deprecated in the Extras channel* [<https://access.redhat.com/articles/3359651>].

Ansible control nodes need to communicate with managed hosts over the network. SSH is used by default, but other protocols might be needed if network devices or Microsoft Windows systems are being managed. On Red Hat Enterprise Linux control nodes, if you are managing Microsoft Windows systems, you also need to have version 0.3.0 or later of the *python2-winrm* RPM package installed (which provides the *pywinrm* Python package).

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 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 it will run on them.

Linux and UNIX managed hosts need to have Python 2 (version 2.6 or later) or Python 3 (version 3.5 or later) installed for most modules to work. For Red Hat Enterprise Linux, install the *python* package.

If SELinux is enabled on the managed hosts, you also need to install the *libselinux-python* package before using modules that are related to any copy, file, or template functions. (Note that if the other Python components are installed, you can use Ansible modules such as *yum* or *package* to ensure that this package is also installed.)

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

**NOTE**

Some modules do not need Python. For example, arguments passed to the Ansible *raw* module are run directly through the configured remote shell instead of going through the module subsystem. This can be useful for managing devices that do not have Python available or cannot have Python installed, or for bootstrapping Python onto a system that does not have it.

However, the *raw* module is difficult to use in a safely idempotent way. If you can use a normal module instead, it is generally better to avoid using *raw* and similar command modules. This is discussed further later in the course.

Microsoft Windows-based Managed Hosts

Ansible includes a number of modules that are specifically designed for Microsoft Windows systems. These are listed in the Windows Modules [https://docs.ansible.com/ansible/2.7/modules/list_of_windows_modules.html] section of the Ansible module index.

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 PowerShell remoting configured. Ansible also requires at least .NET Framework 4.0 or later to be installed on 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-based managed hosts. More information is available on the Ansible web site at https://docs.ansible.com/ansible/2.7/user_guide/windows.html.

Managed Network Devices

You can also use Ansible automation to configure managed network devices such as routers and switches. Ansible includes a large number of 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 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 attend our alternative course *Ansible for Network Automation* (DO457) [<https://www.redhat.com/en/services/training/do457-ansible-network-automation>].



REFERENCES

`ansible(1)` man page

Top Support Policies for Red Hat Ansible Automation

<https://access.redhat.com/ansible-top-support-policies>

Installation Guide – Ansible Documentation

http://docs.ansible.com/ansible/2.7/installation_guide/intro_installation.html

Windows Guides – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/windows.html

Ansible for Networking Automation – Ansible Documentation

<https://docs.ansible.com/ansible/2.7/network/index.html>

► GUIDED EXERCISE

INSTALLING ANSIBLE

In this exercise, you will install Ansible on a control node running Red Hat Enterprise Linux.

OUTCOME

You should be able to install Ansible on a control node.

Log in to **workstation** as **student** using **student** as the password, and run **lab intro-install setup**. This setup script ensures that the managed host, **servera**, is reachable on the network.

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

- 1. Verify that Python 2.7 is installed on **workstation**.

```
[student@workstation ~]$ yum list installed python
```

- 2. Install Ansible on **workstation** so that it can be used as a control node.

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

- 3. Verify that Ansible installation is successful. Execute the **ansible** command with the **--version** option.

```
[student@workstation ~]$ ansible --version
ansible 2.7.1
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/student/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Sep 12 2018, 05:31:16) [GCC 4.8.5 20150623 (Red
  Hat 4.8.5-36)]
```

Cleanup

On **workstation**, run the **lab intro-install cleanup** script to clean up this exercise.

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

This concludes the guided exercise.

SUMMARY

In this chapter, you learned:

- Ansible is an open source automation platform that can adapt to many different workflows and environments.
- Ansible can be used to manage many different types of systems, including servers running Linux, Microsoft Windows, or UNIX, and network devices.
- Ansible Playbooks are human-readable text files that describe the desired state of an IT infrastructure.
- Ansible is built around an agentless architecture in which Ansible is installed on a control node and clients do not need any special agent software.
- 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 by Ansible.

CHAPTER 2

DEPLOYING ANSIBLE

GOAL

Configure Ansible to manage hosts and run ad hoc Ansible commands.

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.
- Run a single Ansible automation task using an ad hoc command and explain some use cases for ad hoc commands.

SECTIONS

- Building an Ansible Inventory (and Guided Exercise)
- Managing Ansible Configuration Files (and Guided Exercise)
- Running Ad Hoc Commands (and Guided Exercise)

LAB

- Deploying Ansible

BUILDING AN ANSIBLE INVENTORY

OBJECTIVE

After completing this section, students should be able to describe Ansible inventory concepts and manage a static inventory file.

Figure 2.0: Creating an Ansible inventory

THE INVENTORY

An *inventory* defines a collection of hosts that Ansible will manage. 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.

Host inventories can be defined in two different ways. A *static* host inventory can be defined by a text file. A *dynamic* host inventory can be generated by a script or other program as needed, using external information providers.

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 an INI-style format and a format expressed as a YAML document. The INI-style format is very common and will be used for most examples in this course.

In its simplest form, an INI-style static inventory file is a list of host names 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 host name 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, 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 more easily apply Ansible plays to specific hosts.

```
[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 of the hosts from the `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 host names 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*, inclusive. 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 earlier example:

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

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

Verifying the Inventory

When in doubt, use the **ansible** command to verify a machine's presence in the inventory:

```
[user@controlnode ~]$ ansible washington1.example.com --list-hosts
  hosts (1):
    washington1.example.com
[user@controlnode ~]$ ansible washington01.example.com --list-hosts
  [WARNING]: provided hosts list is empty, only localhost is available

  hosts (0):
```

You can run the following command to list all hosts in a group:

```
[user@controlnode ~]$ ansible canada --list-hosts
hosts (2):
ontario01.example.com
ontario02.example.com
```

**IMPORTANT**

If the inventory contains a host and a host group with the same name, the **ansible** command prints a warning and targets the host. The host group is ignored.

There are various ways to deal with this situation, the easiest being to 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 define a different location for inventory files in your Ansible configuration file. This is covered in the next section.

The **ansible** and **ansible-playbook** commands that you use to run Ansible ad hoc commands and playbooks later in the course can also 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.

Defining Variables in the Inventory

Values for variables used by playbooks can be specified in host inventory files. These variables only apply to specific hosts or host groups. Normally it is better to define these *inventory variables* in special directories and not directly in the inventory file. This topic is discussed in more depth elsewhere in the course.

DYNAMIC INVENTORY

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 scripts that are available from the upstream Ansible project. If those scripts 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.

This topic is discussed in more detail later in the course.

**REFERENCES****Inventory: Ansible Documentation**

http://docs.ansible.com/ansible/2.7/user_guide/intro_inventory.html

► GUIDED EXERCISE

BUILDING AN ANSIBLE INVENTORY

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

OUTCOMES

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

Log in as the student user on workstation and run **lab deploy-inventory setup**. This setup script ensures that the managed hosts, servera, serverb, serverc, serverd, are reachable on the network.

```
[student@workstation ~]$ lab deploy-inventory setup
```

- ▶ 1. Modify **/etc/ansible/hosts** to include servera.lab.example.com as a managed host.
 - 1.1. Add servera.lab.example.com to the end of the default inventory file, **/etc/ansible/hosts**.

```
[student@workstation ~]$ sudo vim /etc/ansible/hosts  
...output omitted...  
## db-[99:101]-node.example.com  
  
servera.lab.example.com
```

- 1.2. Continue editing the **/etc/ansible/hosts** inventory file by adding a [webservers] group to the bottom of the file with serverb.lab.example.com server as a group member.

```
[student@workstation ~]$ sudo vim /etc/ansible/hosts  
...output omitted...  
## db-[99:101]-node.example.com  
  
servera.lab.example.com  
  
[webservers]  
serverb.lab.example.com
```

- ▶ 2. Verify the managed hosts in the **/etc/ansible/hosts** inventory file.

- 2.1. Use the **ansible all --list-hosts** command to list all managed hosts in the default inventory file.

```
[student@workstation ~]$ ansible all --list-hosts  
hosts (2):  
servera.lab.example.com
```

```
serverb.lab.example.com
```

- 2.2. Use the **ansible ungrouped --list-hosts** command to list only managed hosts that do not belong to a group.

```
[student@workstation ~]$ ansible ungrouped --list-hosts
hosts (1):
servera.lab.example.com
```

- 2.3. Use the **ansible webservers --list-hosts** command to list only managed hosts that belong to the webservers group.

```
[student@workstation ~]$ ansible webservers --list-hosts
hosts (1):
serverb.lab.example.com
```

- 3. Create a custom static inventory file named **inventory** in the **/home/student/deploy-inventory** working directory.

Information about your four managed hosts is listed in the following table. You will 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 group us so that hosts in the United States can be managed as a group.

Server Inventory Specifications

HOST NAME	PURPOSE	LOCATION	ENVIRONMENT
servera.lab.example.com	Web server	Raleigh	Development
serverb.lab.example.com	Web server	Raleigh	Testing
serverc.lab.example.com	Web server	Mountain View	Production
serverd.lab.example.com	Web server	London	Production

- 3.1. Create the **/home/student/deploy-inventory** working directory.

```
[student@workstation ~]$ mkdir /home/student/deploy-inventory
```

- 3.2. Create an **inventory** file in the **/home/student/deploy-inventory** working directory. Use the Server Inventory Specifications table as a guide. Edit the **inventory** file and add the following content:

```
[student@workstation ~]$ cd /home/student/deploy-inventory
[student@workstation deploy-inventory]$ vim inventory
[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
```

- 4. Use variations of the **ansible host-or-group -i inventory --list-hosts** command to verify the managed hosts and groups in the custom **/home/student/deploy-inventory/inventory** inventory file.



IMPORTANT

Your **ansible** command must include the **-i inventory** option. This makes **ansible** use your **inventory** file in the current working directory instead of the system **/etc/ansible/hosts** inventory file.

- 4.1. Use the **ansible all -i inventory --list-hosts** command to list all managed hosts.

```
[student@workstation deploy-inventory]$ ansible all -i inventory --list-hosts
hosts (4):
    servera.lab.example.com
    serverb.lab.example.com
    serverc.lab.example.com
    serverd.lab.example.com
```

- 4.2. Use the **ansible ungrouped -i inventory --list-hosts** command to list all managed hosts listed in the inventory file but are not part of a group. There are no ungrouped managed hosts in this inventory file.

```
[student@workstation deploy-inventory]$ ansible ungrouped -i inventory \
> --list-hosts
[WARNINg]: No hosts matched, nothing to do
```

```
hosts (0):
```

- 4.3. Use the **ansible development -i inventory --list-hosts** command to list all managed hosts listed in the development group.

```
[student@workstation deploy-inventory]$ ansible development -i inventory \
> --list-hosts
hosts (1):
servera.lab.example.com
```

- 4.4. Use the **ansible testing -i inventory --list-hosts** command to list all managed hosts listed in the testing group.

```
[student@workstation deploy-inventory]$ ansible testing -i inventory \
> --list-hosts
hosts (1):
serverb.lab.example.com
```

- 4.5. Use the **ansible production -i inventory --list-hosts** command to list all managed hosts listed in the production group.

```
[student@workstation deploy-inventory]$ ansible production -i inventory \
> --list-hosts
hosts (2):
serverc.lab.example.com
serverd.lab.example.com
```

- 4.6. Use the **ansible us -i inventory --list-hosts** command to list all managed hosts listed in the us group.

```
[student@workstation deploy-inventory]$ ansible us -i inventory --list-hosts
hosts (3):
servera.lab.example.com
serverb.lab.example.com
serverc.lab.example.com
```

- 4.7. You are encouraged to experiment with other variations to confirm managed host entries in the custom inventory file.

Cleanup

On workstation, run the **lab deploy-inventory cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab deploy-inventory cleanup
```

This concludes the guided exercise.

MANAGING ANSIBLE CONFIGURATION FILES

OBJECTIVES

After completing this section, students should be able to describe where Ansible configuration files are located, how they are selected by Ansible, and edit them to apply changes to default settings.

Figure 2.0: Managing Ansible configuration files

CONFIGURING ANSIBLE

The behavior of an Ansible installation can be customized by modifying settings in the Ansible configuration file. Ansible chooses its configuration file from one of several possible locations on the control node.

Using `/etc/ansible/ansible.cfg`

The `ansible` package provides a base configuration file located at `/etc/ansible/ansible.cfg`. This file is used if no other configuration file is found.

Using `~/.ansible.cfg`

Ansible looks for a `.ansible.cfg` file in the user's home directory. This configuration is used instead of the `/etc/ansible/ansible.cfg` if it exists and if there is no `ansible.cfg` file in the current working directory.

Using `./ansible.cfg`

If an `ansible.cfg` file exists in the directory in which the `ansible` command is executed, it is used instead of the global file or the user's personal file. This allows administrators to create a directory structure where different environments or projects are stored in separate directories, with each directory containing a configuration file tailored with a unique set of settings.



IMPORTANT

The recommended practice is to create an `ansible.cfg` file in a directory from which you run Ansible commands. This directory would also contain any files used by your Ansible project, such as an inventory and a playbook. This is the most common location used for the Ansible configuration file. It is unusual to use a `~/.ansible.cfg` or `/etc/ansible/ansible.cfg` file in practice.

Using the `ANSIBLE_CONFIG` environment variable

You can use different configuration files by placing them in different directories and then executing Ansible commands from the appropriate directory, but this method can be restrictive and hard to manage as the number of configuration files grows. A more flexible option is to define the location of the configuration file with the `ANSIBLE_CONFIG` environment variable. When this variable is defined, Ansible uses the configuration file that the variable specifies instead of any of the previously mentioned configuration files.

CONFIGURATION FILE PRECEDENCE

The search order for a configuration file is the reverse of the preceding list. The first file located in the search order is the one that Ansible selects. Ansible only uses configuration settings from the first file that it finds.

Any file specified by the `ANSIBLE_CONFIG` environment variable overrides all other configuration files. If that variable is not set, the directory in which the `ansible` command was run is then checked for an `ansible.cfg` file. If that file is not present, the user's home directory is checked for a `.ansible.cfg` file. The global `/etc/ansible/ansible.cfg` file is only used if no other configuration file is found.

Because of the multitude of locations in which Ansible configuration files can be placed, it can be confusing which configuration file is being used by Ansible. You can run the `ansible --version` command to clearly identify which version of Ansible is installed, and which configuration file is being used.

```
[user@controlnode ~]$ ansible --version
ansible 2.7.0
  config file = /etc/ansible/ansible.cfg
  ...output omitted...
```

Another way to display the active Ansible configuration file is to use the `-v` option when executing Ansible commands on the command line.

```
[user@controlnode ~]$ ansible servers --list-hosts -v
Using /etc/ansible/ansible.cfg as config file
  ...output omitted...
```

Ansible only uses settings from the configuration file with the highest precedence. Even if other files with lower precedence exist, their settings are ignored and not combined with those in the selected configuration file. Therefore, if you choose to create your own configuration file in favor of the global `/etc/ansible/ansible.cfg` configuration file, you need to duplicate all desired settings from that file to your own user-level configuration file. Settings not defined in the user-level configuration file remain set to the built-in defaults, even if they are set to some other value by the global configuration file.

MANAGING SETTINGS IN THE CONFIGURATION FILE

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]** sets defaults for Ansible operation
- **[privilege_escalation]** 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 directives in this file are explained in the following table:

Ansible Configuration

DIRECTIVE	DESCRIPTION
inventory	Specifies the path to the inventory file.
remote_user	The name of the user to log in as on the managed hosts. If not specified, the current user's name is used.
ask_pass	Whether or not to prompt for an SSH password. Can be false if using SSH public key authentication.
become	Whether to automatically switch user on the managed host (typically to root) after connecting. This can also be specified by a play.
become_method	How to switch user (typically sudo , which is the default, but su is an option).
become_user	The user to switch to on the managed host (typically root , which is the default).
become_ask_pass	Whether to prompt for a password for your become_method . Defaults to false .

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. Some of the information needed includes:

- The location of the inventory that lists the managed hosts and host groups
- Which connection protocol to use to communicate with the managed hosts (by default, SSH), and whether or not a nonstandard network port is needed to connect to the server
- Which 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 or not to prompt for an SSH password or **sudo** password to log in or gain privileges

Inventory Location

In the **[defaults]** section, the **inventory** directive can point directly to a static inventory file, or to a directory that contains multiple static inventory files and/or 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.

By default, 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 that is not the case, you can configure Ansible to prompt the local user for the password used by the remote user by setting the directive **ask_pass = true**.

```
[defaults]
inventory = ./inventory

remote_user = root
ask_pass = true
```

Assuming that you are using a Linux control node and OpenSSH on your managed hosts, if you can log in as the remote user with a password then you can probably set up SSH key-based authentication, which would allow you to set **ask_pass = false**.

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 accomplish this.

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
      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-playbook** command with the **--ask-pass** option in order for the command to authenticate as the remote user.

Privilege Escalation

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 `root`. This can be set up in the **[privilege_escalation]** section of the Ansible configuration file.

To enable privilege escalation by default, set the directive **become = true** in the configuration file. Even if this is set by default, there are various ways 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** directive specifies how to escalate privileges. Several options are available, but the default is to use **sudo**. Likewise, the **become_user** directive 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** directive in the configuration file.

NOTE

On Red Hat Enterprise Linux 7, 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 directives 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 trade-offs 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
become_method = sudo
become_user = root
become_ask_pass = false
```

The following table summarizes some of the most commonly modified directives in the Ansible configuration file.

Ansible Settings

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

Non-SSH Connections

The protocol used by Ansible to connect to managed hosts is set by default to `smart`, which determines the most efficient way to use SSH. This can be set to other values in a number of ways.

For example, there is one exception to the rule that SSH is used by default. If you do not have `localhost` in your inventory, Ansible sets up an *implicit localhost* entry to allow you to run ad hoc commands and playbooks that target `localhost`. This special inventory entry is not included in the `all` or `ungrouped` host groups. In addition, instead of using the `smart` SSH connection type, Ansible connects to it using the special `local` connection type by default.

```
[user@controlnode ~]$ ansible localhost --list-hosts
[WARNING]: provided hosts list is empty, only localhost is available

hosts (1):
  localhost
```

The `local` connection type ignores the `remote_user` setting and runs commands directly on the local system. If privilege escalation is being used, it runs `sudo` from the user account that ran the Ansible command, not `remote_user`. This can lead to confusion if the two users have different `sudo` privileges.

If you want to make sure that you connect to `localhost` using SSH like other managed hosts, one approach is to list it in your inventory. But, this will include it in the `all` and `ungrouped` groups, which you may not want to do.

Another approach is to change the protocol used to connect to `localhost`. The best way to do this is to set the `ansible_connection` host variable for `localhost`. To do this, in the directory from which you run Ansible commands, create a `host_vars` subdirectory. In that subdirectory, create a file named `localhost` that contains the line `ansible_connection: smart`. This ensures that the `smart` (SSH) connection protocol is used instead of `local` for `localhost`.

You can use this the other way around as well. If you have `127.0.0.1` listed in your inventory, by default you will connect to it using `smart`. You can also create a `host_vars/127.0.0.1` file containing the line `ansible_connection: local` and it will use `local` instead.

Host variables are covered in more detail later in the course.



NOTE

You can also use *group variables* to change the connection type for an entire host group. This can be done by placing files with the same name as the group in a `group_vars` directory, and ensuring that those files contain settings for the connection variables.

For example, you might want all your Microsoft Windows managed hosts to use the `winrm` protocol and port 5986 for connections. To configure this, you could put all of those managed hosts in group `windows`, and then create a file named `group_vars/windows` containing the following lines:

```
ansible_connection: winrm
ansible_port: 5986
```

CONFIGURATION FILE COMMENTS

There are two comment characters allowed by Ansible configuration files: the hash or number sign (#), and the semicolon (:).

The number sign at the start of a line comments out the entire line. It must not be on the same line with a directive.

The semicolon character comments out everything to the right of it on the line. It can be on the same line as a directive, as long as that directive is to its left.



REFERENCES

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

Configuration file: Ansible Documentation

https://docs.ansible.com/ansible/2.7/installation_guide/intro_configuration.html

► GUIDED EXERCISE

MANAGING ANSIBLE CONFIGURATION FILES

In this exercise, you will customize your Ansible environment by editing an Ansible configuration file.

OUTCOMES

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

Log in as the student user on workstation and run **lab deploy-manage setup**. This script ensures that the managed host, servera, is reachable on the network.

```
[student@workstation ~]$ lab deploy-manage setup
```

- 1. Create the **/home/student/deploy-manage** directory, which will contain the files for this exercise. Change to this newly created directory.

```
[student@workstation ~]$ mkdir /home/student/deploy-manage  
[student@workstation ~]$ cd /home/student/deploy-manage
```

- 2. In your **/home/student/deploy-manage** directory, use a text editor to start editing a new file, **ansible.cfg**.

Create a **[defaults]** section in that file. In that section, add a line which uses the **inventory** directive to specify the **./inventory** file as the default inventory.

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

Save your work and exit the text editor.

- 3. In the **/home/student/deploy-manage** directory, use a text editor to start editing the new static inventory file, **inventory**.

The static inventory should contain three host groups:

- **[myself]** should contain the host localhost.
- **[intranetweb]** should contain the host servera.lab.example.com.
- **[everyone]** should contain the **myself** and **intranetweb** host groups.

- 3.1. In **/home/student/deploy-manage/inventory**, create the **myself** host group by adding the following lines:

```
[myself]
```

```
localhost
```

- 3.2. In **/home/student/deploy-manage/inventory**, create the **intranetweb** host group by adding the following lines:

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

- 3.3. In **/home/student/deploy-manage/inventory**, create the **everyone** host group by adding the following lines:

```
[everyone:children]
myself
intranetweb
```



NOTE

Remember that you do not need to create a special group to be able to select all hosts in the inventory file; you can use the **all** host group. We are doing this to practice creating groups of groups.

- 3.4. Confirm that your final **inventory** file looks like the following:

```
[myself]
localhost

[intranetweb]
servera.lab.example.com

[everyone:children]
myself
intranetweb
```

Save your work and exit the text editor.

- ▶ 4. Use the **ansible** command with the **--list-hosts** option to test the configuration of your inventory file's host groups. This will not actually connect to those hosts.

```
[student@workstation deploy-manage]$ ansible myself --list-hosts
hosts (1):
  localhost
[student@workstation deploy-manage]$ ansible intranetweb --list-hosts
hosts (1):
  servera.lab.example.com
[student@workstation deploy-manage]$ ansible everyone --list-hosts
hosts (2):
  localhost
  servera.lab.example.com
```

- ▶ 5. Open the **/home/student/deploy-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.

- 5.1. Create the **[privilege_escalation]** section in the **/home/student/deploy-manage/ansible.cfg** configuration file by adding the following entry:

```
[privilege_escalation]
```

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

```
become = true
```

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

```
become_method = sudo
```

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

```
become_user = root
```

- 5.5. Enable prompting for the privilege escalation password by setting the **become_ask_pass** directive to **true**.

```
become_ask_pass = true
```

- 5.6. Confirm that the complete **ansible.cfg** file looks like the following:

```
[defaults]
inventory = ./inventory

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

Save your work and exit the text editor.

- 6. Run the **ansible --list-hosts** command again to verify that you are now prompted for the **sudo** password.

When prompted for the sudo password, enter **student**, even though it is not used for this dry run.

```
[student@workstation deploy-manage]$ ansible intranetweb --list-hosts
SUDO password: student
hosts (1):
servera.lab.example.com
```

Cleanup

On workstation, run the **lab deploy-manage cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab deploy-manage cleanup
```

This concludes the guided exercise.

RUNNING AD HOC COMMANDS

OBJECTIVES

After completing this section, students should be able to run a single Ansible automation task using an ad hoc command and explain some use cases for ad hoc commands.

Figure 2.0: Running ad hoc commands

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 are useful for quick tests and changes. For example, you can use an ad hoc command to make sure that a certain line exists in the **/etc/hosts** file on a group of servers. You could use another ad hoc command to efficiently restart a service on many different machines, or ensure that a particular software package is up-to-date.

Ad hoc commands are very useful for quickly performing simple tasks with Ansible. They do have their limits, and in general you will want to use Ansible Playbooks to realize the full power of Ansible. In many situations, however, ad hoc commands are exactly the tool you need to perform simple tasks quickly.

Running Ad Hoc Commands

Use the **ansible** command to run ad hoc commands:

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

The *host-pattern* argument is used to specify the managed hosts on which the ad hoc command should be run. It could be a specific managed host or host group in the inventory. You have already seen this used in conjunction with the **--list-hosts** option, which shows you which hosts are matched by a particular host pattern. You have also already seen that you can use the **-i** option to specify a different inventory location to use than the default in the current Ansible configuration file.

The **-m** option takes as an argument the name of the *module* that Ansible should run on the targeted hosts. Modules are small programs that are executed to implement your task. Some modules need no additional information, but others need additional arguments to specify the details of their operation. The **-a** option takes a list of those arguments as a quoted string.

One of the simplest ad hoc commands uses the **ping** module. This module does not do an ICMP ping, but checks to see if you can run Python-based modules on managed hosts. For example, the following ad hoc command determines whether all managed hosts in the inventory can run standard modules:

```
[user@controlnode ~]$ ansible all -m ping
servera.lab.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
```

```
}
```

Performing Tasks with Modules Using Ad Hoc Commands

Modules are the tools that ad hoc commands use to accomplish tasks. Ansible provides hundreds of modules which do different things. You can usually find a tested, special-purpose module that does what you need as part of the standard installation.

The **ansible-doc -l** command lists all the modules that are installed on the system. You can then use **ansible-doc** to view the documentation of particular modules by name, and find information about what arguments the modules take as options. For example, the following command displays the documentation for the ping module:

```
[user@controlnode ~]$ ansible-doc ping
> PING      (/usr/lib/python2.7/site-packages/ansible/modules/system/ping.py)

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 log in 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 [win_ping] module instead. For Network
targets, use the [net_ping] module instead.

OPTIONS (= is mandatory):
- data
    Data to return for the `ping` return value.
    If this parameter is set to `crash`, the module will cause an exception.
    [Default: pong]

NOTES:
* For Windows targets, use the [win_ping] module instead.
* For Network targets, use the [net_ping] module instead.

AUTHOR: Ansible Core Team, Michael DeHaan

METADATA:
status:
- stableinterface
supported_by: core

EXAMPLES:
# Test we can logon to 'webservers' and execute python with json lib.
# ansible webservers -m ping

# Example from an Ansible Playbook
- ping:

# Induce an exception to see what happens
- ping:
    data: crash

RETURN VALUES:
ping:
description: value provided with the data parameter
returned: success
type: string
sample: pong
```

To learn more about modules, access the online Ansible documentation at http://docs.ansible.com/ansible/2.7/modules/modules_by_category.html.

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

Ansible Modules

MODULE CATEGORY	MODULES
Files modules	<ul style="list-style-type: none"> <code>copy</code>: Copy a local file to the managed host <code>file</code>: Set permissions and other properties of files <code>lineinfile</code>: Ensure a particular line is or is not in a file <code>synchronize</code>: Synchronize content using <code>rsync</code>
Software package modules	<ul style="list-style-type: none"> <code>package</code>: Manage packages using autodetected package manager native to the operating system <code>yum</code>: Manage packages using the YUM package manager <code>apt</code>: Manage packages using the APT package manager <code>dnf</code>: Manage packages using the DNF package manager <code>gem</code>: Manage Ruby gems <code>pip</code>: Manage Python packages from PyPI <code>yum</code>: Manage packages using the YUM package manager
System modules	<ul style="list-style-type: none"> <code>firewalld</code>: Manage arbitrary ports/services using <code>firewalld</code> <code>reboot</code>: Reboot a machine <code>service</code>: Manage services <code>user</code>: Add, remove, and manage user accounts
Net Tools modules	<ul style="list-style-type: none"> <code>get_url</code>: Download files via HTTP, HTTPS, or FTP <code>nmcli</code>: Manage networking <code>uri</code>: Interact with web services

Most modules take arguments. You can find the list of arguments available for a module in the module's documentation. Ad hoc commands pass arguments to modules using the `-a` option. When no argument is needed, omit the `-a` option from the ad hoc command. If multiple arguments need to be specified, supply them as a quoted space-separated list.

For example, the following ad hoc command uses the `user` module to ensure that the `newbie` user exists and has UID 4000 on `servera.lab.example.com`:

```
[user@controlnode ~]$ ansible -m user -a 'name=newbie uid=4000 state=present' \
> servera.lab.example.com
servera.lab.example.com | SUCCESS => {
    "changed": true,
    "comment": "",
    "createhome": true,
    "group": 4000,
    "home": "/home/newbie",
    "name": "newbie",
    "shell": "/bin/bash",
    "state": "present",
    "system": false,
    "uid": 4000
}
```

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 will do nothing. For example, if you run the previous ad hoc command again it should report no changes:

```
[user@controlnode ~]$ ansible -m user -a 'name=newbie uid=4000 state=present' \
> servera.lab.example.com
servera.lab.example.com | SUCCESS => {
    "append": false,
    "changed": false
    "comment": "",
    "group": 4000,
    "home": "/home/newbie",
    "move_home": false,
    "name": "newbie",
    "shell": "/bin/bash",
    "state": "present",
    "uid": 4000
}
```

Running Arbitrary Commands on Managed Hosts

The command module allows administrators to run arbitrary commands on the command line of managed hosts. The command to be run is specified as an argument to the module using the **-a** option. For example, the following command runs the **hostname** command on the managed hosts referenced by the **mymanagedhosts** host pattern.

```
[user@controlnode ~]$ ansible mymanagedhosts -m command -a /usr/bin/hostname
host1.lab.example.com | CHANGED | rc=0 >>
host1.lab.example.com
host2.lab.example.com | CHANGED | rc=0 >>
host2.lab.example.com
```

The previous ad hoc command example returned two lines of output for each managed host. The first line is a status report, which shows the name of the managed host that the ad hoc operation ran on, as well as the outcome of the operation. The second line is the output of the command executed remotely using the Ansible command module.

For better readability and parsing of ad hoc command output, administrators might find it useful to have a single line of output for each operation performed on a managed host. Use the **-o** option to display the output of Ansible ad hoc commands in a single line format.

```
[user@controlnode ~]$ ansible mymanagedhosts -m command -a /usr/bin/hostname -o
host1.lab.example.com | CHANGED | rc=0 >> (stdout) host1.lab.example.com
host2.lab.example.com | CHANGED | rc=0 >> (stdout) host2.lab.example.com
```

The command module allows administrators to quickly execute remote commands on managed hosts. These commands are not processed by the shell on the managed hosts. As such, they cannot access shell environment variables or perform shell operations, such as redirection and piping.

For situations where commands require shell processing, administrators can use the **shell** module. Like the command module, you pass the commands to be executed as arguments to the module in an ad hoc command. Ansible then executes the command remotely on the managed hosts. Unlike the command module, the commands are processed through a shell on the managed hosts. Therefore, shell environment variables are accessible and shell operations such as redirection and piping are also available for use.

The following example illustrates the difference between the `command` and `shell` modules. If you try to execute the built-in Bash command `set` with these two modules, it will only succeed with the `shell` module.

```
[user@controlnode ~]$ ansible localhost -m command -a set
localhost | FAILED | rc=2 >>
[Errno 2] No such file or directory
[user@controlnode ~]$ ansible localhost -m shell -a set
localhost | CHANGED | rc=0 >>
BASH=/bin/sh
BASHOPTS=cmdhist:extquote:force_fignore:hostcomplete:interact
ive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
...output omitted...
```

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



IMPORTANT

In most circumstances, it is a recommended practice that you avoid the `command`, `shell`, and `raw` "run command" modules.

Most other modules are idempotent and can perform change tracking automatically. They can test the state of systems and do nothing if those systems are already in the correct state. By contrast, it is much more complicated to use the "run command" modules in a way that is idempotent. Depending on them might make it harder for you to be confident that rerunning an ad hoc command or playbook would not cause an unexpected failure. When a `shell` or `command` module runs it typically reports a **CHANGED** status based on whether it thinks it affected machine state.

There are times when the "run command" modules are valuable tools and a good solution to a problem. If you do need to use them, it is probably best to try to use the `command` module first, resorting to `shell` or `raw` modules only if you need their special features.

CONFIGURING CONNECTIONS FOR AD HOC COMMANDS

The directives for managed host connections and privilege escalation can be configured in the Ansible configuration file, and they can also be defined using options in ad hoc commands. When defined using options in ad hoc commands, they take precedence over the directive configured in the Ansible configuration file. The following table shows the analogous command-line options for each configuration file directive.

Ansible Command-line Options

CONFIGURATION FILE DIRECTIVES	COMMAND-LINE OPTION
<code>inventory</code>	<code>-i</code>
<code>remote_user</code>	<code>-u</code>

CONFIGURATION FILE DIRECTIVES	COMMAND-LINE OPTION
<code>become</code>	<code>--become, -b</code>
<code>become_method</code>	<code>--become-method</code>
<code>become_user</code>	<code>--become-user</code>
<code>become_ask_pass</code>	<code>--ask-become-pass, -K</code>

Before configuring these directives using command-line options, their currently defined values can be determined by consulting the output of **ansible --help**.

```
[user@controlnode ~]$ ansible --help
...output omitted...
-b, --become           run operations with become (nopasswd implied)
--become-method=BECOME_METHOD
                      privilege escalation method to use (default=sudo),
                      valid choices: [ sudo | su | pbrun | pfexec | runas |
                      doas ]
--become-user=BECOME_USER
...output omitted...
-u REMOTE_USER, --user=REMOTE_USER
                      connect as this user (default=None)
```



REFERENCES

[ansible\(1\) man page](#)

Working with Patterns: Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/intro_patterns.html

Introduction to Ad-Hoc Commands: Ansible Documentation

http://docs.ansible.com/ansible/2.7/user_guide/intro_adhoc.html

Module Index: Ansible Documentation

http://docs.ansible.com/ansible/2.7/modules/modules_by_category

command - Executes a command on a remote node: Ansible Documentation

http://docs.ansible.com/ansible/2.7/modules/command_module.html

shell - Execute commands in nodes: Ansible Documentation

http://docs.ansible.com/ansible/2.7/modules/shell_module.html

► GUIDED EXERCISE

RUNNING AD HOC COMMANDS

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

OUTCOMES

You should be able to execute commands on managed hosts on an ad hoc basis using privilege escalation.

You will execute ad hoc commands on **workstation** and **servera** using the devops user account. This account has the same **sudo** configuration on both **workstation** and **servera**.

Log in as the student user on **workstation** and run **lab deploy-adhoc setup**. This script ensures that the managed host, **servera**, is reachable on the network. It also creates and populates the **/home/student/deploy-adhoc** working directory with materials used in this exercise.

```
[student@workstation ~]$ lab deploy-adhoc setup
```

- ▶ 1. Determine the **sudo** configuration for the devops account on both **workstation** and **servera**.
 - 1.1. Determine the **sudo** configuration for the devops account that was configured when **workstation** was built. Enter **student** if prompted for the password for the **student** account.

```
[student@workstation ~]$ sudo cat /etc/sudoers.d/devops  
[sudo] password for student: student  
devops ALL=(ALL) NOPASSWD: ALL
```

Note that the user has full **sudo** privileges but does not require password authentication.

- 1.2. Determine the **sudo** configuration for the devops account that was configured when **servera** was built.

```
[student@workstation ~]$ ssh devops@servera.lab.example.com  
[devops@servera ~]$ sudo cat /etc/sudoers.d/devops  
devops ALL=(ALL) NOPASSWD: ALL  
[devops@servera ~]$ exit
```

Note that the user has full **sudo** privileges but does not require password authentication.

- ▶ 2. Change directory to **/home/student/deploy-adhoc** and examine the contents of the **ansible.cfg** and **inventory** files.

```
[student@workstation ~]$ cd /home/student/deploy-adhoc
[student@workstation deploy-adhoc]$ cat ansible.cfg
[defaults]
inventory=inventory
[student@workstation deploy-adhoc]$ cat inventory
[control-node]
localhost

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

The configuration file uses the directory's **inventory** file as the Ansible inventory. Note that Ansible is not yet configured to use privilege escalation.

- ▶ 3. Using the **all** host group and the **ping** module, execute an ad hoc command that ensures all managed hosts can run Ansible modules using Python.

```
[student@workstation deploy-adhoc]$ ansible all -m ping
servera.lab.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

- ▶ 4. Using the **command** module, execute an ad hoc command on **workstation** to identify the user account that Ansible uses to perform operations on managed hosts. Use the **localhost** host pattern to connect to **workstation** for the ad hoc command execution. Because you are connecting locally, **workstation** is both the control node and managed host.

```
[student@workstation deploy-adhoc]$ ansible localhost -m command -a 'id'
localhost | CHANGED | rc=0 >>
uid=1000(student) gid=1000(student) groups=1000(student),10(wheel)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Notice that the ad hoc command was performed on the managed host as the **student** user.

- ▶ 5. Execute the previous ad hoc command on **workstation** but connect and perform the operation with the **devops** user account by using the **-u** option.

```
[student@workstation deploy-adhoc]$ ansible localhost -m command -a 'id' -u devops
localhost | CHANGED | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Notice that the ad hoc command was performed on the managed host as the **devops** user.

- 6. Using the command module, execute an ad hoc command on workstation to display the contents of the **/etc/motd** file. Execute the command using the devops account.

```
[student@workstation deploy-adhoc]$ ansible localhost -m command \
> -a 'cat /etc/motd' -u devops
localhost | CHANGED | rc=0 >>
```

Notice that the **/etc/motd** file is currently empty.

- 7. Using the copy module, execute an ad hoc command on workstation to change the contents of the **/etc/motd** file so that it consists of the string "Managed by Ansible" followed by a newline. Execute the command using the devops account, but do not use the **--become** option to switch to root. The ad hoc command should fail due to lack of permissions.

```
[student@workstation deploy-adhoc]$ ansible localhost -m copy \
> -a 'content="Managed by Ansible\n" dest=/etc/motd' -u devops
localhost | FAILED! => {
    "changed": false,
    "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
    "msg": "Destination /etc not writable"
}
```

The ad hoc command failed because the devops user does not have permission to write to the file.

- 8. Run the command again using privilege escalation. You could fix the settings in the **ansible.cfg** file, but for this example just use appropriate command-line options of the **ansible** command.

Using the copy module, execute the previous command on workstation to change the contents of the **/etc/motd** file so that it consists of the string "Managed by Ansible" followed by a newline. Use the devops user to make the connection to the managed host, but perform the operation as the root user using the **--become** option. The use of the **--become** option is sufficient because the default value for the **become_user** directive is set to root in the **/etc/ansible/ansible.cfg** file.

```
[student@workstation deploy-adhoc]$ ansible localhost -m copy \
> -a 'content="Managed by Ansible\n" dest=/etc/motd' -u devops --become
localhost | CHANGED => {
    "changed": true,
    "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 19,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1463518320.68-167292050637471/
source",
    "state": "file",
    "uid": 0
}
```

```
}
```

Note that the command succeeded this time because the ad hoc command was executed with privilege escalation.

- ▶ 9. Run the previous ad hoc command again on all hosts using the `all` host group. This ensures that `/etc/motd` on both `workstation` and `servera` consist of the text "Managed by Ansible".

```
[student@workstation deploy-adhoc]$ ansible all -m copy \
> -a 'content="Managed by Ansible\n" dest=/etc/motd' -u devops --become
localhost | SUCCESS => {
  "changed": false,
  "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
  "dest": "/etc/motd",
  "gid": 0,
  "group": "root",
  "mode": "0644",
  "owner": "root",
  "path": "/etc/motd",
  "secontext": "system_u:object_r:etc_t:s0",
  "size": 19,
  "state": "file",
  "uid": 0
}
servera.lab.example.com | CHANGED => {
  "changed": true,
  "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
  "dest": "/etc/motd",
  "gid": 0,
  "group": "root",
  "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
  "mode": "0644",
  "owner": "root",
  "secontext": "system_u:object_r:etc_t:s0",
  "size": 19,
  "src": "/home/devops/.ansible/tmp/ansible-tmp-1541518645.68-122144769062037/
source",
  "state": "file",
  "uid": 0
}
```

You should see **SUCCESS** for `localhost` and **CHANGED** for `servera`. However, `localhost` should report "**changed**": **false** because the file is already in the correct state. Conversely, `servera` should report "**changed**": **true** because the ad hoc command updated the file to the correct state.

- ▶ 10. Using the `command` module, execute an ad hoc command to run `cat /etc/motd` to verify that the contents of the file have been successfully modified on both `workstation` and `servera`. Use the `all` host group and the `devops` user to specify and make the connection to the managed hosts. You do not need privilege escalation for this command to work.

```
[student@workstation deploy-adhoc]$ ansible all -m command \
> -a 'cat /etc/motd' -u devops
servera.lab.example.com | CHANGED | rc=0 >>
```

```
Managed by Ansible  
localhost | CHANGED | rc=0 >>  
Managed by Ansible
```

Cleanup

On workstation, run the **lab deploy-adhoc cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab deploy-adhoc cleanup
```

This concludes the guided exercise.

▶ LAB

DEPLOYING ANSIBLE

PERFORMANCE CHECKLIST

In this lab, you will configure an Ansible control node for connections to inventory hosts and use ad hoc commands to perform actions on managed hosts.

OUTCOMES

You should be able to configure a control node to run ad hoc commands on managed hosts.

You will use Ansible to manage a number of hosts from `workstation.lab.example.com` as the `student` user. You will set up a project directory containing an `ansible.cfg` file with specific defaults, and an `inventory` directory containing an inventory file.

You will use ad hoc commands to ensure the `/etc/motd` file on all managed hosts consists of specified content.

Log in as the `student` user on `workstation` and run `lab deploy-review setup`. This script ensures that the managed hosts are reachable on the network.

```
[student@workstation ~]$ lab deploy-review setup
```

1. Verify that the `ansible` package is installed on the control node, and run the `ansible --version` command.
2. In the `student` user's home directory on `workstation`, `/home/student`, create a new directory named `deploy-review`. Change to that directory.
3. Create an `ansible.cfg` file in the `deploy-review` directory, which you will use to set the following Ansible defaults:
 - Connect to managed hosts as the `devops` user.
 - Use the `inventory` subdirectory to contain the inventory file.
 - Disable privilege escalation by default. If privilege escalation is enabled from the command line, configure default settings to have Ansible use the `sudo` method to switch to the `root` user account. Ansible should not prompt for the `devops` login password or the `sudo` password.

The managed hosts have been configured with a `devops` user who can log in using SSH key-based authentication and can run any command as `root` using the `sudo` command without a password.

4. Create the `/home/student/deploy-review/inventory` directory.
Download the <http://materials.example.com/labs/deploy-review/inventory> file and save it as a static inventory file named `/home/student/deploy-review/inventory/inventory`.
5. Execute an ad hoc command that targets the `all` host group to verify that `devops` is the remote user and that privilege escalation is disabled by default.

6. Execute an ad hoc command, targeting the `all` host group, that uses the `copy` module to modify the contents of the `/etc/motd` file on all hosts.
Use the `copy` module's `content` option to ensure the `/etc/motd` file consists of the string `"This server is managed by Ansible.\n"` as a single line. (The `\n` used with the `content` option causes the module to put a newline at the end of the string.)
You must request privilege escalation from the command line to make this work with your current `ansible.cfg` defaults.
7. If you run the same ad hoc command again, you should see that the `copy` module detects that the files are already correct and does not change them. Look for the ad hoc command to report `SUCCESS` and the line `"changed": false` for each managed host.
8. To confirm this another way, run an ad hoc command that targets the `all` host group, and which uses the `command` module to execute the `cat /etc/motd` command. Output from the `ansible` command should display the string `"This server is managed by Ansible."` for all hosts. You do not need privilege escalation for this ad hoc command.
9. Run `lab deploy-review grade` on workstation to check your work.

```
[student@workstation deploy-review]$ lab deploy-review grade
```

Cleanup

On workstation, run the `lab deploy-review cleanup` script to clean up the resources created in this lab.

```
[student@workstation ~]$ lab deploy-review cleanup
```

This concludes the lab.

► SOLUTION

DEPLOYING ANSIBLE

PERFORMANCE CHECKLIST

In this lab, you will configure an Ansible control node for connections to inventory hosts and use ad hoc commands to perform actions on managed hosts.

OUTCOMES

You should be able to configure a control node to run ad hoc commands on managed hosts.

You will use Ansible to manage a number of hosts from `workstation.lab.example.com` as the `student` user. You will set up a project directory containing an `ansible.cfg` file with specific defaults, and an `inventory` directory containing an inventory file.

You will use ad hoc commands to ensure the `/etc/motd` file on all managed hosts consists of specified content.

Log in as the `student` user on `workstation` and run `lab deploy-review setup`. This script ensures that the managed hosts are reachable on the network.

```
[student@workstation ~]$ lab deploy-review setup
```

1. Verify that the `ansible` package is installed on the control node, and run the `ansible --version` command.
 - 1.1. Verify that the `ansible` package is installed.

```
[student@workstation ~]$ yum list installed ansible
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
ansible.noarch      2.7.1-1.el7ae      @ansible
```

- 1.2. Run the `ansible --version` command to confirm the version of Ansible that is installed.

```
[student@workstation ~]$ ansible --version
ansible 2.7.1
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/student/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Sep 12 2018, 05:31:16) [GCC 4.8.5 20150623 (Red Hat 4.8.5-36)]
```

2. In the `student` user's home directory on `workstation`, `/home/student`, create a new directory named `deploy-review`. Change to that directory.

```
[student@workstation ~]$ mkdir /home/student/deploy-review  
[student@workstation ~]$ cd /home/student/deploy-review
```

3. Create an **ansible.cfg** file in the **deploy-review** directory, which you will use to set the following Ansible defaults:

- Connect to managed hosts as the devops user.
- Use the **inventory** subdirectory to contain the inventory file.
- Disable privilege escalation by default. If privilege escalation is enabled from the command line, configure default settings to have Ansible use the **sudo** method to switch to the **root** user account. Ansible should not prompt for the devops login password or the **sudo** password.

The managed hosts have been configured with a devops user who can log in using SSH key-based authentication and can run any command as **root** using the **sudo** command without a password.

- 3.1. Use a text editor to create the **/home/student/deploy-review/ansible.cfg** file. Create a **[defaults]** section. Add a **remote_user** directive to have Ansible use the **devops** user when connecting to managed hosts. Add an **inventory** directive to configure Ansible to use the **/home/student/deploy-review/inventory** directory as the default location for the inventory file.

```
[defaults]  
remote_user = devops  
inventory = inventory
```

- 3.2. In the **/home/student/deploy-review/ansible.cfg** file, create the **[privilege_escalation]** section and add the following entries to disable privilege escalation. Set the privilege escalation method to use the **root** account with **sudo** and without password authentication.

```
[privilege_escalation]  
become = False  
become_method = sudo  
become_user = root  
become_ask_pass = False
```

- 3.3. The completed **ansible.cfg** file should read as follows:

```
[defaults]  
remote_user = devops  
inventory = inventory  
  
[privilege_escalation]  
become = False  
become_method = sudo  
become_user = root  
become_ask_pass = False
```

Save your work and exit the editor.

4. Create the **/home/student/deploy-review/inventory** directory.

Download the <http://materials.example.com/labs/deploy-review/inventory> file and save it as a static inventory file named **/home/student/deploy-review/inventory/inventory**.

4.1. Create the **/home/student/deploy-review/inventory** directory.

```
[student@workstation deploy-review]$ mkdir inventory
```

4.2. Download the <http://materials.example.com/labs/deploy-review/inventory> file to the **/home/student/deploy-review/inventory** directory.

```
[student@workstation deploy-review]$ wget -O inventory/inventory \
> http://materials.example.com/labs/deploy-review/inventory
```

4.3. Inspect the contents of the **/home/student/deploy-review/inventory/inventory** file.

```
[student@workstation deploy-review]$ cat inventory/inventory
[internetweb]
serverb.lab.example.com

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

5. Execute an ad hoc command that targets the **all** host group to verify that devops is the remote user and that privilege escalation is disabled by default.

```
[student@workstation deploy-review]$ ansible all -m command -a 'id'
serverb.lab.example.com | CHANGED | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023

serverc.lab.example.com | CHANGED | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023

servera.lab.example.com | CHANGED | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023

serverd.lab.example.com | CHANGED | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Your results may be returned in a different order.

6. Execute an ad hoc command, targeting the `all` host group, that uses the `copy` module to modify the contents of the `/etc/motd` file on all hosts.

Use the `copy` module's `content` option to ensure the `/etc/motd` file consists of the string `"This server is managed by Ansible.\n"` as a single line. (The `\n` used with the `content` option causes the module to put a newline at the end of the string.)

You must request privilege escalation from the command line to make this work with your current `ansible.cfg` defaults.

```
[student@workstation deploy-review]$ ansible all -m copy \
> -a 'content="This server is managed by Ansible.\n" dest=/etc/motd' --become
servera.lab.example.com | CHANGED => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.56-280761564717921/
source",
    "state": "file",
    "uid": 0
}
serverb.lab.example.com | CHANGED => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.51-224886037138847/
source",
    "state": "file",
    "uid": 0
}
serverc.lab.example.com | CHANGED => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.56-242019037094684/
source",
```

```
"state": "file",
"uid": 0
}
serverd.lab.example.com | CHANGED => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.58-48889952156589/
source",
    "state": "file",
    "uid": 0
}
```

7. If you run the same ad hoc command again, you should see that the copy module detects that the files are already correct and does not change them. Look for the ad hoc command to report **SUCCESS** and the line "**changed": false** for each managed host.

```
[student@workstation deploy-review]$ ansible all -m copy \
> -a 'content="This server is managed by Ansible.\n" dest=/etc/motd' --become
servera.lab.example.com | SUCCESS => {
    "changed": false,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/motd",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "state": "file",
    "uid": 0
}
serverd.lab.example.com | SUCCESS => {
    "changed": false,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/motd",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "state": "file",
    "uid": 0
}
serverc.lab.example.com | SUCCESS => {
    "changed": false,
```

```
"checksum": "93d304488245bb2769752b95e0180607effc69ad",
"dest": "/etc/motd",
"gid": 0,
"group": "root",
"mode": "0644",
"owner": "root",
"path": "/etc/motd",
"secontext": "system_u:object_r:etc_t:s0",
"size": 35,
"state": "file",
"uid": 0
}
serverb.lab.example.com | SUCCESS => {
    "changed": false,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/motd",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "state": "file",
    "uid": 0
}
```

8. To confirm this another way, run an ad hoc command that targets the `all` host group, and which uses the `command` module to execute the `cat /etc/motd` command. Output from the `ansible` command should display the string "**This server is managed by Ansible.**" for all hosts. You do not need privilege escalation for this ad hoc command.

```
[student@workstation deploy-review]$ ansible all -m command -a 'cat /etc/motd'
serverb.lab.example.com | CHANGED | rc=0 >>
This server is managed by Ansible.

servera.lab.example.com | CHANGED | rc=0 >>
This server is managed by Ansible.

serverd.lab.example.com | CHANGED | rc=0 >>
This server is managed by Ansible.

serverc.lab.example.com | CHANGED | rc=0 >>
This server is managed by Ansible.
```

9. Run `lab deploy-review grade` on workstation to check your work.

```
[student@workstation deploy-review]$ lab deploy-review grade
```

Cleanup

On workstation, run the `lab deploy-review cleanup` script to clean up the resources created in this lab.

```
[student@workstation ~]$ lab deploy-review cleanup
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Any system on which Ansible is installed and which has access to the right configuration files and playbooks to manage remote systems (*managed hosts*) is called a *control node*.
- Managed hosts are defined in the *inventory*. Host patterns are used to reference managed hosts defined in an inventory.
- Inventories can be static files or dynamically generated by a program from an external source, such as a directory service or cloud management system.
- The location of the inventory is controlled by the Ansible configuration file in use, but most frequently is kept with the playbook files.
- Ansible looks for its configuration file in a number of places in order of precedence. The first configuration file found is used; all others are ignored.
- The **ansible** command is used to perform *ad hoc commands* on managed hosts.
- Ad hoc commands determine the operation to perform through the use of *modules* and their arguments.
- Ad hoc commands requiring additional permissions can make use of Ansible's *privilege escalation* features.

CHAPTER 3

IMPLEMENTING PLAYBOOKS

GOAL

Write a simple Ansible Playbook and run it to automate tasks on multiple hosts.

OBJECTIVES

- Write a basic Ansible Playbook and run it using the **ansible-playbook** command.
- Write a playbook that uses multiple plays and per-play privilege escalation.
- Effectively use **ansible-doc** to learn how to use new modules to implement tasks for a play.

SECTIONS

- Writing and Running Playbooks (and Guided Exercise)
- Implementing Multiple Plays (and Guided Exercise)

LAB

- Implementing Playbooks

WRITING AND RUNNING PLAYBOOKS

OBJECTIVE

After completing this section, students should be able to write a basic Ansible Playbook and run it using the **ansible-playbook** command.

Figure 3.0: Implementing Ansible Playbooks

ANSIBLE PLAYBOOKS AND AD HOC COMMANDS

Ad hoc commands can run a single, simple task against a set of targeted hosts as a one-time command. The real power of Ansible, however, is in learning how to use playbooks to run multiple, complex tasks against a set of targeted hosts in an easily repeatable manner.

A *play* is an ordered set of tasks that is run against hosts selected from your inventory. A *playbook* is a text file that contains a list of one or more plays to run in order.

Plays allow 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.

FORMAT OF AN ANSIBLE PLAYBOOK

To help you understand the format of a playbook, we will review an ad hoc command that you saw in a previous chapter:

```
[student@workstation ~]$ ansible -m user -a "name=newbie uid=4000 state=present" \
> servera.lab.example.com
```

This can be rewritten as a single task play and saved in a playbook. The resulting playbook appears as follows:

Example 3.1. A Simple Playbook

```
---
- name: Configure important user consistently
  hosts: servera.lab.example.com
  tasks:
    - name: newbie exists with UID 4000
      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 there are two basic rules.

- 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

Only the space character can be used for indentation; tab characters are not allowed.

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 autoindents subsequent lines.

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

A playbook begins with a line consisting of three dashes (---) as a start of document marker. It may end with three dots (...) as an end of document marker, although in practice this is often omitted.

In 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 Example 3.1, “A Simple Playbook”, 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 original example play has three keys, **name**, **hosts**, and **tasks**, because 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, the **name** attribute. The **name** key associates an arbitrary string with the play as a label. This identifies what the play is for. 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** attribute, which specifies the hosts against which the play's tasks are run. Like the argument for the **ansible** command, the **hosts** attribute 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 the **tasks** attribute, whose value specifies a list of the tasks to run for this play. This example has a single task, which runs the **user** module with specific arguments (to ensure user **newbie** exists and has UID 4000).

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

The **tasks** attribute 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 our 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 of your tasks to help document the purpose of each step of the automation process.
- **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** attribute with multiple tasks, using the **service** module to ensure that several network services are enabled to start at boot:

```
tasks:
  - name: web server is enabled
    service:
      name: httpd
      enabled: true

  - name: NTP server is enabled
    service:
      name: chronyd
      enabled: true

  - name: Postfix is enabled
    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.

The playbooks you have seen so far are basic examples, and you will see more sophisticated examples of what you can do with plays and tasks as this course continues.

RUNNING PLAYBOOKS

The **ansible-playbook** 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:

```
[student@workstation ~]$ ansible-playbook 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.

```
[student@workstation playdemo]$ cat webserver.yml
---
- name: play to setup web server
  hosts: servera.lab.example.com
  tasks:
    - name: latest httpd version installed
      yum:
        name: httpd
        state: latest
...
[student@workstation playdemo]$ ansible-playbook webserver.yml

PLAY [play to setup 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
```

Note that 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 **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 its specification was met. In this case, it means that the *httpd* package probably was not installed or was not the latest version.

In general, tasks in Ansible Playbooks are idempotent, and it is safe to run the 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:

```
[student@workstation playdemo]$ ansible-playbook webserver.yml

PLAY [play to setup 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
```

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

Increasing Output Verbosity

The default output provided by the **ansible-playbook** command does not provide detailed task execution information. The **ansible-playbook -v** command provides additional information, with up to four total levels.

Configuring the Output Verbosity of Playbook Execution

OPTION	DESCRIPTION
-v	The task results are displayed.
-vv	Both the task results and task configuration are displayed.
-vvv	Includes information about connections to managed hosts.
-vvvv	Adds extra verbosity options to the connection plug-ins, including the users being used in the managed hosts to execute scripts, and what scripts have been executed.

Syntax Verification

Prior to executing a playbook, it is good practice to perform a verification to ensure that the syntax of its contents is correct. The **ansible-playbook** command offers a **--syntax-check** option that you can use to verify the syntax of a playbook. The following example shows the successful syntax verification of a playbook.

```
[student@workstation ~]$ ansible-playbook --syntax-check webserver.yml

playbook: webserver.yml
```

When syntax verification 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 verification of a playbook where the space separator is missing after the **name** attribute for the play.

```
[student@workstation ~]$ ansible-playbook --syntax-check webserver.yml
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 setup web server
```

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

Executing a Dry Run

You can use the **-C** option to perform a *dry run* of the playbook execution. 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 *httpd* package is installed on a managed host. Note that the dry run reports that the task would effect a change on the managed host.

```
[student@workstation ~]$ ansible-playbook -C webserver.yml

PLAY [play to setup 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
```



REFERENCES

ansible-playbook(1) man page

Intro to Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_intro.html

Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks.html

Check Mode ("Dry Run") – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_checkmode.html

► GUIDED EXERCISE

WRITING AND RUNNING PLAYBOOKS

In this exercise, you will 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-playbook** command.

Log in as the **student** user on **workstation** and run **lab playbook-basic setup**. This setup script ensures that the managed hosts, **serverc.lab.example.com** and **serverd.lab.example.com**, are configured for the lab and are reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed in the working directory on the control node.

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

The **/home/student/playbook-basic** working directory has been created on **workstation** for this exercise. This directory has already been populated with an **ansible.cfg** configuration file, and also an **inventory** inventory file, which defines a **web** group that includes both managed hosts listed above as members.

In this directory, use a text editor to create a playbook named **site.yml**. This playbook contains one play, which should target members of the **web** host group. The playbook should use tasks to ensure that the following conditions are met on the managed hosts:

1. The **httpd** package is present, using the **yum** module.
2. The local **files/index.html** file is copied to **/var/www/html/index.html** on each managed host, using the **copy** module.
3. The **httpd** service is started and enabled, using the **service** module.

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

After the playbook is written, verify its syntax and then use **ansible-playbook** to run the playbook to implement the configuration.

- 1. To make all playbook exercises easier, if you use the **Vi** text editor you may want to use it to edit your **~/.vimrc** file (create it if necessary), to ensure it contains the following line:

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

This is optional, but it will set up the **vi** command so that the **Tab** key automatically indents using two space characters for YAML files. This may make it easier for you to edit Ansible Playbooks.

- 2. Change to the **/home/student/playbook-basic** directory.

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

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

```
---
```

- 3.2. 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
```

- 3.3. 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 now appear as follows:

```
---
```

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

- 4. 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.

- 4.1. Add a **tasks** keyword indented by two spaces (aligned with the **hosts** keyword) to start the list of tasks. Your file should now appear as follows:

```
---
```

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

- 4.2. 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 **httpd package is present**. Use the yum 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 appear as follows:

```
- name: httpd package is present
  yum:
    name: httpd
    state: present
```

- 4.3. 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 copy module. The

module keywords should set the **src** key to **files/index.html** and the **dest** key to **/var/www/html/index.html**. The task should appear as follows:

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

- 4.4. 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 **httpd is started**. Use the **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 appear as follows:

```
- name: httpd is started
  service:
    name: httpd
    state: started
    enabled: true
```

- 4.5. 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: httpd package is present
      yum:
        name: httpd
        state: present

    - name: correct index.html is present
      copy:
        src: files/index.html
        dest: /var/www/html/index.html

    - name: httpd is started
      service:
        name: httpd
        state: started
        enabled: true
```

Save the file and exit your text editor.

- 5. Before running your playbook, run the **ansible-playbook --syntax-check site.yml** command to verify that its syntax is correct. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation playbook-basic]$ ansible-playbook --syntax-check site.yml
playbook: site.yml
```

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

```
[student@workstation playbook-basic]$ ansible-playbook site.yml

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

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

TASK [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 [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
serverd.lab.example.com      : ok=4      changed=3      unreachable=0      failed=0
```

- 7. 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-playbook site.yml

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

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

TASK [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 [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
serverd.lab.example.com      : ok=4      changed=0      unreachable=0      failed=0
```

- 8. Use the **curl** command to verify that both `serverc` and `serverd` 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.
```

Cleanup

On workstation, run the **lab playbook-basic cleanup** script to clean up the resources created in this exercise.

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

This concludes the guided exercise.

IMPLEMENTING MULTIPLE PLAYS

OBJECTIVES

After completing this section, students should be able to:

- Write a playbook that uses multiple plays and per-play privilege escalation.
- Effectively use **ansible-doc** to learn how to use new modules to implement tasks for a play.

Figure 3.0: Implementing multiple plays

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 may apply its tasks to a separate set of hosts.

This can be very useful when orchestrating a complex deployment which may 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
      yum:
        name: httpd
        status: present

    - name: second task
      service:
        name: httpd
        enabled: true

- name: second play
  hosts: database.example.com
  tasks:
    - name: first task
```

```
service:
  name: mariadb
  enabled: true
```

REMOTE USERS AND PRIVILEGE ESCALATION IN PLAYS

Plays can use different remote users or privilege escalation settings for a play than what is specified by the defaults in the configuration file. These are set 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. As with ad hoc commands, the user account used for task execution depends on various keywords in the Ansible configuration file, **/etc/ansible/ansible.cfg**. The user that runs the tasks can be defined by the **remote_user** keyword. However, if privilege escalation is enabled, other keywords such **become_user** can also have an impact.

If the remote user defined in the Ansible configuration for task execution is not suitable, it can be overridden by using the **remote_user** keyword within a play.

```
remote_user: remoteuser
```

Privilege Escalation Attributes

Additional keywords are also available to define privilege escalation parameters from within a playbook. The **become** boolean keyword can be used to enable or disable privilege escalation regardless of how it is defined in the Ansible 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, the **become_method** keyword can be used to define the privilege escalation method to use during a specific play. The example below specifies that **sudo** be used for privilege escalation.

```
become_method: sudo
```

Additionally, with privilege escalation enabled, the **become_user** keyword can define the user account to use for privilege escalation within the context of a specific play.

```
become_user: privileged_user
```

The following example demonstrates the use of these keywords in a play:

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

  tasks:
    - name: server.example.com in /etc/hosts
```

```
lineinfile:
  path: /etc/hosts
  line: '192.0.2.42 server.example.com server'
  state: present
```

FINDING MODULES FOR TASKS

Module Documentation

The large number of modules packaged with Ansible provides administrators with many tools for common administrative tasks. Earlier in this course, we discussed the Ansible documentation website at <http://docs.ansible.com>. The **Module Index** on the website is an easy way to browse the list of modules shipped with Ansible. For example, modules for user and service management can be found under *Systems Modules* and modules for database administration can be found under *Database Modules*.

For each module, the Ansible documentation website provides a summary of its functions and instructions on how each specific function can be invoked with options to the module. The documentation also provides useful examples that show you how to use each module and how to set their keywords in a task.

You have already worked with the **ansible-doc** command to look up information about modules installed on the local system. As a review, to see a list of the modules available on a control node, run the **ansible-doc -l** command. This displays a list of module names and a synopsis of their functions.

```
[student@workstation modules]$ ansible-doc -l
a10_server          Manage A10 Networks ... devices' server object.
a10_server_axapi3   Manage A10 Networks ... devices
a10_service_group   Manage A10 Networks ... devices' service groups.
a10_virtual_server  Manage A10 Networks ... devices' virtual servers.
...output omitted...
zfs_facts            Gather facts about ZFS datasets.
znode                Create, ... and update znodes using ZooKeeper
zpool_facts          Gather facts about ZFS pools.
zypper               Manage packages on SUSE and openSUSE
zypper_repository    Add and remove Zypper repositories
```

Use the **ansible-doc [module name]** command to display detailed documentation for a module. Like the Ansible documentation website, the command provides a synopsis of the module's function, details of its various options, and examples. The following example shows the documentation displayed for the yum module.

```
[student@workstation modules]$ ansible-doc yum
> YUM      (/usr/lib/python2.7/site-packages/ansible/modules/packaging/os/yum.py)

Installs, upgrade, downgrades, removes, and lists packages and groups
with the 'yum' package manager. This module only works on Python 2. If
you require Python 3 support see the [dnf] module.

* note: This module has a corresponding action plugin.

OPTIONS (= is mandatory):

- allow_downgrade
  Specify if the named package and version is allowed to downgrade a maybe
```

```
already installed higher version of that package. Note that setting
allow_downgrade=True can make this module behave in a non-idempotent way.
The task could end up with a set of packages that does not match the
complete list of specified packages to install (because dependencies
between the downgraded package and others can cause changes to the
packages which were in the earlier transaction).
[Default: no]
type: bool
version_added: 2.4

- autoremove
  If `yes', removes all "leaf" packages from the system that were
  originally installed as dependencies of user-installed packages but which
  are no longer required by any such package. Should be used alone or when
  state is `absent'
  NOTE: This feature requires yum >= 3.4.3 (RHEL/CentOS 7+)
  [Default: False]
  type: bool
  version_added: 2.7

...output omitted...

EXAMPLES:
- name: install the latest version of Apache
  yum:
    name: httpd
    state: latest

- name: ensure a list of packages installed
  yum:
    name: "{{ packages }}"
  vars:
    packages:
      - httpd
      - httpd-tools

- name: remove the Apache package
  yum:
    name: httpd
    state: absent

...output omitted...
```

The **ansible-doc** command also offers the **-s** option, which produces example output that can serve as a model for how to use a particular module in a playbook. 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 remind administrators of the use of each option. The following example shows this output for the yum module.

```
[student@workstation ~]$ ansible-doc -s yum
- name: Manages packages with the `yum' package manager
  yum:
    allow_downgrade:      # Specify if the named package ...
    autoremove:           # If `yes', removes all "leaf" packages ...
    bugfix:               # If set to `yes', ...
    conf_file:            # The remote yum configuration file ...
```

```
disable_excludes:      # Disable the excludes ...
disable_gpg_check:   # Whether to disable the GPG ...
disable_plugin:       # `Plugin` name to disable ...
disablerepo:          # `Repopid` of repositories ...
download_only:        # Only download the packages, ...
enable_plugin:        # `Plugin` name to enable ...
enablerrepo:          # `Repopid` of repositories to enable ...
exclude:              # Package name(s) to exclude ...
installroot:          # Specifies an alternative installroot, ...
list:                 # Package name to run ...
name:                 # A package name or package specifier ...
releasever:           # Specifies an alternative release ...
security:             # If set to `yes`, ...
skip_broken:          # Skip packages with ...
state:                # Whether to install ... or remove ... a package.
update_cache:          # Force yum to check if cache ...
update_only:           # When using latest, only update ...
use_backend:           # This module supports `yum` ...
validate_certs:        # This only applies if using a https url ...
```

Module Maintenance

Ansible ships with a large number of modules that can be used for many tasks. The upstream community is very active, and these modules may be in different stages of development. The **ansible-doc** documentation for the module is expected to specify who maintains that module in the upstream Ansible community, and what its development status is. This is indicated in the **METADATA** section at the end of the output of **ansible-doc** for that module.

The **status** field records the development status of the module:

- **stableinterface**: The module's keywords are stable, and every effort will be made not to remove keywords or change their meaning.
- **preview**: The module is in technology preview, and might be unstable, its keywords might change, or it might require libraries or web services that are themselves subject to incompatible changes.
- **deprecated**: The module is deprecated, and will no longer be available in some future release.
- **removed**: The module has been removed from the release, but a stub exists for documentation purposes to help former users migrate to new modules.



NOTE

The **stableinterface** status only indicates that a module's interface is stable, it does not rate the module's code quality.

The **supported_by** field records who maintains the module in the upstream Ansible community. Possible values are:

- **core**: Maintained by the "core" Ansible developers upstream, and always included with Ansible.
- **curated**: Modules submitted and maintained by partners or companies in the community. Maintainers of these modules must watch for any issues reported or pull requests raised against the module. Upstream "core" developers review proposed changes to curated modules after the community maintainers have approved the changes. Core committers also ensure that any

issues with these modules due to changes in the Ansible engine are remediated. These modules are currently included with Ansible, but might be packaged separately at some point in the future.

- **community:** Modules not supported by the core upstream developers, partners, or companies, but maintained entirely by the general open source community. Modules in this category are still fully usable, but the response rate to issues is purely up to the community. These modules are also currently included with Ansible, but will probably be packaged separately at some point in the future.

The upstream Ansible community has an issue tracker for Ansible and its integrated modules at <https://github.com/ansible/ansible/issues>.

Sometimes, a module does not exist for something you want to do. As an end user, you can also write your own private modules, or get modules from a third party. Ansible searches for custom modules in the location specified by the `ANSIBLE_LIBRARY` environment variable, or if that is not set, by a `library` keyword in the current Ansible configuration file. Ansible also searches for custom modules in the `./library` directory relative to the playbook currently being run.

```
library = /usr/share/my_modules
```

Information on writing modules is beyond the scope of this course. Documentation on how to do this is available at https://docs.ansible.com/ansible/2.7/dev_guide/developing_modules.html.

**IMPORTANT**

Use the **ansible-doc** command to find and learn how to use modules for your tasks.

When possible, try to avoid the **command**, **shell**, and **raw** modules in playbooks, even though they might seem simple to use. Because these take arbitrary commands, it is very easy to write non-idempotent playbooks with these modules.

For example, the following task using the **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
  shell: echo "nameserver 192.0.2.1" > /etc/resolv.conf
```

There are several ways to write tasks using the **shell** module in an idempotent manner, and sometimes making those changes and using **shell** is the best approach. A quicker solution may be to use **ansible-doc** to discover the **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
  copy:
    dest: /etc/resolv.conf
    content: "nameserver 192.0.2.1\n"
```

The **copy** module tests to see if the state has already been met, and if so, it makes no changes. The **shell** module allows a lot of flexibility, but also requires more attention to ensure that it runs in an idempotent way.

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

PLAYBOOK SYNTAX VARIATIONS

The last part of this chapter 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 or hash symbol (#) is a comment. If there is content to the left of the comment, precede the number symbol 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 there are spaces contained in the string. If desired, you can enclose strings in either double quotes or single quotes.

```
this is a string
```

```
'this is another string'
```

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

There are two ways to write multiline strings. 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
    a very long,
    long, long, long
    sentence.
```

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 curly braces, as follows:

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

In most cases the inline block format should be avoided 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 key=value Playbook Shorthand

Some playbooks might use an older 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  
    service: name=httpd enabled=true state=started
```

Normally you would write the same task like as follows:

```
tasks:  
  - name: normal form  
    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 easier to differentiate. Your eyes can run straight down the play with less left-to-right motion. Also, the normal syntax is native YAML; the shorthand form is not. Syntax highlighting tools in modern 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 older playbooks from other people, and the syntax does still function.



REFERENCES

ansible-playbook(1) and **ansible-doc(1)** man pages

Intro to Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_intro.html

Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks.html

Developing Modules – Ansible Documentation

https://docs.ansible.com/ansible/2.7/dev_guide/developing_modules.html

Module Support – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/modules_support.html

► GUIDED EXERCISE

IMPLEMENTING MULTIPLE PLAYS

In this exercise, you will write and use a playbook containing multiple plays, and use it to perform administration tasks on managed hosts.

OUTCOMES

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

Log in as the **student** user on **workstation** and run **lab playbook-multi setup**. This setup script ensures that the managed host, **servera.lab.example.com**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory file are installed on the control node.

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

A developer responsible for your company's intranet web site has asked you to write a playbook to help automate the setup of the server environment on **servera.lab.example.com**.

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

In this directory, create a playbook named **intranet.yml** which contains two plays. The first play requires privilege escalation and must perform the following tasks in the specified order:

1. Use the **yum** module to ensure that the latest versions of the **httpd** and **firewalld** packages are installed.
2. Ensure that the **firewalld** service is enabled and started.
3. Ensure that **firewalld** is configured to allow connections to the **httpd** service.
4. Ensure that the **httpd** service is enabled and started.
5. Ensure that the managed host's **/var/www/html/index.html** file consists of the content "**Welcome to the example.com intranet!**".

The second play does not require privilege escalation and should run a single task using the **uri** module to confirm that the URL **http://servera.lab.example.com** returns an HTTP status code of **200**. To validate the web server's content, configure the **uri** module to return the web request content as part of the task results.

According to recommended practices, plays and tasks should have names that document their purpose, but this is not required. The example solution names plays and tasks.

Do not forget that you can use the **ansible-doc** command to get help with finding and using the modules for your tasks.

After you have written the playbook, verify its syntax and then execute the playbook to configure and test the web server. Use the **-v** option with the **ansible-playbook** command to display the

response from the web server. Verify the web server's response matches the expected value of "Welcome to the example.com intranet!\n".

- 1. Change to the working directory, /home/student/playbook-multi.

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

- 2. Create a new playbook, /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.

- 2.1. Create and open a new playbook, /home/student/playbook-multi/intranet.yml, and add a line consisting of three dashes to the beginning of the file to indicate the start of the YAML file.

```
---
```

- 2.2. Add the following line to the /home/student/playbook-multi/intranet.yml file to denote the start of a play with a name of **Enable intranet services**.

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

- 2.3. Add the following line to indicate that the play applies to the servera.lab.example.com managed host. Be sure to 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
```

- 2.4. Add the following line to enable privilege escalation. Be sure to indent the line with two spaces (aligning with the keywords above it) to indicate it is part of the first play.

```
become: yes
```

- 3. 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:
```

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

- 4.1. Under the **tasks** keyword in the first play, add the following lines to the /home/student/playbook-multi/intranet.yml file. This creates the task that ensures that the latest versions of the *httpd* and *firewalld* packages are installed.

Be sure to indent the first line of the task with four spaces, a dash, and a space. This indicates that the task is an item in the **tasks** list for the first play.

The first line provides a descriptive name for the task. The second line is indented with six spaces and calls the *yum* module. The next line is indented eight spaces and is a **name** keyword. It specifies which packages the *yum* module should ensure are up-to-date. The *yum* module's **name** keyword (which is different from the task name)

can take a list of packages, which is indented ten spaces on the two following lines. After the list, the 8-space indented **state** keyword specifies that the **yum** module should ensure that the latest version of the packages be installed.

```
- name: latest version of httpd and firewalld installed
  yum:
    name:
      - httpd
      - firewalld
    state: latest
```

- 5. Add a task to the first play's list that ensures that the correct content is in **/var/www/html/index.html**.

- 5.1. Add the following lines to the **/home/student/playbook-multi/intranet.yml** file to create the task that confirms the **/var/www/html/index.html** file is populated with the correct content. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with six spaces and calls the **copy** module. The remaining entries are indented with eight spaces and pass the necessary arguments to ensure that the correct content is in the web page.

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

- 6. Define two more tasks in the play to ensure that the **firewalld** service is running and will start on boot, and will allow connections to the **httpd** service.

- 6.1. Add the following lines to the **/home/student/playbook-multi/intranet.yml** file to create the task for ensuring that the **firewalld** service is enabled and running. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with eight spaces and calls the **service** module. The remaining entries are indented with ten spaces and pass the necessary arguments to ensure that the **firewalld** service is enabled and started.

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

- 6.2. Add the following lines to the **/home/student/playbook-multi/intranet.yml** file to create the task to ensure **firewalld** allows HTTP connections from remote systems. Be sure to indent the line with four spaces, a dash,

and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with six spaces and calls the `firewalld` module. The remaining entries are indented with eight spaces and pass the necessary arguments to ensure that remote HTTP connections are permanently allowed.

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

- ▶ 7. Add a final task to the first play's list that ensures that the `httpd` service is running and will start at boot.

7.1. Add the following lines to the `/home/student/playbook-multi/intranet.yml` file to create the task to ensure the `httpd` service is enabled and running. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

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

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

- ▶ 8. In `/home/student/playbook-multi/intranet.yml`, define a second play targeted at `localhost` which will test the intranet web server. It does not need privilege escalation.

8.1. Add the following line to the `/home/student/playbook-multi/intranet.yml` file to denote the start of a second play.

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

8.2. Add the following line to the `/home/student/playbook-multi/intranet.yml` file to indicate that the play applies to the `localhost` managed host. Be sure to indent the line with two spaces to indicate that it is contained by the second play.

```
hosts: localhost
```

8.3. Add the following line to the `/home/student/playbook-multi/intranet.yml` file to disable privilege escalation. Be sure to align the indentation of the `become` keyword with the `hosts` keyword above it.

```
become: no
```

- 9. Add the following line to the **/home/student/playbook-multi/intranet.yml** file to define the beginning of the **tasks** list. Be sure to indent the line with two spaces to indicate that it is contained by the second play.

```
tasks:
```

- 10. Add a single task to the second play, and use the `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.

- 10.1. Add the following lines to the **/home/student/playbook-multi/intranet.yml** file to create the task for verifying web services from the control node. Be sure to indent the first line with four spaces, a dash, and a space. This indicates that the task is an item in the second play's **tasks** list.

The first line provides a descriptive name for the task. The second line is indented with six spaces and calls the `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.

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

- 11. Verify that the final **/home/student/playbook-multi/intranet.yml** playbook reflects the following structured content.

```
---
- name: Enable intranet services
  hosts: servera.lab.example.com
  become: yes
  tasks:
    - name: latest version of httpd and firewalld installed
      yum:
        name:
          - httpd
          - firewalld
        state: latest

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

    - name: firewalld enabled and running
      service:
        name: firewalld
        enabled: true
        state: started

    - name: firewalld permits http service
```

```
firewalld:  
  service: http  
  permanent: true  
  state: enabled  
  immediate: yes  
  
- name: httpd enabled and running  
  service:  
    name: httpd  
    enabled: true  
    state: started  
  
- name: Test intranet web server  
  hosts: localhost  
  become: no  
  tasks:  
    - name: connect to intranet web server  
      uri:  
        url: http://servera.lab.example.com  
        return_content: yes  
        status_code: 200
```

- 12. Save and close the file.
- 13. Run the **ansible-playbook --syntax-check** command to verify the syntax of the **intranet.yml** playbook.
- ```
[student@workstation playbook-multi]$ ansible-playbook --syntax-check intranet.yml
playbook: intranet.yml
```
- 14. Execute the playbook using the **-v** option to output detailed results for each task. Read through the output generated to ensure that all tasks completed successfully. Verify that an HTTP GET request to <http://servera.lab.example.com> provides the correct content.

```
[student@workstation playbook-multi]$ ansible-playbook -v intranet.yml
...output omitted...

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] => {"changed": true, ...output omitted...}

TASK [test html page is installed] *****
changed: [servera.lab.example.com] => {"changed": true, ...output omitted...}

TASK [firewalld enabled and running] *****
ok: [servera.lab.example.com] => {"changed": false, ...output omitted...}

TASK [firewalld permits http service] *****
```

```

changed: [servera.lab.example.com] => {"changed": true, ...output omitted...

TASK [httpd enabled and running] ****
changed: [servera.lab.example.com] => {"changed": true, ...output omitted...

PLAY [Test intranet web server] ****

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

TASK [connect to intranet web server] ****
ok: [localhost] => {"accept_ranges": "bytes", "changed": false, "connection": "close", "content"①: "Welcome to the example.com intranet!\n", "content_length": "37", "content_type": "text/html; charset=UTF-8", "cookies": {}, "cookies_string": "", "date": "...output omitted...", "etag": "\"25-5790ddbcc5a48\"", "last_modified": "...output omitted...", "msg": "OK (37 bytes)", "redirected": false, "server": "Apache/2.4.6 (Red Hat Enterprise Linux)", "status"②: 200, "url": "http://servera.lab.example.com"}

PLAY RECAP ****
localhost : ok=2 changed=0 unreachable=0 failed=0
servera.lab.example.com : ok=6 changed=4 unreachable=0 failed=0

```

- ①** The server responded with the desired content, **Welcome to the example.com intranet!**\n.
- ②** The server responded with an HTTP status code of **200**.

## Cleanup

On workstation, run the **lab playbook-multi cleanup** script to clean up the resources created in this exercise.

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

This concludes the guided exercise.

## ► LAB

# IMPLEMENTING PLAYBOOKS

### PERFORMANCE CHECKLIST

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

### OUTCOMES

You should be able to construct and execute a playbook to install, configure, and verify the status of web and database services on a managed host.

Log in to **workstation** as student using **student** as the password, and run **lab playbook-review setup**. This setup script ensures that the managed host, **serverb.lab.example.com**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory file are installed on the control node.

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

A developer responsible for the company's internet website has asked you to write an Ansible Playbook to automate the setup of his server environment on **serverb.lab.example.com**.

A working directory, **/home/student/playbook-review**, has been created on **workstation** 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.

In this directory, create a playbook named **internet.yml**, which will contain two plays. The first play will require privilege escalation and must perform the following tasks in the specified order:

1. Use the `yum` module to ensure the latest versions of the following packages are installed: `firewalld`, `httpd`, `php`, `php-mysql`, and `mariadb-server`.
2. Ensure that the `firewalld` service is enabled and started.
3. Ensure that the `firewalld` service is configured to allow connections to the ports used by the `httpd` service.
4. Ensure that the `httpd` service is enabled and started.
5. Ensure that the `mariadb` service is enabled and started.
6. Use the `get_url` module to ensure that the content at the URL `http://materials.example.com/labs/playbook-review/index.php` has been installed as the file `/var/www/html/index.php` on the managed host.

The second play does not require privilege escalation and should run a single task using the `uri` module to confirm that the URL `http://serverb.lab.example.com/` returns an HTTP status code of 200.

According to recommended practices, plays and tasks should have names that document their purpose, but this is not required. The example solution names plays and tasks.

Do not forget that you can use the **ansible-doc** command to get help with finding and using the modules for your tasks.

After the playbook is written, verify its syntax and then execute the playbook to implement the configuration. Verify your work by executing **lab playbook-review grade**.



### NOTE

The playbook used by this lab is very similar to the one you wrote in the preceding guided exercise in this chapter. If you do not want to create this lab's playbook from scratch, you can use that exercise's playbook as a starting point for this lab.

If you do, be careful to target the correct hosts and change the tasks to match the instructions for this exercise.

1. Change to the working directory, **/home/student/playbook-review**.
2. Create a new playbook, **/home/student/playbook-review/internet.yml**, and add the necessary entries to start a first play named **Enable internet services** and specify its intended managed host, **serverb.lab.example.com**. Add an entry to enable privilege escalation.
3. Add the necessary entries to the **/home/student/playbook-review/internet.yml** file to define the tasks in the first play for configuring the managed host.
4. In **/home/student/playbook-review/internet.yml**, define another play for the task to be performed on the control node to test access to the web server that should be running on the **serverb** managed host. This play does not require privilege escalation.
5. Verify the syntax of the **internet.yml** playbook by using the **ansible-playbook** command.
6. Use **ansible-playbook** to run the playbook. Read through the output generated to ensure that all tasks completed successfully.

## Evaluation

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

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

## Cleanup

On workstation, run the **lab playbook-review cleanup** script to clean up the resources created in this lab.

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

This concludes the lab.

## ► SOLUTION

# IMPLEMENTING PLAYBOOKS

### PERFORMANCE CHECKLIST

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

### OUTCOMES

You should be able to construct and execute a playbook to install, configure, and verify the status of web and database services on a managed host.

Log in to **workstation** as student using **student** as the password, and run **lab playbook-review setup**. This setup script ensures that the managed host, **serverb.lab.example.com**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory file are installed on the control node.

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

A developer responsible for the company's internet website has asked you to write an Ansible Playbook to automate the setup of his server environment on **serverb.lab.example.com**.

A working directory, **/home/student/playbook-review**, has been created on **workstation** 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.

In this directory, create a playbook named **internet.yml**, which will contain two plays. The first play will require privilege escalation and must perform the following tasks in the specified order:

1. Use the `yum` module to ensure the latest versions of the following packages are installed: `firewalld`, `httpd`, `php`, `php-mysql`, and `mariadb-server`.
2. Ensure that the `firewalld` service is enabled and started.
3. Ensure that the `firewalld` service is configured to allow connections to the ports used by the `httpd` service.
4. Ensure that the `httpd` service is enabled and started.
5. Ensure that the `mariadb` service is enabled and started.
6. Use the `get_url` module to ensure that the content at the URL `http://materials.example.com/labs/playbook-review/index.php` has been installed as the file `/var/www/html/index.php` on the managed host.

The second play does not require privilege escalation and should run a single task using the `uri` module to confirm that the URL `http://serverb.lab.example.com/` returns an HTTP status code of 200.

According to recommended practices, plays and tasks should have names that document their purpose, but this is not required. The example solution names plays and tasks.

Do not forget that you can use the **ansible-doc** command to get help with finding and using the modules for your tasks.

After the playbook is written, verify its syntax and then execute the playbook to implement the configuration. Verify your work by executing **lab playbook-review grade**.



### NOTE

The playbook used by this lab is very similar to the one you wrote in the preceding guided exercise in this chapter. If you do not want to create this lab's playbook from scratch, you can use that exercise's playbook as a starting point for this lab.

If you do, be careful to target the correct hosts and change the tasks to match the instructions for this exercise.

1. Change to the working directory, **/home/student/playbook-review**.

```
[student@workstation ~] cd /home/student/playbook-review
```

2. Create a new playbook, **/home/student/playbook-review/internet.yml**, and add the necessary entries to start a first play named **Enable internet services** and specify its intended managed host, **serverb.lab.example.com**. Add an entry to enable privilege escalation.
  - 2.1. Add the following entry to the beginning of **/home/student/playbook-review/internet.yml** to begin the YAML format.

```

```

- 2.2. Add the following entry to denote the start of a play with a name of **Enable internet services**.

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

- 2.3. Add the following entry to indicate that the play applies to the **serverb** managed host. Be sure to indent the entry with two spaces.

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

- 2.4. Add the following entry to enable privilege escalation. Be sure to indent the entry with two spaces.

```
become: yes
```

3. Add the necessary entries to the **/home/student/playbook-review/internet.yml** file to define the tasks in the first play for configuring the managed host.

- 3.1. Add the following entry to define the beginning of the **tasks** list. Be sure to indent the entry with two spaces.

```
tasks:
```

- 3.2. Add the following entry to create a new task that ensures that the latest versions of the necessary packages are installed.

```
- name: latest version of all required packages installed
 yum:
 name:
 - firewalld
 - httpd
 - mariadb-server
 - php
 - php-mysql
 state: latest
```

- 3.3. Add the necessary entries to the **/home/student/playbook-review/internet.yml** file to define the firewall configuration tasks.

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

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

- 3.4. Add the necessary entries to define the service management tasks.

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

- name: mariadb enabled and running
 service:
 name: mariadb
 enabled: true
 state: started
```

- 3.5. Add the necessary entries to define the final task for generating web content for testing.

```
- name: test php page is installed
 get_url:
 url: "http://materials.example.com/labs/playbook-review/index.php"
 dest: /var/www/html/index.php
 mode: 0644
```

4. In **/home/student/playbook-review/internet.yml**, define another play for the task to be performed on the control node to test access to the web server that should be running on the **serverb** managed host. This play does not require privilege escalation.

- 4.1. Add the following entry to denote the start of a second play with a name of **Test internet web server**.

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

- 4.2. Add the following entry to indicate that the play applies to the **localhost** managed host. Be sure to indent the entry with two spaces.

```
hosts: localhost
```

- 4.3. Add the following line after the **hosts** keyword to disable privilege escalation for the second play. Be sure to indent the entry with two spaces.

```
become: no
```

- 4.4. Add an entry to the **/home/student/playbook-review/internet.yml** file to define the beginning of the **tasks** list. Be sure to indent the entry with two spaces.

```
tasks:
```

- 4.5. Add the following lines to create the task for verifying the managed host's web services from the control node.

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

5. Verify the syntax of the **internet.yml** playbook by using the **ansible-playbook** command.

```
[student@workstation playbook-review]$ ansible-playbook --syntax-check \
> internet.yml

playbook: internet.yml
```

6. Use **ansible-playbook** to run the playbook. Read through the output generated to ensure that all tasks completed successfully.

```
[student@workstation playbook-review]$ ansible-playbook 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 installed] ****
changed: [serverb.lab.example.com]

PLAY [Test internet web server] ****

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

TASK [connect to internet web server] ****
ok: [localhost]

PLAY RECAP ****
localhost : ok=2 changed=0 unreachable=0 failed=0
serverb.lab.example.com : ok=7 changed=5 unreachable=0 failed=0
```

## Evaluation

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

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

## Cleanup

On workstation, run the **lab playbook-review cleanup** script to clean up the resources created in this lab.

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

This concludes the lab.

# SUMMARY

---

In this chapter, you learned:

- A *play* is an ordered list of tasks, which runs 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.
- YAML files are structured using space indentation to represent the data hierarchy.
- Tasks are implemented using standardized code packaged as Ansible *modules*.
- The **ansible-doc** command can list installed modules, and provide documentation and example code snippets of how to use them in playbooks.
- The **ansible-playbook** command is used to verify playbook syntax and run playbooks.



## CHAPTER 4

# MANAGING VARIABLES AND FACTS

### GOAL

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

After completing this section, students should be able to create and reference variables in playbooks, affecting particular hosts or host groups, the play, or the global environment, and to describe how variable precedence works.

## INTRODUCTION TO ANSIBLE VARIABLES

Ansible supports variables that can be used to store values that can 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<br>file1                   |
| remoteserver\$1        | remote_server_1<br>remote_server1 |

## DEFINING VARIABLES

Variables can be defined in a variety of places in an Ansible project. However, this can be simplified to three basic scope levels:

- *Global scope*: Variables set from the command line or Ansible configuration.
- *Play scope*: Variables set in the play and related structures.
- *Host scope*: Variables set on host groups and individual hosts by the inventory, fact gathering, or registered tasks.

If the same variable name is defined at more than one level, the level with the highest precedence wins. A narrow scope takes precedence over a wider scope: variables defined by the inventory are overridden by variables defined by the playbook, which are overridden by variables defined on the command line.

A detailed discussion of variable precedence is available in the Ansible documentation, a link to which is provided in the References at the end of this section.

## 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 playbooks, you can define your own variables and then invoke those values in a task. For example, a variable named `web_package` can be defined with a value of `httpd`. A task can then call the variable using the `yum` module to install the `httpd` package.

Playbook variables can be defined in multiple ways. One common method is to place a variable in a `vars` block at the beginning of a playbook:

```
- 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 the playbook, the `vars_files` directive may be used, 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 that file or those files in YAML format:

```
user: joe
home: /home/joe
```

### Using Variables in Playbooks

After variables have been declared, administrators can use the variables in tasks. Variables are referenced by placing the variable name in double curly 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:

```
yum:
 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*, which apply to a specific host; and *group variables*, which 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. This is an older approach and not preferred, but you may still encounter it.

- 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 of the 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 uses an obsolete syntax.

## Using `group_vars` and `host_vars` Directories

The preferred approach to defining variables for hosts and host groups is to create two directories, `group_vars` and `host_vars`, in the same working directory as the inventory file or directory. 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 there are two data centers to manage and the data center hosts are defined in the `~/project/inventory` inventory file:

```
[admin@station project]$ cat ~/project/inventory
[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 the value to define varies 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 the value to be defined varies for each host in every data center, then define variable 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 of 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
```

## OVERRIDING VARIABLES FROM THE COMMAND LINE

Inventory variables are overridden by variables set in a playbook, but both kinds of variables may be overridden through arguments passed to the `ansible` or `ansible-playbook` commands 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-playbook main.yml -e "package=apache"
```

## VARIABLES AND ARRAYS

Instead of assigning configuration data that relates to the same element (a list of packages, a list of services, a list of users, and so on), to multiple variables, administrators can use *arrays*. One consequence of this is that an array can be browsed.

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
user3_home_dir: /users/acook
```

This could be rewritten as an array called **users**:

```
users:
 bjones:
 first_name: Bob
 last_name: Jones
 home_dir: /users/bjones
 acook:
 first_name: Anne
 last_name: Cook
 home_dir: /users/acook
```

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.

## REGISTERED VARIABLES

Administrators can use `register` statement to capture the output of a command. The output is saved into a variable that can be used later for either debugging purposes or to achieve something else, such as a particular configuration based on a command's output.

The following playbook 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
 yum:
 name: httpd
 state: installed
 register: install_result

 - debug: var=install_result
```

When you run the playbook, the `debug` module is used to dump the value of the `install_result` registered variable to the terminal.

```
[user@demo ~]$ ansible-playbook playbook.yml
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-2.4.6-40.el7.x86_64 providing httpd is already installed"
]
 }
}

PLAY RECAP ****
```

```
demo.example.com : ok=3 changed=0 unreachable=0 failed=0
```



## REFERENCES

### **Inventory – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/user\\_guide/intro\\_inventory.html](https://docs.ansible.com/ansible/2.7/user_guide/intro_inventory.html)

### **Variables – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/user\\_guide/playbooks\\_variables.html](https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html)

### **Variable Precedence: Where Should I Put A Variable?**

[https://docs.ansible.com/ansible/2.7/user\\_guide/playbooks\\_variables.html#variable-precedence-where-should-i-put-a-variable](https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable)

### **YAML Syntax – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/reference\\_appendices/YAMLSyntax.html](https://docs.ansible.com/ansible/2.7/reference_appendices/YAMLSyntax.html)

## ► GUIDED EXERCISE

# MANAGING VARIABLES

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

## OUTCOMES

You should be able to:

- Define variables in a playbook.
- Create tasks that use defined variables.

On `workstation`, run the lab setup script to confirm that the environment is ready for the exercise to begin. This script creates the **data-variables** working directory, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab data-variables setup
```

- 1. On `workstation`, as the `student` user, change into the `~/data-variables` directory.

```
[student@workstation ~]$ cd ~/data-variables
[student@workstation data-variables]$
```

- 2. Over the next several steps, you will create a playbook that installs the Apache web server and opens the ports for the service to be reachable. The playbook queries the web server to ensure it is up and running.

Create the `playbook.yml` playbook and define the following variables in the `vars` section:

### Variables

| VARIABLE                      | DESCRIPTION                                                          |
|-------------------------------|----------------------------------------------------------------------|
| <code>web_pkg</code>          | <b>Name of the package to install for the web server.</b>            |
| <code>firewall_pkg</code>     | <b>Name of the firewall package.</b>                                 |
| <code>web_service</code>      | <b>Name of the web service to manage.</b>                            |
| <code>firewall_service</code> | <b>Name of the firewall service to manage.</b>                       |
| <code>python_pkg</code>       | <b>Name of the required package for the <code>uri</code> module.</b> |
| <code>rule</code>             | <b>Name of service to open.</b>                                      |

```
- name: Deploy and start Apache HTTPD service
hosts: webserver
vars:
```

```
web_pkg: httpd
firewall_pkg: firewalld
web_service: httpd
firewall_service: firewalld
python_pkg: python-httpplib2
rule: http
```

- 3. Create the **tasks** block and create the first task, which should use the `yum` module to make sure the latest versions of the required packages are installed.

```
tasks:
 - name: Required packages are installed and up to date
 yum:
 name:
 - "{{ web_pkg }}"
 - "{{ firewall_pkg }}"
 - "{{ python_pkg }}"
 state: latest
```



#### NOTE

You can use `ansible-doc yum` to review the syntax for the `yum` module. The syntax shows that its `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 each package is up-to-date.

- 4. Create two tasks to make sure that the `httpd` and `firewalld` services are started and enabled.

```
- name: The {{ firewall_service }} service is started and enabled
 service:
 name: "{{ firewall_service }}"
 enabled: true
 state: started

- name: The {{ web_service }} service is started and enabled
 service:
 name: "{{ web_service }}"
 enabled: true
 state: started
```



#### NOTE

The `service` module works differently from the `yum` module, as documented by `ansible-doc service`. Its `name` directive takes the name of exactly one service to work with.

You can write a single task that ensures both of these 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
copy:
 content: "Example web content"
 dest: /var/www/html/index.html
```

- 6. Add a task that uses the `firewalld` module to ensure the firewall ports are open for the `firewalld` service named in the `rule` variable.

```
- name: The firewall port for {{ rule }} is open
firewalld:
 service: "{{ rule }}"
 permanent: true
 immediate: true
 state: enabled
```

- 7. Create a new play that queries the web service to ensure everything has been correctly configured. It should run on `localhost`. Because of that fact, Ansible does not have to change identity, so set the `become` module to **false**. You can use the `uri` module to check a URL. For this task, check for a status code of 200 to confirm the web server on `servera.lab.example.com` is running and correctly configured.

```
- name: Verify the Apache service
hosts: localhost
become: false
tasks:
 - name: Ensure the webserver is reachable
 uri:
 url: http://servera.lab.example.com
 status_code: 200
```

- 8. When completed, the playbook should appear as follows. 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: python-httpplib2
 rule: http

tasks:
 - name: Required packages are installed and up to date
 yum:
 name:
 - "{{ web_pkg }}"
 - "{{ firewall_pkg }}"
 - "{{ python_pkg }}"
 state: latest
```

```

- name: The {{ firewall_service }} service is started and enabled
 service:
 name: "{{ firewall_service }}"
 enabled: true
 state: started

- name: The {{ web_service }} service is started and enabled
 service:
 name: "{{ web_service }}"
 enabled: true
 state: started

- name: Web content is in place
 copy:
 content: "Example web content"
 dest: /var/www/html/index.html

- name: The firewall port for {{ rule }} is open
 firewalld:
 service: "{{ rule }}"
 permanent: true
 immediate: true
 state: enabled

- name: Verify the Apache service
 hosts: localhost
 become: false
 tasks:
 - name: Ensure the webserver is reachable
 uri:
 url: http://servera.lab.example.com
 status_code: 200

```

- 9. Before you run the playbook, use the **ansible-playbook --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-playbook --syntax-check playbook.yml
playbook: playbook.yml
```

- 10. Use the **ansible-playbook** 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-playbook 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: [localhost]

TASK [Ensure the webserver is reachable] ****
ok: [localhost]

PLAY RECAP ****
localhost : ok=2 changed=0 unreachable=0 failed=0
servera.lab.example.com : ok=6 changed=4 unreachable=0 failed=0
```

## Cleanup

On workstation, run the **lab data-variables cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab data-variables cleanup
```

This concludes the guided exercise.

# MANAGING SECRETS

## OBJECTIVES

After completing this section, students should be able to encrypt sensitive variables using Ansible Vault, and run playbooks that reference Vault-encrypted variable files.

## ANSIBLE VAULT

Ansible may need access to sensitive data such as passwords or API keys in order to configure managed hosts. Normally, this information might be 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 which 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 structured data file used by Ansible. To use Ansible Vault, a command-line tool named **ansible-vault** is used to create, edit, encrypt, decrypt, and view files. Ansible Vault can encrypt any structured 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. The command prompts for the new vault password and then opens a file using the default editor, **vi**. You can set and export the **EDITOR** environment variable to specify a different default editor by setting and exporting. For example, to set the default editor to **nano**, **export EDITOR=nano**.

```
[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 older versions may 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
less 458 (POSIX regular expressions)
Copyright (C) 1984-2012 Mark Nudelman

less comes with NO WARRANTY, to the extent permitted by law.
For information about the terms of redistribution,
see the file named README in the less distribution.
Homepage: http://www.greenwoodsoftware.com/less
my_secret: "yJJvPqhsiusmmPPZdnjndkdnYNDjdz782meUZcw"
```

## 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. At most one input file may be used 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 once. 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-playbook** command. If you do not provide the password, the playbook returns an error:

```
[student@demo ~]$ ansible-playbook site.yml
ERROR: A vault password must be specified to decrypt vars/api_key.yml
```

To provide the vault password to the playbook, use the **--vault-id** option. For example, to provide the vault password interactively, use **--vault-id @prompt** as illustrated in the following example:

```
[student@demo ~]$ ansible-playbook --vault-id @prompt site.yml
Vault password (default): redhat
```



### IMPORTANT

If you are using a release of Ansible earlier than version 2.4, you need to use the **--ask-vault-pass** option to interactively provide the vault password. You can still use this option if all vault-encrypted files used by the playbook were encrypted with the same password.

```
[student@demo ~]$ ansible-playbook --ask-vault-pass site.yml
Vault password: redhat
```

Alternatively, you can use the **--vault-password-file** option to specify a file that stores the encryption password in plain text. The password should 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-playbook --vault-password-file=vault-pw-file site.yml
```

You can also use the `ANSIBLE_VAULT_PASSWORD_FILE` environment variable to specify the default location of the password file.



### IMPORTANT

Starting with Ansible 2.4, you can use multiple Ansible Vault passwords with `ansible-playbook`. To use multiple passwords, pass multiple `--vault-id` or `--vault-password-file` options to the `ansible-playbook` command.

```
[student@demo ~]$ ansible-playbook \
> --vault-id one@prompt --vault-id two@prompt site.yml
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 `ansible-vault` command, however, when you run `ansible-playbook` then the password for the matching ID is tried before any others. If it does not match, the other passwords you provided will be tried next. 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 matching host groups to which they apply. The `host_vars` directory normally contains variable files with names matching host names of managed hosts to which they apply.

However, instead of using files in `group_vars` or `host_vars`, you also can use directories for each host group or managed host. Those directories can then contain multiple variable files, all of which are used by the host group or managed host. For example, in the following project directory for `playbook.yml`, members of the `webservers` host group uses variables in the `group_vars/webservers/vars` file, and `demo.example.com` uses the variables in both `host_vars/demo.example.com/vars` and `host_vars/demo.example.com/vault`:

```
.
├── ansible.cfg
├── group_vars
│ └── webservers
│ └── vars
└── host_vars
 └── demo.example.com
 ├── vars
 └── vault
└── inventory
└── playbook.yml
```

In this scenario, the advantage is that most variables for `demo.example.com` can be placed in the `vars` file, but sensitive variables can be kept secret by placing them separately in the `vault` file. Then the administrator can use `ansible-vault` to encrypt the `vault` file, while leaving the `vars` file as plain text.

There is nothing special about the file names being used in this example inside the `host_vars/demo.example.com` directory. That directory could contain more files, some encrypted by Ansible Vault and some which are not.

Playbook variables (as opposed to inventory variables) can also be protected with Ansible Vault. Sensitive playbook variables can be placed in a separate file which is encrypted with Ansible Vault and which is included in the playbook through a `vars_files` directive. This can be useful, because playbook variables take precedence over inventory variables.

## Speeding up Vault Operations

By default, Ansible uses functions from the `python-crypto` package to encrypt and decrypt vault files. If there are many encrypted files, decrypting them at startup may cause a perceptible delay. To speed this up, install the `python-cryptography` package:

```
[student@demo ~]$ sudo yum install python-cryptography
```

The `python-cryptography` package provides a Python library which exposes cryptographic recipes and primitives. The default Ansible installation uses PyCrypto for these cryptographic operations.

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 all the vault passwords you provided until it finds the right one.

Figure 4.0: Executing with Ansible Vault



### REFERENCES

`ansible-playbook(1)` and `ansible-vault(1)` man pages

#### **Vault – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/user\\_guide/playbooks\\_vault.html](https://docs.ansible.com/ansible/2.7/user_guide/playbooks_vault.html)

#### **Variables and Vaults – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/user\\_guide/playbooks\\_best\\_practices.html#best-practices-for-variables-and-vaults](https://docs.ansible.com/ansible/2.7/user_guide/playbooks_best_practices.html#best-practices-for-variables-and-vaults)

## ► GUIDED EXERCISE

# MANAGING SECRETS

In this exercise, you will encrypt sensitive variables with Ansible Vault to protect them, and then run a playbook that uses those variables.

## OUTCOMES

You should be able to:

- Use variables defined in an encrypted file to execute a playbook.

Log in to `workstation` as `student` using `student` as the password.

On `workstation`, run the `lab data-secret setup` script. This script ensures that Ansible is installed on `workstation` and creates a working directory for this exercise. This directory includes an inventory file that points to `servera.lab.example.com` as a managed host, which is part of the `devservers` group.

```
[student@workstation ~]$ lab data-secret setup
```

- 1. On `workstation`, as the `student` user, change to the `~/data-secret` directory.

```
[student@workstation ~]$ cd ~/data-secret
```

- 2. Edit the contents of the provided encrypted file, `secret.yml`. The file can be decrypted using `redhat` as the password. Uncomment the `username` and `pwhash` variable entries.

- 2.1. Edit the encrypted `secret.yml` file in `~/data-secret`. Provide a password of `redhat` for the vault 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. They should appear as follows:

```
username: ansibleuser1
pw: 6jf...uxhP1
```

Save the file.

- 3. Create a playbook that uses the variables defined in the `secret.yml` encrypted file. Name the playbook `create_users.yml` and create it under the `~/data-secret` directory.

Configure the playbook to use the `devservers` host group, which was defined by the lab setup script in the inventory file. Run this playbook as the `devops` user on the remote managed host. Configure the playbook to create the `ansibleuser1` user defined by the

username variable. Set the user's password using the password hash stored in the pwhash variable.

The `create_users.yml` file should appear as follows:

```

- 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
 user:
 name: "{{ username }}"
 password: "{{ pwhash }}"
```

- ▶ 4. Use the `ansible-playbook --syntax-check` command to verify the syntax of the `create_users.yml` playbook. Use the `--ask-vault-pass` option to prompt for the vault password, which guards `secret.yml`. Resolve any syntax errors before you continue.

```
[student@workstation data-secret]$ ansible-playbook --syntax-check \
> --ask-vault-pass create_users.yml
Vault password (default): redhat

playbook: create_users.yml
```



#### NOTE

Instead of using `--ask-vault-pass`, you can use the newer `--vault-id @prompt` option to do the same thing.

- ▶ 5. Create a password file to use for the playbook execution instead of asking for a password. The file should be called `vault-pass` and it should store the `redhat` vault password as a plain text. 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. Execute the Ansible Playbook, using the vault password file to create the `ansibleuser1` user on a remote system and using the passwords stored as variables in the `secret.yml` Ansible Vault encrypted file. Use the `vault-pass` vault password file.

```
[student@workstation data-secret]$ ansible-playbook \
> --vault-password-file=vault-pass create_users.yml

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
```

- 7. Verify that the playbook ran correctly. The user `ansibleuser1` should exist and have the correct password on `servera.lab.example.com`. Test this by using `ssh` to log in as that user on `servera.lab.example.com`. The password for `ansibleuser1` is `redhat`. To make sure that `ssh` only tries to authenticate by password and not by an SSH key, use the `-o PreferredAuthentications=password` option when you log in.

Log off from `servera` when you are finished.

```
[student@workstation data-secret]$ ssh -o PreferredAuthentications=password \
> ansibleuser1@servera.lab.example.com
ansibleuser1@servera.lab.example.com's password: redhat
Warning: Permanently added 'servera.lab.example.com,172.25.250.10' (ECDSA) to the
list of known hosts.
[ansibleuser1@servera ~]$ exit
```

## Cleanup

On workstation, run the `lab data-secret cleanup` script to clean up this exercise.

```
[student@workstation ~]$ lab data-secret cleanup
```

This concludes the guided exercise.

# MANAGING FACTS

---

## OBJECTIVES

After completing this section, students should be able to reference data about managed hosts using Ansible facts, and configure custom facts on managed hosts.

## 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 of the facts gathered for a managed host might include:

- The host name.
- The kernel version.
- The network interfaces.
- The IP addresses.
- The version of the operating system.
- Various environment variables.
- The number of CPUs.
- The available or free memory.
- The available disk space.

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:

- A server can be restarted by a conditional task which is run based on a fact containing the managed host's current kernel version.
- The MySQL configuration file can be customized depending on the available memory reported by a fact.
- The IPv4 address used in a configuration file can be set based on the value of a fact.

Normally, every play runs the `setup` module automatically before the first task in order to gather facts. This is reported as the `Gathering Facts` task in Ansible 2.3 and later, or simply as `setup` in older versions of Ansible. By default, you do not need to have a task to run `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 `debug` module to print the value of the `ansible_facts` variable.

```
- name: Fact dump
hosts: all
tasks:
 - name: Print all facts
```

```
debug:
 var: ansible_facts
```

When you run the playbook, the facts are displayed in the job output:

```
[user@demo ~]$ ansible-playbook 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": [
 "172.25.250.10"
],
 "all_ipv6_addresses": [
 "fe80::5054:ff:fe00:fa0a"
],
 "ansible_local": {},
 "apparmor": {
 "status": "disabled"
 },
 "architecture": "x86_64",
 "bios_date": "01/01/2011",
 "bios_version": "0.5.1",
 "cmdline": {
 "BOOT_IMAGE": "/boot/vmlinuz-3.10.0-327.el7.x86_64",
 "LANG": "en_US.UTF-8",
 "console": "ttyS0,115200n8",
 "crashkernel": "auto",
 "net.ifnames": "0",
 "no_timer_check": true,
 "ro": true,
 "root": "UUID=2460ab6e-e869-4011-acae-31b2e8c05a3b"
 },
 ...output omitted...
 }
}
```

The playbook displays the content of the `ansible_facts` variable in JSON format as a hash/dictionary of variables. You can browse the output to see what facts are gathered, to find facts that you might want to use in your plays.

The following table shows some facts which might be gathered from a managed node and may be useful in a playbook:

### Examples of Ansible Facts

| FACT                        | VARIABLE                               |
|-----------------------------|----------------------------------------|
| Short host name             | <code>ansible_facts['hostname']</code> |
| Fully qualified domain name | <code>ansible_facts['fqdn']</code>     |

| FACT                                              | VARIABLE                                                      |
|---------------------------------------------------|---------------------------------------------------------------|
| Main IPv4 address (based on routing)              | ansible_facts['default_ipv4']['address']                      |
| List of the names of all network interfaces       | ansible_facts['interfaces']                                   |
| Size of the <code>/dev/vda1</code> disk partition | ansible_facts['devices']['vda']['partitions']['vda1']['size'] |
| List of DNS servers                               | ansible_facts['dns']['nameservers']                           |
| Version of the currently running kernel           | ansible_facts['kernel']                                       |



### NOTE

Remember that when a variable's value is a hash/dictionary, there are two syntaxes that 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
 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. Moreover, facts can also be used to create dynamic groups of hosts that match particular criteria.

```
[user@demo ~]$ ansible-playbook playbook.yml
PLAY ****
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"
}
```

```
PLAY RECAP ****
demo1.example.com : ok=2 changed=0 unreachable=0 failed=0
```

## ANSIBLE FACTS INJECTED AS VARIABLES

Before Ansible 2.5, facts were 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 would have been called `ansible_distribution`.

Many older playbooks still use facts injected as variables instead of the new syntax that is namespaced under the `ansible_facts` variable. You can use an ad hoc command to run the `setup` module to print the value of all facts in this form. In the following example, an ad hoc command is used to run the `setup` module on the managed host `demo1.example.com`:

```
[user@demo ~]$ ansible demo1.example.com -m setup
demo1.example.com | SUCCESS => {
 "ansible_facts": {
 "ansible_all_ipv4_addresses": [
 "172.25.250.10"
],
 "ansible_all_ipv6_addresses": [
 "fe80::5054:ff:fe00:fa0a"
],
 "ansible_apparmor": {
 "status": "disabled"
 },
 "ansible_architecture": "x86_64",
 "ansible_bios_date": "01/01/2011",
 "ansible_bios_version": "0.5.1",
 "ansible_cmdline": {
 "BOOT_IMAGE": "/boot/vmlinuz-3.10.0-327.el7.x86_64",
 "LANG": "en_US.UTF-8",
 "console": "ttyS0,115200n8",
 "crashkernel": "auto",
 "net.ifnames": "0",
 "no_timer_check": true,
 "ro": true,
 "root": "UUID=2460ab6e-e869-4011-acae-31b2e8c05a3b"
 }
 ...
 ...output omitted...
}
```

The following table compares the old and new fact names.

**Comparison of Selected Ansible Fact Names**

| ANSIBLE_FACTS FORM                                    | OLD FACT VARIABLE FORM                       |
|-------------------------------------------------------|----------------------------------------------|
| <code>ansible_facts['hostname']</code>                | <code>ansible_hostname</code>                |
| <code>ansible_facts['fqdn']</code>                    | <code>ansible_fqdn</code>                    |
| <code>ansible_facts['default_ipv4']['address']</code> | <code>ansible_default_ipv4['address']</code> |

| ANSIBLE_FACTS FORM                                            | OLD FACT VARIABLE FORM                               |
|---------------------------------------------------------------|------------------------------------------------------|
| ansible['interfaces']                                         | ansible_interfaces                                   |
| ansible_facts['devices'][‘vda’][‘partitions’][‘vda1’][‘size’] | ansible_devices[‘vda’][‘partitions’][‘vda1’][‘size’] |
| ansible_facts[‘dns’][‘nameservers’]                           | ansible_dns[‘nameservers’]                           |
| ansible_facts[‘kernel’]                                       | ansible_kernel                                       |



### IMPORTANT

Currently, Ansible recognizes both the new fact naming system (using `ansible_facts`) and the old pre-2.5 "facts injected as separate variables" naming system.

You can turn off the old naming system by setting the `inject_facts_as_vars` parameter in the **[default]** section of the Ansible configuration file to `false`. The default setting is currently `true`.

The default value of `inject_facts_as_vars` will probably change to `false` in a future version of Ansible. 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 old namespace will result in the following error:

```
...output omitted...
TASK [Show me the facts] ****
fatal: [demo.example.com]: FAILED! => {"msg": "The task includes an option
with an undefined variable. The error was: 'ansible_distribution' is
undefined\n\nThe error appears to have been in
'/home/student/demo/playbook.yml': line 5, column 7, but may\nbe elsewhere in
the file depending on the exact syntax problem.\n\nThe offending line appears
to be:\n\n tasks:\n - name: Show me the facts\n ^ here\n"}
...output omitted...
```

## TURNING OFF FACT GATHERING

Sometimes, you do not want to gather facts for your play. There are a couple of reasons why this might be the case. It might be that you are not using any facts and want to speed up the play or reduce load caused by the play on the managed hosts. It might be that the managed hosts cannot run the `setup` module for some reason, or 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 gathers no facts automatically
 hosts: large_farm
 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 `setup` module:

```
tasks:
 - name: Manually gather facts
 setup:
...output omitted...
```

## CUSTOM FACTS

Administrators can create *custom facts* which are stored locally on each managed host. These facts are integrated into the list of standard facts gathered by the `setup` module when it runs on the managed host. These allow the managed host to provide arbitrary variables to Ansible which can be used to adjust the behavior of plays.

Custom facts can be defined in a static file, formatted as an INI file or using JSON. They can also be executable scripts which generate JSON output, just like a dynamic inventory script.

Custom facts allow administrators to define certain values for managed hosts, which plays might use to populate configuration files or conditionally run tasks. Dynamic custom facts allow the values for these facts, or even which facts are provided, to be determined programmatically when the play is run.

By default, the `setup` module loads custom facts from files and scripts in each managed host's `/etc/ansible/facts.d` directory. The name of each file or script must end in `.fact` in order to be used. Dynamic custom fact scripts must output JSON-formatted facts and must be executable.

This is an example of a static custom facts file 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
```

The same facts could be provided 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.

Custom facts are stored by the `setup` module 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 preceding custom facts are produced by a file saved as `/etc/ansible/facts.d/custom факт` on the managed host. In that case, the value of `ansible_facts.ansible_local['custom']` [`'users'`][`'user1'`] is **joe**.

You can check the structure of your custom facts by running the `setup` module on the managed hosts with an ad hoc command.

```
[user@demo ~]$ ansible demo1.example.com -m setup
demo1.example.com | SUCCESS => {
 "ansible_facts": {
 ...output omitted...
 "ansible_local": {
 "custom": {
 "packages": {
 "db_package": "mariadb-server",
 "web_package": "httpd"
 },
 "users": {
 "user1": "joe",
 "user2": "jane"
 }
 }
 },
 ...output omitted...
 },
 "changed": false
}
```

Custom facts can be used the same way as default facts in playbooks:

```
[user@demo ~]$ cat playbook.yml

- hosts: all
 tasks:
 - name: Prints various Ansible facts
 debug:
 msg: >
 The package to install on {{ ansible_fqdn }}
 is {{ ansible_facts.ansible_local.custom.packages.web_package }}

[user@demo ~]$ ansible-playbook playbook.yml
PLAY ****

ok: [demo1.example.com]

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
```

## MAGIC VARIABLES

Some variables are not facts or configured through the `setup` module, but are also automatically set by Ansible. These *magic variables* can also be useful to get information specific to a particular managed host.

Four of the most useful 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 the current managed host is in.

### `groups`

Lists all groups and hosts in the inventory.

### `inventory_hostname`

Contains the host name for the current managed host as configured in the inventory. This may be different from the host name reported by facts for various reasons.

There are a number of other "magic variables" as well. For more information, see [https://docs.ansible.com/ansible/2.7/user\\_guide/playbooks\\_variables.html#variable-precedence-where-should-i-put-a-variable](https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable). One way to get insight into their values is to use the `debug` module to report on the contents of the `hostvars` variable for a particular host:

```
[user@demo ~]$ ansible localhost -m debug -a 'var=hostvars["localhost"]'
localhost | SUCCESS => {
 "hostvars[\"localhost\"]": {
 "ansible_check_mode": false,
 "ansible_connection": "local",
 "ansible_diff_mode": false,
 "ansible_facts": {},
 "ansibleforks": 5,
 "ansible_inventory_sources": [
 "/home/student/demo/inventory"
],
 "ansible_playbook_python": "/usr/bin/python2",
 "ansible_python_interpreter": "/usr/bin/python2",
 "ansible_verbosity": 0,
 "ansible_version": {
 "full": "2.7.0",
 "major": 2,
 "minor": 7,
 "revision": 0,
 "string": "2.7.0"
 },
 "group_names": [],
 "groups": {
 "all": [
 "serverb.lab.example.com"
],
 "ungrouped": [],
 "webservers": [
 "serverb.lab.example.com"
]
 }
 }
}
```

```
},
"inventory_hostname": "localhost",
"inventory_hostname_short": "localhost",
"omit": "__omit_place_holder__18d132963728b2cbf7143dd49dc4bf5745fe5ec3",
"playbook_dir": "/home/student/demo"
}
}
```

Figure 4.0: Defining and using Ansible custom facts



## REFERENCES

### **setup - Gathers facts about remote hosts – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/modules/setup\\_module.html](https://docs.ansible.com/ansible/2.7/modules/setup_module.html)

### **Local Facts (Facts.d) – Variables – Ansible Documentation**

[https://docs.ansible.com/ansible/2.7/user\\_guide/playbooks\\_variables.html#local-facts-facts-d](https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html#local-facts-facts-d)

## ► GUIDED EXERCISE

# MANAGING FACTS

In this exercise, you will 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.

On `workstation`, run the lab setup script to confirm the environment is ready for the exercise to begin. This script creates the working directory, `data-facts`, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab data-facts setup
```

- 1. On `workstation`, as the `student` user, change into the `~/data-facts` directory.

```
[student@workstation ~]$ cd ~/data-facts
[student@workstation data-facts]$
```

- 2. The Ansible `setup` module retrieves facts from systems. Run an ad hoc command to retrieve the facts for all servers in the `webserver` group. The output displays all the facts gathered for `servera.lab.example.com` in JSON format. Review some of the variables displayed.

```
[student@workstation data-facts]$ ansible webserver -m setup
...output omitted...
servera.lab.example.com | SUCCESS => {
 "ansible_facts": {
 "ansible_all_ipv4_addresses": [
 "172.25.250.10"
],
 "ansible_all_ipv6_addresses": [
 "fe80::5054:ff:fe00:fa0a"
],
 ...
}
```

- 3. On `workstation`, create a fact file named `/home/student/data-facts/custom.fact`. The fact file defines the package to install and the service to start on `servera`. The file should read as follows:

```
[general]
package = httpd
service = httpd
```

```
state = started
```

- ▶ 4. Create the **setup\_facts.yml** playbook to make the **/etc/ansible/facts.d** remote directory and to save the **custom факт** file to that directory.

```

- name: Install remote facts
 hosts: webserver
 vars:
 remote_dir: /etc/ansible/facts.d
 facts_file: custom.fact
 tasks:
 - name: Create the remote directory
 file:
 state: directory
 recurse: yes
 path: "{{ remote_dir }}"
 - name: Install the new facts
 copy:
 src: "{{ facts_file }}"
 dest: "{{ remote_dir }}"
```

- ▶ 5. Run an ad hoc command with the **setup** module. Search for the **ansible\_local** section in the output. There should not be any custom facts at this point.

```
[student@workstation data-facts]$ ansible webserver -m setup
servera.lab.example.com | SUCCESS => {
 "ansible_facts": {
 ...output omitted...
 "ansible_local": {}
 ...output omitted...
 },
 "changed": false
}
```

- ▶ 6. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation data-facts]$ ansible-playbook --syntax-check setup_facts.yml
playbook: setup_facts.yml
```

- ▶ 7. Run the **setup\_facts.yml** playbook.

```
[student@workstation data-facts]$ ansible-playbook setup_facts.yml
PLAY [Install remote facts] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
```

```
TASK [Create the remote directory] *****
changed: [servera.lab.example.com]

TASK [Install the new facts] *****
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com : ok=3 changed=2 unreachable=0 failed=0
```

- ▶ 8. It is now possible to create the main playbook that uses both default and user facts to configure servera. Over the next several steps, you will add to the playbook file. Start the **playbook.yml** playbook file with the following:

```

- name: Install Apache and starts the service
 hosts: webserver
```

- ▶ 9. Continue editing the **playbook.yml** file by creating the first task that installs the *httpd* package. Use the user fact for the name of the package.

```
tasks:
- name: Install the required package
 yum:
 name: "{{ ansible_facts.ansible_local.custom.general.package }}"
 state: latest
```

- ▶ 10. Create another task that uses the custom fact to start the *httpd* service.

```
- name: Start the service
 service:
 name: "{{ ansible_facts.ansible_local.custom.general.service }}"
 state: "{{ ansible_facts.ansible_local.custom.general.state }}"
```

- ▶ 11. When completed with all the tasks, the full playbook should look like the following. Review the playbook and ensure all the tasks are defined.

```

- name: Install Apache and starts the service
 hosts: webserver

tasks:
- name: Install the required package
 yum:
 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 }}"
```

- 12. Before running the playbook, use an ad hoc command to verify the `httpd` service is not currently running on `servera`.

```
[student@workstation data-facts]$ ansible servera.lab.example.com -m command \
> -a 'systemctl status httpd'
servera.lab.example.com | FAILED | rc=4 >>
Unit httpd.service could not be found.non-zero return code
```

- 13. Verify the syntax of the playbook by running `ansible-playbook --syntax-check`. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation data-facts]$ ansible-playbook --syntax-check playbook.yml

playbook: playbook.yml
```

- 14. Run the playbook using the `ansible-playbook` command. Watch the output as Ansible installs the package and then enables the service.

```
[student@workstation data-facts]$ ansible-playbook 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
```

- 15. Use an ad hoc command to execute `systemctl` to determine if the `httpd` service is now running on `servera`.

```
[student@workstation data-facts]$ ansible servera.lab.example.com -m command \
> -a 'systemctl status httpd'
servera.lab.example.com | CHANGED | rc=0 >>
● httpd.service - The Apache HTTP Server
 Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset: disabled)
 Active: active (running) since Mon 2018-05-16 17:17:20 PDT; 12s ago
 Docs: man:httpd(8)
 man:apachectl(8)
 Main PID: 32658 (httpd)
 Status: "Total requests: 0; Current requests/sec: 0; Current traffic: 0 B/sec"
 CGroup: /system.slice/httpd.service
```

*...output omitted...*

## Cleanup

On workstation, run the **lab data-facts cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab data-facts cleanup
```

This concludes the guided exercise.

## ► LAB

# MANAGING VARIABLES AND FACTS

## PERFORMANCE CHECKLIST

In this lab, you will 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.

Log in as the student user on workstation and run **lab data-review setup**. The script creates the **data-review** project directory and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab data-review setup
```

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.

A working directory, `/home/student/data-review`, has been created on workstation for this project. The directory has already been populated with an `ansible.cfg` and an `inventory` file. The managed host, `serverb.lab.example.com`, is defined in this inventory as a member of the `webserver` host group.

The `files` subdirectory contains an `httpd.conf` configuration file, which configures the Apache web service for basic authentication. The subdirectory also contains a `.htaccess` file which can be used to control access to the web server's document root directory.

In the project directory, create a playbook named `playbook.yml`, which installs and configures the firewall and web service on `serverb.lab.example.com`. The playbook should also create an `index.html` file that identifies the host name and IP address of the managed host using Ansible facts. This content file should only be visible to web users who successfully authenticate. To maintain the security of the password used for basic authentication, define the user password as a variable in a file encrypted by Ansible Vault.

After the playbook is created, execute the playbook and then verify the results by retrieving the content file using basic authentication over HTTPS.

1. Create the `playbook.yml` playbook. Begin the playbook by identifying the `webserver` host group as the managed host. Define the following play variables:

### Variables

| VARIABLE                  | VALUES                 |
|---------------------------|------------------------|
| <code>firewall_pkg</code> | <code>firewalld</code> |
| <code>web_pkg</code>      | <code>httpd</code>     |

| VARIABLE       | VALUES                              |
|----------------|-------------------------------------|
| web_svc        | <b>httpd</b>                        |
| ssl_pkg        | <b>mod_ssl</b>                      |
| python_pkg     | <b>python-passlib</b>               |
| httpdconf_src  | <b>files/httpd.conf</b>             |
| httpdconf_file | <b>/etc/httpd/conf/httpd.conf</b>   |
| htaccess_src   | <b>files/.htaccess</b>              |
| secrets_dir    | <b>/etc/httpd/secrets</b>           |
| secrets_file   | <b>"{{ secrets_dir }}/htpasswd"</b> |
| web_root       | <b>/var/www/html</b>                |
| web_user       | <b>guest</b>                        |

2. Add a directive to the play that adds additional variables from a variable file named **vars/secret.yml**. This file will contain a variable that specifies the password for the web user. You will create this file later in the lab.
3. 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**, **ssl\_pkg**, and **python\_pkg** variables.
4. Add a second task to the playbook that ensures that the file specified by the **httpdconf\_src** variable has been copied (with the **copy** module) to the location specified by the **httpdconf\_file** variable on the managed host. The file should be owned by the **root** user and the **root** group. Also set **0644** as the file permissions.
5. Add a third task that uses the **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 should be owned by the **apache** user and the **apache** group. Also set **0500** as the file permissions.
6. Add a fourth task that uses the **htpasswd** module to create an **htpasswd** file to be used for basic authentication of web users. The path attribute should be defined by the **secrets\_file** variable. The file should contain an entry for the user defined by the **web\_user** variable. The user's password will be defined by the **web\_pass** variable to be added later. The file should be owned by the **apache** user and the **apache** group. Also set **0400** as the file permissions.
7. Add a fifth task that uses the **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 should be owned by the **apache** user and the **apache** group. Set **0400** as the file permissions.
8. Add a sixth task that uses the **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 to the **copy** module to specify the content of the file, and Ansible facts to specify the host name and IP address.

9. Add a seventh task that uses the `firewalld` module to open up on the firewall the `https` service needed for users to access web services on the managed host. This firewall change should be permanent and should take place immediately.
10. Add an eighth task that uses the `service` module to enable and restart 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. Add a final task that uses the `service` module to enable and restart the firewall service on the managed host so that all configuration changes take effect.
12. Create a file encrypted with Ansible Vault, named `vars/secret.yml`. It should set the `web_pass` variable to `redhat`, which will be the web user's password.
13. Run the `playbook.yml` playbook.
14. On `workstation`, use `curl` to ensure that the web server is reachable over HTTPS. Verify that basic authentication is working by authenticating as the `guest` user and using `redhat` as the password.

## Evaluation

Run the `lab data-review grade` command on `workstation` to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

## Cleanup

On `workstation`, run the `lab data-review cleanup` script to clean up this exercise.

```
[student@workstation ~]$ lab data-review cleanup
```

This concludes the lab.

## ► SOLUTION

# MANAGING VARIABLES AND FACTS

### PERFORMANCE CHECKLIST

In this lab, you will 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.

Log in as the student user on `workstation` and run `lab data-review setup`. The script creates the `data-review` project directory and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab data-review setup
```

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.

A working directory, `/home/student/data-review`, has been created on `workstation` for this project. The directory has already been populated with an `ansible.cfg` and an `inventory` file. The managed host, `serverb.lab.example.com`, is defined in this inventory as a member of the `webserver` host group.

The `files` subdirectory contains an `httpd.conf` configuration file, which configures the Apache web service for basic authentication. The subdirectory also contains a `.htaccess` file which can be used to control access to the web server's document root directory.

In the project directory, create a playbook named `playbook.yml`, which installs and configures the firewall and web service on `serverb.lab.example.com`. The playbook should also create an `index.html` file that identifies the host name and IP address of the managed host using Ansible facts. This content file should only be visible to web users who successfully authenticate. To maintain the security of the password used for basic authentication, define the user password as a variable in a file encrypted by Ansible Vault.

After the playbook is created, execute the playbook and then verify the results by retrieving the content file using basic authentication over HTTPS.

1. Create the `playbook.yml` playbook. Begin the playbook by identifying the `webserver` host group as the managed host. Define the following play variables:

#### Variables

| VARIABLE                  | VALUES                 |
|---------------------------|------------------------|
| <code>firewall_pkg</code> | <code>firewalld</code> |
| <code>web_pkg</code>      | <code>httpd</code>     |

| VARIABLE       | VALUES                              |
|----------------|-------------------------------------|
| web_svc        | <b>httpd</b>                        |
| ssl_pkg        | <b>mod_ssl</b>                      |
| python_pkg     | <b>python-passlib</b>               |
| httpdconf_src  | <b>files/httpd.conf</b>             |
| httpdconf_file | <b>/etc/httpd/conf/httpd.conf</b>   |
| htaccess_src   | <b>files/.htaccess</b>              |
| secrets_dir    | <b>/etc/httpd/secrets</b>           |
| secrets_file   | <b>"{{ secrets_dir }}/htpasswd"</b> |
| web_root       | <b>/var/www/html</b>                |
| web_user       | <b>guest</b>                        |

- 1.1. Change to the **data-review** project 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
 web_pkg: httpd
 web_svc: httpd
 ssl_pkg: mod_ssl
 python_pkg: python-passlib
 httpdconf_src: files/httpd.conf
 httpdconf_file: /etc/httpd/conf/httpd.conf
 htaccess_src: files/.htaccess
 secrets_dir: /etc/httpd/secrets
 secrets_file: "{{ secrets_dir }}/htpasswd"
 web_root: /var/www/html
 web_user: guest
```

2. Add a directive to the play that adds additional variables from a variable file named **vars/secret.yml**. This file will contain a variable that specifies the password for the web user. You will create this file later in the lab.

Using the **vars\_files** keyword, add the following lines to the playbook to instruct Ansible to use variables found in the **vars/secret.yml** variable file.

```
vars_files:
 - vars/secret.yml
```

3. 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`, `ssl_pkg`, and `python_pkg` variables.

- 3.1. Define the beginning of the **tasks** section by adding the following line to the playbook:

```
tasks:
```

- 3.2. Add the following lines to the playbook to define a task that uses the `yum` module to install the required packages.

```
- name: Latest version of necessary packages installed
 yum:
 name:
 - "{{ firewall_pkg }}"
 - "{{ web_pkg }}"
 - "{{ ssl_pkg }}"
 - "{{ python_pkg }}"
 state: latest
```

4. Add a second task to the playbook that ensures that the file specified by the `httpdconf_src` variable has been copied (with the `copy` module) to the location specified by the `httpdconf_file` variable on the managed host. The file should be owned by the `root` user and the `root` group. Also set 0644 as the file permissions.

Add the following lines to the playbook to define a task that uses the `copy` module to copy the contents of the file defined by the `httpdconf_src` variable to the location specified by the `httpdconf_file` variable.

```
- name: Configure web service
 copy:
 src: "{{ httpdconf_src }}"
 dest: "{{ httpdconf_file }}"
 owner: root
 group: root
 mode: 0644
```

5. Add a third task that uses the `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 should be owned by the `apache` user and the `apache` group. Also set 0500 as the file permissions.

Add the following lines to the playbook to define a task that uses the `file` module to create the directory defined by the `secrets_dir` variable.

```
- name: Secrets directory exists
 file:
 path: "{{ secrets_dir }}"
 state: directory
 owner: apache
 group: apache
 mode: 0500
```

6. Add a fourth task that uses the `htpasswd` module to create an `htpasswd` file to be used for basic authentication of web users. The `path` attribute should be defined by the

`secrets_file` variable. The file should contain an entry for the user defined by the `web_user` variable. The user's password will be defined by the `web_pass` variable to be added later. The file should be owned by the `apache` user and the `apache` group. Also set **0400** as the file permissions.

Add the following lines to the playbook to define a task that uses the `htpasswd` module to create the password file defined by the `secrets_file` variable. Define the user to be added using the `web_user` variable. Define the user's password using the `web_pass` variable, which will be defined in an encrypted file later.

```
- name: Web user exists in secrets file
 htpasswd:
 path: "{{ secrets_file }}"
 name: "{{ web_user }}"
 password: "{{ web_pass }}"
 owner: apache
 group: apache
 mode: 0400
```

7. Add a fifth task that uses the `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 should be owned by the `apache` user and the `apache` group. Set 0400 as the file permissions.

Add the following lines to the playbook to define a task which uses the `copy` module to create the `.htaccess` file using the file defined by the `htaccess_src` variable.

```
- name: .htaccess file installed in docroot
 copy:
 src: "{{ htaccess_src }}"
 dest: "{{ web_root }}/.htaccess"
 owner: apache
 group: apache
 mode: 0400
```

8. Add a sixth task that uses the `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 to the `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 playbook to define a task that uses the `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
 copy:
 content: "{{ ansible_facts['fqdn'] }} ({{ ansible_facts['default_ipv4'] }}['address']) has been customized by Ansible.\n"
 dest: "{{ web_root }}/index.html"
```

9. Add a seventh task that uses the `firewalld` module to open up on the firewall the `https` service 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 playbook to define a task that uses the `firewalld` module to open the HTTPS port for the web service. You can use the `firewalld` service `https` to do this.

```
- name: Open the port for the web server
 firewalld:
 service: https
 state: enabled
 immediate: true
 permanent: true
```

10. Add an eighth task that uses the `service` module to enable and restart 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.

Add the following lines to the playbook to define a task that uses the `service` module to enable and restart the web service.

```
- name: Web service enabled and restarted
 service:
 name: "{{ web_svc }}"
 state: restarted
 enabled: true
```

11. Add a final task that uses the `service` module to enable and restart the firewall service on the managed host so that all configuration changes take effect.

- 11.1. Add the following lines to the playbook to define a task that uses the `service` module to enable and restart the firewall service.

```
- name: Firewall service enable and restarted
 service:
 name: firewalld
 state: restarted
 enabled: true
```

- 11.2. The completed playbook should appear as follows:

```

- name: Install and configure webserver with basic auth
 hosts: webserver
 vars:
 firewall_pkg: firewalld
 web_pkg: httpd
 web_svc: httpd
 ssl_pkg: mod_ssl
 python_pkg: python-passlib
 httpdconf_src: files/httpd.conf
 httpdconf_file: /etc/httpd/conf/httpd.conf
 htaccess_src: files/.htaccess
 secrets_dir: /etc/httpd/secrets
 secrets_file: "{{ secrets_dir }}/htpasswd"
```

```
web_root: /var/www/html
web_user: guest
vars_files:
 - vars/secret.yml
tasks:
 - name: Latest version of necessary packages installed
 yum:
 name:
 - "{{ firewall_pkg }}"
 - "{{ web_pkg }}"
 - "{{ ssl_pkg }}"
 - "{{ python_pkg }}"
 state: latest
 - name: Configure web service
 copy:
 src: "{{ httpdconf_src }}"
 dest: "{{ httpdconf_file }}"
 owner: root
 group: root
 mode: 0644
 - name: Secrets directory exists
 file:
 path: "{{ secrets_dir }}"
 state: directory
 owner: apache
 group: apache
 mode: 0500
 - name: Web user exists in secrets file
 htpasswd:
 path: "{{ secrets_file }}"
 name: "{{ web_user }}"
 password: "{{ web_pass }}"
 owner: apache
 group: apache
 mode: 0400
 - name: .htaccess file installed in docroot
 copy:
 src: "{{ htaccess_src }}"
 dest: "{{ web_root }}/.htaccess"
 owner: apache
 group: apache
 mode: 0400
 - name: Create index.html
 copy:
 content: "{{ ansible_facts['fqdn'] }} ({{ ansible_facts['default_ipv4']['address'] }}) has been customized by Ansible.\n"
 dest: "{{ web_root }}/index.html"
 - name: Open the port for the web server
 firewalld:
 service: https
 state: enabled
 immediate: true
 permanent: true
 - name: Web service enabled and restarted
 service:
 name: "{{ web_svc }}"
```

```
state: restarted
enabled: true
- name: Firewall service enable and restarted
 service:
 name: firewalld
 state: restarted
 enabled: true
```

- 11.3. Exit the text editor and save the **playbook.yml** playbook.
12. Create a file encrypted with Ansible Vault, named **vars/secret.yml**. It should set the `web_pass` variable to **redhat**, which will be the web user's password.
- 12.1. Create a subdirectory named **vars** in the project directory.

```
[student@workstation data-review]$ mkdir vars
```

- 12.2. 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
```

- 12.3. Add the following variable definition to the file.
- ```
web_pass: redhat
```
- 12.4. Exit the file editor and save the changes to the file.
13. Run the **playbook.yml** playbook.

- 13.1. Before running the playbook, verify that its syntax is correct by running **ansible-playbook --syntax-check**. Use the **--ask-vault-pass** 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-playbook --syntax-check \
> --ask-vault-pass playbook.yml
Vault password: redhat

playbook: playbook.yml
```

- 13.2. Using the **ansible-playbook** command, run the playbook with the **--ask-vault-pass** option. Enter **redhat** when prompted for the password.

```
[student@workstation data-review]$ ansible-playbook playbook.yml --ask-vault-pass
Vault password: redhat
PLAY [Install and configure webserver with basic auth] ****
...output omitted...

PLAY RECAP ****
```

```
serverb.lab.example.com      : ok=10    changed=9     unreachable=0      failed=0
```

14. On *workstation*, use **curl** to ensure that the web server is reachable over HTTPS. Verify that basic authentication is working by authenticating as the **guest** user and using **redhat** as the password.

- 14.1. On *workstation*, use **curl** to ensure that the web server has been successfully started and is reachable. Use the **-u** option to specify **guest** as the user for authentication. Also use the **-k** to disable verification of the web server's SSL certificate. When prompted, enter **redhat** as the password.

If the following message appears, it indicates that the web server has been installed, the firewall has been updated with a new rule, and that basic authentication is working.

```
[student@workstation data-review]$ curl https://serverb.lab.example.com \
> -k -u guest
Enter host password for user 'guest': redhat
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

Evaluation

Run the **lab data-review grade** command on *workstation* to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

Cleanup

On *workstation*, run the **lab data-review cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab data-review cleanup
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Ansible *variables* allow administrators to reuse values across files in an entire Ansible project.
- Variables can be defined for hosts and host groups in the inventory file.
- Variables can be defined for playbooks by using facts and external files. They can also be defined on the command line.
- The **register** keyword can be used to capture the output of a command in a variable.
- It is better to store inventory variables in files in the **host_vars** and **group_vars** directory relative to the inventory, rather than in the inventory file itself.
- Ansible Vault is one way to protect sensitive data such as password hashes and private keys for deployment using Ansible Playbooks.
- Ansible Vault can be used to create and encrypt a text file if it does not already exist or to encrypt and decrypt existing files.
- Red Hat recommends that users keep most variables in a normal file and sensitive variables in a second file protected by Ansible Vault.
- Ansible *facts* are variables that are automatically discovered by Ansible from a managed host.

CHAPTER 5

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

After completing this section, students should be able to use loops to write efficient tasks, and use conditions to control when to run tasks.

TASK ITERATION WITH LOOPS

Using loops saves administrators from the need to write multiple tasks that use the same module. For example, instead of writing five tasks to ensure five users exist, you can write one task that iterates over a list of five users to ensure they all exist.

Ansible supports iterating a task over a set of items using 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 `service` module twice in order to ensure two network services are running:

```
- name: Postfix is running
  service:
    name: postfix
    state: started

- name: Dovecot is running
  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 both services are running:

```
- name: Postfix and Dovecot are running
  service:
    name: "{{ item }}"
    state: started
  loop:
    - postfix
    - dovecot
```

The list used by **loop** can be provided by a variable. In the following example, the variable **mail_services** contains the list of services that need to be running.

```

vars:
  mail_services:
    - postfix
    - dovecot

tasks:
  - name: Postfix and Dovecot are running
    service:
      name: "{{ item }}"
      state: started
    loop: "{{ mail_services }}"

```

Loops over a List of Hashes/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 hash/dictionary. Each hash/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 were prefixed with **with_**, followed by the name of an Ansible look-up plug-in (an advanced feature that is 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.

A few examples are listed in the table below:

Earlier-Style Ansible Loops

LOOP KEYWORD	DESCRIPTION
with_items	Behaves the same as the loop keyword for simple lists, such as a list of strings or a list of hashes/dictionaries. Unlike loop , if lists of lists are provided to with_items , they are flattened into a single-level list. The loop variable item holds the list item used during each iteration.

LOOP KEYWORD	DESCRIPTION
with_file	This keyword requires a list of control node file names. The loop variable item holds the content of a corresponding file from the file list during each iteration.
with_sequence	Instead of requiring a list, this keyword requires parameters to generate a list of values based on a numeric sequence. The loop variable item holds the value of one of the generated items in the generated sequence during each iteration.

An example of **with_items** in a playbook is shown below:

```
vars:
  data:
    - user0
    - user1
    - user2
tasks:
  - name: "with_items"
    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 old 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 old 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. There is a good reference on how to convert the old loops to the new syntax, as well as examples of how to loop over items that are not simple lists, in the Ansible documentation in the section "Migrating from with_X to loop" [https://docs.ansible.com/ansible/2.7/user_guide/playbooks_loops.html#migrating-from-with-x-to-loop] of the *Ansible User Guide*.

You will likely encounter tasks from older 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
  shell: "echo This is my item: {{ item }}"
  loop:
    - one
    - two

  register: echo_results 1

- name: Show echo_results variable
  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 above playbook yields the following output:

```

[student@workstation loopdemo]$ ansible-playbook loop_register.yml
PLAY [Loop Register Test] *****
TASK [Looping Echo Task] *****
...output omitted...
TASK [Show echo_results variable] *****
ok: [localhost] => {
  "echo_results": {1
    "changed": true,
    "msg": "All items completed",
    "results": [2
      {3
        "_ansible_ignore_errors": null,
        ...output omitted...
        "changed": true,
        "cmd": "echo This is my item: one",
        "delta": "0:00:00.011865",
        "end": "2018-11-01 16:32:56.080433",
        "failed": false,
        ...output omitted...
        "item": "one",
        "rc": 0,
        "start": "2018-11-01 16:32:56.068568",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "This is my item: one",
        "stdout_lines": [
          "This is my item: one"
        ]
      },
      {4
        "_ansible_ignore_errors": null,
        ...output omitted...
        "changed": true,
        "cmd": "echo This is my item: two",
        "delta": "0:00:00.011142",
        "end": "2018-11-01 16:32:56.080433",
        "failed": false,
        "item": "two",
        "rc": 0,
        "start": "2018-11-01 16:32:56.068568",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "This is my item: two",
        "stdout_lines": [
          "This is my item: two"
        ]
      }
    ]
  }
}
```

```
        "end": "2018-11-01 16:32:56.828196",
        "failed": false,
        ...output omitted...
      "item": "two",
      "rc": 0,
      "start": "2018-11-01 16:32:56.817054",
      "stderr": "",
      "stderr_lines": [],
      "stdout": "This is my item: two",
      "stdout_lines": [
        "This is my item: two"
      ]
    }
  ]
}
...output omitted...
```

- ➊ The { character indicates the start of the **echo_results** variable is composed of key-value pairs.
- ➋ The **results** key contains the results from the previous task. The [character indicates the start of a list.
- ➌ 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.
- ➍ The start of task result metadata for the second item.
- ➎ The] character indicates the end of the **results** list.

In the above, the **results** key contains a list. Below, the playbook is modified such 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
      shell: "echo This is my item: {{ item }}"
      loop:
        - one
        - two
      register: echo_results

    - name: Show stdout from the previous task.
      debug:
        msg: "STDOUT from previous task: {{ item.stdout }}"
      loop: echo_results['results']
```

After executing the above playbook, the output is:

```
PLAY [Loop Register Test] ****
TASK [Looping Echo Task] ****
...output omitted...
```

```

TASK [Show stdout from the previous task.] ****
ok: [localhost] => (item={...output omitted...}) => {
    "msg": "STDOUT from previous task: This is my item: one"
}
ok: [localhost] => (item={...output omitted...}) => {
    "msg": "STDOUT from previous task: This is my item: two"
}
...output omitted...

```

RUNNING TASKS CONDITIONALLY

Ansible can use *conditionals* to execute tasks or plays when certain conditions are met. For example, a conditional can be used to determine the available memory on a managed host before Ansible installs or configures a service.

Conditionals allow administrators 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:

- A hard limit can be defined in a variable (for example, `min_memory`) and compared against the available memory on a managed host.
- The output of a command can be captured and evaluated by Ansible to determine whether or not 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).
- The number of CPUs can be evaluated 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
      yum:
        name: httpd
      when: run_my_task

```

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"
      yum:
        name: "{{ my_service }}"
        when: my_service is defined
```

The following table shows some of the operations that administrators can use when working with conditionals:

Example Conditionals

OPERATION	EXAMPLE
Equal (value is a string)	<code>ansible_machine == "x86_64"</code>
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> . The values of <code>0</code> , <code>False</code> , or <code>no</code> evaluate to <code>false</code> .	<code>memory_available</code>
Boolean variable is <code>false</code> .	<code>not memory_available</code>
First variable's value is present as a value in second variable's list	<code>ansible_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_distribution` variable is a fact determined during the **Gathering Facts** task, and identifies the managed host's operating system distribution. The variable `supported_distros` was created by the playbook author, and contains a list of operating system distributions that the playbook supports. If the value of `ansible_distribution` is in the `supported_distros` list, the conditional passes and the task runs.

```
---
```

```
- name: Demonstrate the "in" keyword
  hosts: all
  vars:
    supported_distros:
      - RedHat
      - Fedora
  tasks:
    - name: Install httpd using yum, where supported
      yum:
        name: http
        state: present
      when: ansible_distribution in supported_distros
```



IMPORTANT

Notice 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 hash/dictionary, and the `when` statement is simply one more key in the task 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 you should 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_distribution == "RedHat" or ansible_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 will be met if the remote host is a Red Hat Enterprise Linux 7.5 host, and the installed kernel is the specified version:

```
when: ansible_distribution_version == "7.5" and ansible_kernel ==
  "3.10.0-327.el7.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 of 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_distribution_version == "7.5"
  - ansible_kernel == "3.10.0-327.el7.x86_64"
```

This format improves readability, a key goal of well-written Ansible Playbooks.

- More complex conditional statements can be expressed 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 7 or Fedora 28. 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_distribution == "RedHat" and
    ansible_distribution_major_version == "7" )
  or
  ( ansible_distribution == "Fedora" and
    ansible_distribution_major_version == "28" )
```

COMBINING LOOPS AND CONDITIONAL TASKS

You can combine loops and conditionals.

In the following example, the *mariadb-server* package is installed by the **yum** module if there is a file system mounted on / with more than 300 MB free. The **ansible_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 representing a mounted file system where both conditions are true.

```
- name: install mariadb-server if enough space on root
  yum:
    name: mariadb-server
    state: latest
  loop: "{{ ansible_mounts }}"
  when: item.mount == "/" and item.size_available > 300000000
```



IMPORTANT

When you use **when** with **loop** for a task, the **when** statement is checked for each item.

Here is another example that combines conditionals and register variables. The following annotated playbook will restart 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
```

```

command: /usr/bin/systemctl is-active postfix 1
ignore_errors: yes 2
register: result 3

- name: Restart Apache HTTPD based on Postfix status
  service:
    name: httpd
    state: restarted
    when: result.rc == 04

```

- 1** Is Postfix running or not?
- 2** If it is not running and the command fails, do not stop processing.
- 3** Saves information on the module's result in a variable named `result`.
- 4** Evaluates 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/2.7/user_guide/playbooks_loops.html

Tests – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_tests.html

Conditionals – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_conditionals.html

What Makes A Valid Variable Name – Variables – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html#what-makes-a-valid-variable-name

► GUIDED EXERCISE

WRITING LOOPS AND CONDITIONAL TASKS

In this exercise, you will write a playbook containing tasks that have conditionals and loops.

OUTCOMES

You should be able to:

- Implement Ansible conditionals using the **when** keyword.
- Implement task iteration using the **loop** keyword in conjunction with conditionals.

Scenario Overview

This exercise contains a playbook, **database_setup.yml**, that installs a database and creates user accounts on designated remote hosts. As implemented, the playbook can only support installing a database on a Red Hat Enterprise Linux remote host.

Through the use of conditional tasks, the playbook is structured in a way to support the gradual addition of other Linux distributions at a later time. The tasks associated with creating users are kept in a separate file, **database_users_tasks.yml**, because these tasks can also be used with other Linux distributions.

Both the **database_setup.yml** and **database_users_tasks.yml** files are incomplete. As a result, the **database_setup.yml** playbook is not functional. In this exercise, you will implement **loop** and **when** keywords for tasks in these files to create a correctly functioning playbook.

On **workstation**, run the lab setup script to confirm the environment is ready for the lab to begin. The script creates the working directory, called **control-flow**, and populates it with an Ansible configuration file, a host inventory, and partially completed playbook files.

```
[student@workstation ~]$ lab control-flow setup
```

- 1. As the student user on **workstation**, change to the **/home/student/control-flow** directory.

```
[student@workstation ~]$ cd ~/control-flow
[student@workstation control-flow]$
```

- 2. Review the **database_setup.yml** playbook file. Add appropriate conditional statements to each task in the playbook file.

- 2.1. Review the **database_setup.yml** file using a text editor.

```
---
- name: Database Setup play
  hosts: database_servers
```

```

vars:
  min_ram_size_bytes: 20000000000 ①
  supported_distros: ②
    - RedHat
    #- Centos
tasks:
  - name: Setup Database tasks on supported hosts w/ Min. RAM
    include_tasks: "{{ ansible_distribution }}_database_tasks.yml"
    #Add a conditional here③

  - name: Print a message for unsupported Distros
    debug:
      msg: >
        {{ inventory_hostname }} is a
        {{ ansible_distribution }}-based host, which is not one
        of the supported distributions ({{ supported_distros }})

    #Add a conditional here④

  - name: Print a message for systems with insufficient RAM
    debug:
      msg: >
        {{ inventory_hostname }} does not meet the minimum
        RAM requirements of {{ min_ram_size_bytes }} bytes.

    #Add a conditional here⑤

```

- ① ② These variables define the minimum RAM requirements for database servers and a list of supported Linux distributions for database hosts. Your database servers must have at least 2 GB of RAM and the list of supported distributions is currently just a single item: **RedHat** (matching Red Hat Enterprise Linux).
- ③ The first task, which includes tasks to install a database, should only be executed if the remote server meets two criteria. First, the operating system of the remote server must be one of the distributions specified by **supported_distros** variable. Second, the remote server must have the minimum RAM specified by the playbook's **min_ram_size_bytes** variable. If either of these criteria are not met, the database installation tasks are not included in the play.
- ④ The second task prints a message if a remote server does not match one of the supported operating system distributions.
- ⑤ The third task prints a message if a remote server does not meet the RAM requirement for a database server, specified by the **min_ram_size_bytes** variable.

2.2. Edit the first task in the **database_setup.yml** file.

```

- name: Setup Database tasks on supported hosts w/ Min. RAM
  include_tasks: "{{ ansible_distribution }}_database_tasks.yml"
  #Add a conditional here

```

Replace the **#Add a conditional here** with a **when** statement to test the Linux distribution and RAM requirements. The variable **ansible_memtotal_mb** is an

Ansible fact that provides the RAM of the remote host, in mebibytes (MiB, binary megabytes). Recall that 1 MiB equals $1024 * 1024$ (1048576) bytes.

After editing, save the file. The task should now appear as follows:

```
- name: Setup Database tasks on supported hosts w/ Min. RAM
  include_tasks: "{{ ansible_distribution }}_database_tasks.yml"
  when:
    - ansible_distribution in supported_distros
    - ansible_memtotal_mb*1024*1024 >= min_ram_size_bytes
```

2.3. Edit the second task in the `database_setup.yml` file.

```
- name: Print a message for unsupported Distros
  debug:
    msg: >
      {{ inventory_hostname }} is a
      {{ ansible_distribution }}-based host, which is not one
      of the supported distributions ({{ supported_distros }})
  #Add a conditional here
```

Replace **#Add a conditional here** with a `when` statement to test that the remote host is not one of the supported distributions. After editing, save the file. The task should now appear as follows:

```
- name: Print a message for unsupported Distros
  debug:
    msg: >
      {{ inventory_hostname }} is a
      {{ ansible_distribution }}-based host, which is not one
      of the supported distributions ({{ supported_distros }})
  when: ansible_distribution not in supported_distros
```

2.4. Edit the third task in the `database_setup.yml` file.

```
- name: Print a message for systems with insufficient RAM
  debug:
    msg: >
      {{ inventory_hostname }} does not meet the minimum
      RAM requirements of {{ min_ram_size_bytes }} bytes.
  #Add a conditional here
```

Replace **#Add a conditional here** with a `when` statement to test if the remote does not meet the RAM requirements. After editing, save the file. The task should now appear as follows:

```
- name: Print a message for systems with insufficient RAM
  debug:
    msg: >
      {{ inventory_hostname }} does not meet the minimum
      RAM requirements of {{ min_ram_size_bytes }} bytes.
```

```
when: ansible_memtotal_mb*1024*1024 < min_ram_size_bytes
```

When you have completed the edits, the **database_setup.yml** file should contain the following:

```
---
```

```
- name: Database Setup play
  hosts: database_servers
  vars:
    min_ram_size_bytes: 2000000000
    supported_distros:
      - RedHat
      #- Centos
  tasks:
    - name: Setup Database tasks on supported hosts w/ Min. RAM
      include_tasks: "{{ ansible_distribution }}_database_tasks.yml"
      when:
        - ansible_distribution in supported_distros
        - ansible_memtotal_mb*1024*1024 >= min_ram_size_bytes

    - name: Print a message for unsupported Distros
      debug:
        msg: >
          {{ inventory_hostname }} is a
          {{ ansible_distribution }}-based host, which is not one
          of the supported distributions ({{ supported_distros }})
      when: ansible_distribution not in supported_distros

    - name: Print a message for systems with insufficient RAM
      debug:
        msg: >
          {{ inventory_hostname }} does not meet the minimum
          RAM requirements of {{ min_ram_size_bytes }} bytes.
      when: ansible_memtotal_mb*1024*1024 < min_ram_size_bytes
```

- 3. If the remote host meets the RAM requirements and is installed with Red Hat Enterprise Linux, the tasks from the **RedHat_database_tasks.yml** file are used to perform database installation. Review the **RedHat_database_tasks.yml** file.

```
#For RHEL, using mariadb as the database service
- name: Set the 'db_service' fact
  set_fact:①
  db_service: mariadb

#RHEL packages for mariadb service
- name: Ensure database packages are installed
  yum:
    name:②
    - mariadb-server
    - mariadb-bench
    - mariadb-libs
    - mariadb-test

- name: Ensure the database service is started
```

```
service:  
  name: "{{ db_service }}"  
  state: started  
  enabled: true  
  
#Below tasks are also reused by other distros  
- name: Create Database Users  
  
  include_tasks: database_user_tasks.yml③
```

- ➊ The `set_fact` module defines a variable, `db_service`, set to `mariadb`. This allows each distribution to utilize a different database service.
 - ➋ A simple loop should not be used to install multiple packages. Some modules, such as the `yum` module, support using a list as a parameter value.
 - ➌ A task list is included for creating user accounts on the remote database host. These tasks are saved in a separate file to facilitate task reuse for other Linux distributions.
- ▶ 4. Edit the `database_users_tasks.yml` task file. Update the first task with a simple loop to create all of the permission groups defined in the group variable `host_permission_groups`. The `host_permission_groups` variable is defined in the file `group_vars/database_servers.yml`.
- 4.1. Review the contents of the `group_vars/database_servers.yml` file.

```
[student@workstation control-flow]$ cat group_vars/database_servers.yml  
host_permission_groups:  
- dbadmin  
- dbuser
```

The file defines a simple list variable, `host_permission_groups`, which contains group names. These group names only apply for hosts in the `database_servers` host group.

- 4.2. Open the `database_users_tasks.yml` task file in an editor. Review the first task in the `database_users_tasks.yml` task file.

```
- name: Ensure database permission groups exist  
group:  
  name:  
  state: present  
#Add a loop
```

- 4.3. Replace the line `#Add a loop` with a loop statement to iterate over the values of the `host_permission_groups`. Replace the value of the `name` keyword with the appropriate loop iteration variable.

The first task now contains:

```
- name: Ensure database permission groups exist  
group:  
  name: "{{ item }}"  
  state: present  
loop: "{{ host_permission_groups }}"
```

- 5. Update the second task of the **database_users_tasks.yml** file to loop over users in the `user_list` variable. The `user_list` variable is defined for all inventory hosts in the **group_vars/all.yml** file.

Add each created user to a group identical to the user's **role** value. Use the user's **username** value for the value of the **name** keyword in the `user` module.

Lastly, add a **when** statement to ensure that a user is only created if that user's role matches one of the values present in the `host_permission_groups` variable.

- 5.1. Review the contents of the **group_vars/all.yml** file.

```
[student@workstation control-flow]$ cat group_vars/all.yml
user_list:
  - name: John Davis
    username: jdavis
    role: dbadmin
  - name: Jennifer Smith
    username: jsmith
    role: dbuser
...output omitted...
```

The file defines a list variable, `user_list`, which contains a hash/dictionary representing a unique user. For each user, a **name**, **username**, and **role** attribute are defined. In this exercise, the **role** attribute corresponds to a Linux permission group. This variable is defined for all hosts in the inventory, including hosts in the **database_servers** host group.

- 5.2. Review the second task in the **database_users_tasks.yml** file.

```
- name: Ensure Database Users exist
  user:
    name:
    groups:
    append: yes
```

```
state: present
```

- 5.3. Add a **loop** statement to iterate over the values in the `user_list` variable.

```
loop: "{{ user_list }}"
```

- 5.4. Update the value of the **name** keyword of the `user` module to be the user's **username**.

```
name: "{{ item.username }}"
```

- 5.5. Update the value of the **groups** keyword of the `user` module to be the user's **role**.

```
groups: "{{ item.role }}"
```

- 5.6. Add a **when** statement to the second task that tests if a user's **role** matches one of the values in the `host_permission_groups` variable.

```
when: item.role in host_permission_groups
```

The second task should now contain the following:

```
- name: Ensure Database Users exist
  user:
    name: "{{ item.username }}"
    groups: "{{ item.role }}"
    append: yes
    state: present
  loop: "{{ user_list }}"
  when: item.role in host_permission_groups
```

- 6. Check the syntax of the `database_setup.yml` playbook using the **ansible-playbook --syntax-check** command:

```
[student@workstation control-flow]$ ansible-playbook \
> --syntax-check database_setup.yml

playbook: database_setup.yml
```

- 7. Execute the `database_setup.yml` playbook using the **ansible-playbook** command.

```
[student@workstation control-flow]$ ansible-playbook database_setup.yml

PLAY [Database Setup play] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Setup Database tasks on supported hosts w/ Min. RAM] ****
skipping: [servera.lab.example.com]
```

```

TASK [Print a message for unsupported Distros] ****
skipping: [servera.lab.example.com]

TASK [Print a message for systems with insufficient RAM] ****
ok: [servera.lab.example.com] => {
    "msg": "servera.lab.example.com does not meet the minimum RAM
requirements of 2000000000 bytes.\n"
}

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

```

The output indicates that `servera` does not meet minimum RAM requirements. As a result, a database is not installed on `servera`.

- 8. Change the playbook's minimum RAM requirements from 2 GB to 500 MB. Execute the playbook again and review the results.

- 8.1. Open the `database_setup.yml` playbook in a text editor. Update the value of the `min_ram_size_bytes` variable to **500000000**, and save the changes. The top of the playbook now matches the snippet below:

```

---
- name: Database Setup play
  hosts: database_servers
  vars:
    min_ram_size_bytes: 500000000
    supported_distros:
      - RedHat

```

Save the file.

- 8.2. Execute the `database_setup.yml` playbook again.

```

[student@workstation control-flow]$ ansible-playbook database_setup.yml

PLAY [Database Setup play] ****

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

TASK [Setup Database tasks on supported hosts w/ Min. RAM] ****
included: ...output omitted.../RedHat_database_tasks.yml ...output omitted

TASK [Set the 'db_service' fact] ****
ok: [servera.lab.example.com]

TASK [Ensure database packages are installed] ****
changed: [servera.lab.example.com]

TASK [Ensure the database service is started] ****
changed: [servera.lab.example.com]

TASK [Create Database Users] ****
included: ...output omitted.../database_user_tasks.yml ...output omitted...

```

```
TASK [Ensure database permission groups exist] ****
ok: [servera.lab.example.com] => (item=dbadmin)
ok: [servera.lab.example.com] => (item=dbuser)

TASK [Ensure Database Users exist] ****
changed: [servera.lab.example.com] => (item={u'username': u'jdavis', u'role': u'dbadmin', u'name': u'John Davis'})
changed: [servera.lab.example.com] => (item={u'username': u'jsmith', u'role': u'dbuser', u'name': u'Jennifer Smith'})
skipping: [servera.lab.example.com] => (item={u'username': u'sjohnson', u'role': u'webadmin', u'name': u'Sarah Johnson'})
skipping: [servera.lab.example.com] => (item={u'username': u'mjones', u'role': u'webdev', u'name': u'Matt Jones'})

TASK [Print a message for unsupported Distros] ****
skipping: [servera.lab.example.com]

TASK [Print a message for systems with insufficient RAM] ****
skipping: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=8      changed=4      unreachable=0      failed=0
```

The output indicates that a database was successfully installed. Users with a database-related role can now log in to `servera`.

Cleanup

On `workstation`, run the `lab control-flow cleanup` script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab control-flow cleanup
```

This concludes the guided exercise.

IMPLEMENTING HANDLERS

OBJECTIVES

After completing this section, students should be able to implement a task that runs only when another task changes the managed host.

Figure 5.0: Implementing Handlers

ANSIBLE HANDLERS

Ansible modules are designed to be *idempotent*. This means that in a properly written playbook, the playbook and its tasks can be run multiple times without changing the managed host unless they need to make a change to get the managed host to the desired state.

However, sometimes when a task does make a change to the system, a further task may need to be run. For example, a change to a service's configuration file may 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 has a globally unique name and is triggered at the end of a block of tasks in a playbook. If no task notifies the handler by name then the handler will not run. If one or more tasks notify the handler, the handler will run exactly 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.

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
    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
    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 triggers when notified by the **template** task that a change happened. A task may 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  
    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  
    service:  
      name: mariadb  
      state: restarted  
  
  - name: restart apache  
    service:  
      name: httpd  
      state: restarted
```

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 will not run until *all* of the tasks under **tasks** have been processed. (There are some minor exceptions to this.)
- Handler names exist in a global namespace. If two handlers are incorrectly given the same name, only one will run.
- Even if more than one task notifies a handler, the handler will only run once. If no tasks notify it, a handler will 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. The handler is skipped unless another task notifies it. 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

Intro to Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_intro.html

► GUIDED EXERCISE

IMPLEMENTING HANDLERS

In this exercise, you will implement handlers in playbooks.

OUTCOMES

You should be able to:

- Define handlers in playbooks and notify them for configuration change.

Run **lab control-handlers setup** on workstation to configure the environment for the exercise. This script creates the **control-handlers** project directory and downloads the Ansible configuration file and the host inventory file needed for the exercise. The project directory also contains a partially complete playbook, **configure_db.yml**.

```
[student@workstation ~]$ lab control-handlers setup
```

- 1. On `workstation.lab.example.com`, open a new terminal and change to the `~/control-handlers` project directory.

```
[student@workstation ~]$ cd ~/control-handlers
[student@workstation control-handlers]$
```

- 2. In that directory, use a text editor to edit the **configure_db.yml** playbook file. This playbook will install and configure a database server. When the database server configuration changes, the playbook triggers a restart of the database service and configures the database administrative password.
- 2.1. Using a text editor, review the **configure_db.yml** playbook. It begins with the initialization of some variables:

```
---
- name: MariaDB server is installed
  hosts: databases
  vars:
    db_packages:
      - mariadb-server
      - MySQL-python
    db_service: mariadb
    resources_url: http://materials.example.com/labs/control-handlers
    config_file_url: "{{ resources_url }}/{my.cnf.standard}"
    config_file_dst: /etc/my.cnf
  tasks:
```

- db_packages:** Defines the name of the packages to install for the database service.
- db_service:** Which defines the name of the database service.

- **resources_url**: The URL for the resource directory where remote configuration files are located.
 - **config_file_url**: The URL of the database configuration file to install.
 - **config_file_dst**: The location of the installed configuration file on the managed hosts.
- 2.2. In the **configure_db.yml** file, define a task that uses the `yum` module to install the required database packages as defined by the **db_packages** variable. If the task changes the system, the database was not installed, and you need to notify the **set db password** handler to set your initial database user and password. Remember that the handler task, if it is notified, will not run until every task in the **tasks** section has run.

The task should read as follows:

```
tasks:  
  - name: "{{ db_packages }} packages are installed"  
    yum:  
      name: "{{ db_packages }}"  
      state: present  
    notify:  
      - set db password
```

- 2.3. Add a task to start and enable the database service. The task should read as follows:

```
- name: Make sure the database service is running  
  service:  
    name: "{{ db_service }}"  
    state: started  
    enabled: true
```

- 2.4. Add a task to download **my.cnf.standard** to `/etc/my.cnf` on the managed host, using the `get_url` module. Add a condition that notifies the **restart db service** handler to restart the database service after a configuration file change. The task should read:

```
- name: The {{ config_file_dst }} file has been installed  
  get_url:  
    url: "{{ config_file_url }}"  
    dest: "{{ config_file_dst }}"  
    owner: mysql  
    group: mysql  
    force: yes  
  notify:  
    - restart db service
```

- 2.5. Add the **handlers** keyword to define the start of the handler tasks. Define the first handler, **restart db service**, which restarts the `mariadb` service. It should read as follows:

```
handlers:  
  - name: restart db service
```

```
service:  
  name: "{{ db_service }}"  
  state: restarted
```

- 2.6. Define the second handler, **set db password**, which sets the administrative password for the database service. The handler uses the `mysql_user` module to perform the command. The handler should read as follows:

```
- name: set db password  
  mysql_user:  
    name: root  
    password: redhat
```

When completed, the playbook should appear as follows:

```
---  
- name: MariaDB server is installed  
  hosts: databases  
  vars:  
    db_packages:  
      - mariadb-server  
      - MySQL-python  
    db_service: mariadb  
    resources_url: http://materials.example.com/labs/control-handlers  
    config_file_url: "{{ resources_url }}/my.cnf.standard"  
    config_file_dst: /etc/my.cnf  
  tasks:  
    - name: "{{ db_packages }} packages are installed"  
      yum:  
        name: "{{ db_packages }}"  
        state: present  
      notify:  
        - set db password  
  
    - name: Make sure the database service is running  
      service:  
        name: "{{ db_service }}"  
        state: started  
        enabled: true  
  
    - name: The {{ config_file_dst }} file has been installed  
      get_url:  
        url: "{{ config_file_url }}"  
        dest: "{{ config_file_dst }}"  
        owner: mysql  
        group: mysql  
        force: yes  
      notify:  
        - restart db service  
  
  handlers:  
    - name: restart db service  
      service:  
        name: "{{ db_service }}"  
        state: restarted
```

```
- name: set db password
  mysql_user:
    name: root
    password: redhat
```

- 3. Before running the playbook, verify that its syntax is correct by running **ansible-playbook** 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 control-handlers]$ ansible-playbook configure_db.yml \
> --syntax-check

playbook: configure_db.yml
```

- 4. Run the **configure_db.yml** playbook. The output shows that the handlers are being executed.

```
[student@workstation control-handlers]$ ansible-playbook configure_db.yml

PLAY [Installing MariaDB server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install [u'mariadb-server', u'MySQL-python'] package] ****
changed: [servera.lab.example.com]

TASK [Make sure the database service is running] ****
changed: [servera.lab.example.com]

TASK [The /etc/my.cnf file has been installed] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [restart db service] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [set db password] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=6      changed=5      unreachable=0      failed=0
```

- 5. Run the playbook again.

```
[student@workstation control-handlers]$ ansible-playbook configure_db.yml

PLAY [Installing MariaDB server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [[u'mariadb-server', u'MySQL-python'] packages are installed] ****
```

```
ok: [servera.lab.example.com]

TASK [Make sure the database service is running] ****
ok: [servera.lab.example.com]

TASK [The /etc/my.cnf file has been installed] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=4    changed=0    unreachable=0    failed=0
```

This time the handlers are skipped. In the event that the remote configuration file is changed in the future, executing the playbook would trigger the **restart db service** handler but not the **set db password** handler.

Cleanup

On workstation, run the **lab control-handlers cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab control-handlers cleanup
```

This concludes the guided exercise.

HANDLING TASK FAILURE

OBJECTIVES

After completing this section, students should be able to control what happens when a task fails, and what conditions cause a task to fail.

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 aborts the rest of the play on that host, skipping 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 running some other task conditionally. There are a number of Ansible features that 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 `yum` module will fail, but having `ignore_errors` set to `yes` allows execution to continue.

```
- name: Latest version of notapkg is installed
  yum:
    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 will 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.

The following snippet shows how to use the `force_handlers` keyword in a play to force execution of the handler even if a task fails:

```
---
- hosts: all
  force_handlers: yes
  tasks:
    - name: a task which always notifies its handler
      command: /bin/true
      notify: restart the database

    - name: a task which fails because the package doesn't exist
```

```
yum:  
  name: notapkg  
  state: latest  
  
handlers:  
  - name: restart the database  
    service:  
      name: mariadb  
      state: restarted
```

**NOTE**

Remember that handlers are notified when a task reports a **changed** result but are not notified when it reports an **ok** or **failed** result.

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 may successfully execute a command, but the command's output indicates a failure.

For example, you can run a script that outputs an error message and use that message to define the failed state for the task. The following snippet shows how the **failed_when** keyword can be used in a task:

```
tasks:  
  - name: Run user creation script  
    shell: /usr/local/bin/create_users.sh  
    register: command_result  
    failed_when: "'Password missing' in command_result.stdout"
```

The **fail** module can also be used to force a task failure. The above scenario can alternatively be written as two tasks:

```
tasks:  
  - name: Run user creation script  
    shell: /usr/local/bin/create_users.sh  
    register: command_result  
    ignore_errors: yes  
  
  - name: Report script failure  
    fail:  
      msg: "The password is missing in the output"  
      when: "'Password missing' in command_result.stdout"
```

You can use the **fail** module to provide a clear failure message for the task. This approach also enables delayed failure, allowing you to 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.

The **changed_when** keyword can be used to control when a task reports that it has changed. For example, the **shell** module in the next example is being used to get a Kerberos credential which will be used by subsequent tasks. It normally would always report **changed** when it runs. To suppress that change, **changed_when: false** is set so that it only reports **ok** or **failed**.

```
- name: get Kerberos credentials as "admin"
  shell: echo "{{ krb_admin_pass }}" | kinit -f admin
  changed_when: false
```

The following example uses the **shell** module to report **changed** based on the output of the module that is collected by a registered variable:

```
tasks:
  - shell:
      cmd: /usr/local/bin/upgrade-database
      register: command_result
      changed_when: "'Success' in command_result.stdout"
      notify:
        - restart_database

handlers:
  - name: restart_database
    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:
  - block:
      - name: package needed by yum
        yum:
          name: yum-plugin-versionlock
          state: present
      - name: lock version of tzdata
        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, tasks in its **rescue** block are executed in order 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 will 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. Even if tasks defined in the **block** clause fail, tasks defined in the **rescue** and **always** clauses are executed.

```
tasks:  
  - block:  
    - name: upgrade the database  
      shell:  
        cmd: /usr/local/lib/upgrade-database  
  rescue:  
    - name: revert the database upgrade  
      shell:  
        cmd: /usr/local/lib/revert-database  
  always:  
    - name: always restart the database  
      service:  
        name: mariadb  
        state: restarted
```

The **when** condition on a **block** clause also applies to its **rescue** and **always** clauses if present.

Figure 5.0: Handling task failure



REFERENCES

Error Handling in Playbooks – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_error_handling.html

Error Handling – Blocks – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_blocks.html#error-handling

► GUIDED EXERCISE

HANDLING TASK FAILURE

In this exercise, you will explore different ways to handle task failure in an Ansible Playbook.

OUTCOMES

You should be able to:

- 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/always in playbooks.

On **workstation**, run the lab setup script to confirm the environment is ready for the lab to begin. This script creates the working directory, **~/control-errors**.

```
[student@workstation ~]$ lab control-errors setup
```

- 1. On **workstation.lab.example.com**, change to the **~/control-errors** project directory.

```
[student@workstation ~]$ cd ~/control-errors  
[student@workstation control-errors]$
```

- 2. The lab script created an Ansible configuration file as well as an inventory file that contains the server **servera.lab.example.com** in the **databases** group. Review the file before proceeding.

- 3. Create the **playbook.yml** playbook, which contains a play with two tasks. Write the first task to contain a deliberate error that will cause it to fail.

- 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**. The variables will be used to install the required packages and start the server.

The **http** value is an intentional error in the package name. The file should read as follows:

```
---  
- 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 **yum** module and the two variables, `web_package` and `db_package`. The tasks will install the required packages. The tasks should read as follows:

```
tasks:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: present

  - name: Install {{ db_package }} package
    yum:
      name: "{{ db_package }}"
      state: present
```

- 4. Run the playbook and watch the output of the play.

```
[student@workstation control-errors]$ ansible-playbook 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, "msg": "No package matching 'http' found available, installed or updated", "rc": 126, "results": ...output omitted...}
to retry, use: --limit @/home/student/control-errors/playbook.retry

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

The task failed because there is no existing package called **http**. Because the first task failed, the second task was not run.

- 5. Update the first task to ignore any errors by adding the **ignore_errors** keyword. The tasks should read as follows:

```
tasks:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: present
      ignore_errors: yes

  - name: Install {{ db_package }} package
    yum:
      name: "{{ db_package }}"
      state: present
```

- 6. Run the playbook again and watch the output of the play.

```
[student@workstation control-errors]$ ansible-playbook 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, "msg": "No
package matching 'http' found available, installed or updated", "rc": 126,
"results": ...output omitted...}
...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
```

Despite the fact that the first task failed, Ansible executed the second one.

- 7. In this step, you will set up a **block** keyword so 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 read as follows:

```
- block:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: present
```

- 7.2. Nest the task that installs the *mariadb-server* package in a **rescue** clause. The task will execute if the task listed in the **block** clause fails. The block clause should read as follows:

```
rescue:
  - name: Install {{ db_package }} package
    yum:
      name: "{{ db_package }}"
      state: present
```

- 7.3. Finally, add an **always** clause to start the database server upon installation using the **service** module. The clause should read as follows:

```
always:
  - name: Start {{ db_service }} service
    service:
      name: "{{ db_service }}"
```

```
state: started
```

7.4. The completed task section should read as follows:

```
tasks:
  - block:
    - name: Install {{ web_package }} package
      yum:
        name: "{{ web_package }}"
        state: present
  rescue:
    - name: Install {{ db_package }} package
      yum:
        name: "{{ db_package }}"
        state: present
  always:
    - name: Start {{ db_service }} service
      service:
        name: "{{ db_service }}"
        state: started
```

► 8. Now 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. Then the task in the **always** block runs.

```
[student@workstation control-errors]$ ansible-playbook 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, "msg": "No package matching 'http' found available, installed or updated", "rc": 126, "results": [...output omitted...]}

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=1
```

8.2. Edit the playbook, correcting the value of the **web_package** variable to read **httpd**. That will cause 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** will still run.

```
[student@workstation control-errors]$ ansible-playbook 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
```

- ▶ 9. This step explores how to control the condition that causes a task to be reported as "changed" to the managed host.
- 9.1. Edit the playbook to add two tasks to the start of the play, preceding the **block**. The first task uses the **command** module to run the **date** command and register the result in the **command_result** variable. The second task uses the **debug** module to print the standard output of the first task's command.

```
tasks:
  - name: Check local time
    command: date
    register: command_result

  - name: Print local time
    debug:
      var: command_result.stdout
```

- 9.2. Run the playbook. You should see that the first task, which runs the **command** module, reports **changed**, even though it did not change the remote system; it only collected information about the time. That is because the **command** module cannot tell the difference between a command that collects data and a command that changes state.

```
[student@workstation control-errors]$ ansible-playbook 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": "Fri Nov  9 15:30:39 EST 2018"
}

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
```

If you run the playbook again, the **Check local time** task returns **changed** again.

- 9.3. That **command** task should not report **changed** every time it runs because it is not changing the managed host. Because you know that the task will never change a managed host, add the line **changed_when: false** to the task to suppress the change.

```
tasks:
  - name: Check local time
    command: date
    register: command_result
    changed_when: false

  - name: Print local time
    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-playbook 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": "Fri Nov  9 15:35:39 EST 2018"
}

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
```

- 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 will report failure when you run the play.

Be careful with your indentation to make sure the keyword is correctly set on the task.

```
- block:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: present
    failed_when: web_package == "httpd"
```

- 10.2. Run the playbook.

```
[student@workstation control-errors]$ ansible-playbook 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": "Fri Nov  9 15:40:37 EST 2018"
}

TASK [Install httpd package] *****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false,
"failed_when_result": true, "msg": "", "rc": 0, "results": [
"httpd-2.4.6-80.el7.x86_64 providing httpd is already installed"]}

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=1
```

Look carefully at the output. The **Install httpd package** task *reports* that it failed, but it actually ran and made sure the package is installed first. The

failed_when keyword changes the status 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.

Cleanup

On workstation, run the **lab control-errors cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab control-errors cleanup
```

This concludes the guided exercise.

► LAB

IMPLEMENTING TASK CONTROL

PERFORMANCE CHECKLIST

In this lab, you will install the Apache web server and secure it using mod_ssl. You will use conditions, handlers, and task failure handling in your playbook to deploy the environment.

OUTCOMES

You should be able to define conditionals in Ansible Playbooks, set up loops that iterate over elements, define handlers in playbooks, and handle task errors.

Log in as the student user on workstation and run **lab control-review setup**. This script ensures that the managed host, serverb, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab control-review setup
```

1. On workstation.lab.example.com, change to the ~/control-review project directory.
2. The project directory contains a partially completed playbook, **playbook.yml**. Using a text editor, add a task that uses the **fail** module under the **#Fail Fast Message** comment. Be sure to provide an appropriate name for the task. This task should only be executed when the remote system does not meet the minimum requirements.

The minimum requirements for the remote host are listed below:

- Has at least the amount of RAM specified by the **min_ram_megabytes** variable. The **min_ram_megabytes** variable is defined in the **vars.yml** file and has a value of **256**.
 - Is running Red Hat Enterprise Linux.
3. Add a single task to the playbook under the **#Install all Packages** comment to install any missing packages. Required packages are specified by the **packages** variable, which is defined in the **vars.yml** file.

The task name should be **Ensure required packages are present**. Use the **package** module to install packages, not the **yum** module.

4. Add a single task to the playbook under the **#Enable and start services** comment to start all services. All services specified by the **services** variable, which is defined in the **vars.yml** file, should be started and enabled. Be sure to provide an appropriate name for the task.
5. Add a task block to the playbook under the **#Block of config tasks** comment. This block contains two tasks:
 - A task to ensure 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 message: **One or more of the configuration changes failed, but the web service is still active..**

Be sure to provide an appropriate name for all tasks.

6. The playbook configures the remote host to listen for standard HTTPS requests. Add a single task to the playbook under the **#Configure the firewall** comment to configure firewalld.

This task should ensure that the remote host allows standard HTTP and HTTPS connections. These configuration changes should be effective immediately and persist after a system reboot. Be sure to 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 project directory, `~/control-review`, 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 custom certificate expires on August 9, 2021, and the web server response should match the string **Configured for both HTTP and HTTPS.**

Evaluation

Run the **lab control-review grade** command on workstation to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

Cleanup

Run the **lab control-review cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab control-review cleanup
```

► SOLUTION

IMPLEMENTING TASK CONTROL

PERFORMANCE CHECKLIST

In this lab, you will install the Apache web server and secure it using mod_ssl. You will use conditions, handlers, and task failure handling in your playbook to deploy the environment.

OUTCOMES

You should be able to define conditionals in Ansible Playbooks, set up loops that iterate over elements, define handlers in playbooks, and handle task errors.

Log in as the student user on workstation and run **lab control-review setup**. This script ensures that the managed host, serverb, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab control-review setup
```

1. On workstation.lab.example.com, change to the ~/control-review project directory.

```
[student@workstation ~]$ cd ~/control-review
[student@workstation control-review]$
```

2. The project directory contains a partially completed playbook, **playbook.yml**. Using a text editor, add a task that uses the fail module under the **#Fail Fast Message** comment. Be sure to provide an appropriate name for the task. This task should only be executed when the remote system does not meet the minimum requirements.

The minimum requirements for the remote host are listed below:

- Has at least the amount of RAM specified by the **min_ram_megabytes** variable. The **min_ram_megabytes** variable is defined in the **vars.yml** file and has a value of **256**.
- Is running Red Hat Enterprise Linux.

The completed task matches:

```
tasks:
  #Fail Fast Message
  - name: Show Failed System Requirements Message
    fail:
      msg: "The {{ inventory_hostname }} did not meet minimum reqs."
    when: >
      ansible_memtotal_mb*1024*1024 < min_ram_megabytes*1000000 or
      ansible_distribution != "RedHat"
```

3. Add a single task to the playbook under the **#Install all Packages** comment to install any missing packages. Required packages are specified by the packages variable, which is defined in the **vars.yml** file.

The task name should be **Ensure required packages are present**. Use the package module to install packages, not the yum module.

The completed task matches:

```
#Install all Packages
- name: Ensure required packages are present
  package:
    name: "{{ item }}"
    state: present
  loop: "{{ packages }}"
```

4. Add a single task to the playbook under the **#Enable and start services** comment to start all services. All services specified by the services variable, which is defined in the **vars.yml** file, should be started and enabled. Be sure to provide an appropriate name for the task.

The completed task matches:

```
#Enable and start services
- name: Ensure services are started and enabled
  service:
    name: "{{ item }}"
    state: started
    enabled: yes
  loop: "{{ services }}"
```

5. Add a task block to the playbook under the **#Block of config tasks** comment. This block contains two tasks:

- A task to ensure 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 message: **One or more of the configuration changes failed, but the web service is still active..**

Be sure to provide an appropriate name for all tasks.

The completed task block matches below:

```
#Block of config tasks
- block:
  - name: Create SSL cert directory
    file:
      path: "{{ ssl_cert_dir }}"
      state: directory

  - name: Copy Config Files
    copy:
```

```
src: "{{ item.src }}"
dest: "{{ item.dest }}"
loop: "{{ web_config_files }}"
notify: restart web service

rescue:
- name: Configuration Error Message
debug:
msg: >
    One or more of the configuration
    changes failed, but the web service
    is still active.
```

6. The playbook configures the remote host to listen for standard HTTPS requests. Add a single task to the playbook under the **#Configure the firewall** comment to configure firewalld.

This task should ensure that the remote host allows standard HTTP and HTTPS connections. These configuration changes should be effective immediately and persist after a system reboot. Be sure to provide an appropriate name for the task.

The completed task matches:

```
#Configure the firewall
- name: ensure web server ports are open
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.

A **handlers** section is added to the end of the playbook:

```
handlers:
- name: restart web service
service:
  name: "{{ web_service }}"
  state: restarted
```

The completed playbook contains:

```
---
- name: Playbook Control Lab
hosts: webservers
vars_files: vars.yml
tasks:
#Fail Fast Message
- name: Show Failed System Requirements Message
fail:
msg: "The {{ inventory_hostname }} did not meet minimum reqs."
```

```

when: >
  ansible_memtotal_mb*1024*1024 < min_ram_megabytes*1000000 or
  ansible_distribution != "RedHat"

#Install all Packages
- name: Ensure required packages are present
  package:
    name: "{{ item }}"
    state: present
  loop: "{{ packages }}"

#Enable and start services
- name: Ensure services are started and enabled
  service:
    name: "{{ item }}"
    state: started
    enabled: yes
  loop: "{{ services }}"

#Block of config tasks
- block:
    - name: Create SSL cert directory
      file:
        path: "{{ ssl_cert_dir }}"
        state: directory

    - name: Copy Config Files
      copy:
        src: "{{ item.src }}"
        dest: "{{ item.dest }}"
      loop: "{{ web_config_files }}"
      notify: restart web service

  rescue:
    - name: Configuration Error Message
      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
  firewalld:
    service: "{{ item }}"
    immediate: true
    permanent: true
    state: enabled
  loop:
    - http
    - https

#Add handlers
handlers:
  - name: restart web service
    service:

```

```
name: "{{ web_service }}"
state: restarted
```

8. From the project directory, `~/control-review`, run the `playbook.yml` playbook. The playbook should execute without errors, and trigger the execution of the handler task.

```
[student@workstation control-review]$ ansible-playbook 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] => (item=httpd)
changed: [serverb.lab.example.com] => (item=mod_ssl)
ok: [serverb.lab.example.com] => (item=firewalld)

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={'dest': '/etc/httpd/conf.d/ssl',
  'src': 'server.key'})
changed: [serverb.lab.example.com] => (item={'dest': '/etc/httpd/conf.d/ssl',
  'src': 'server.crt'})
changed: [serverb.lab.example.com] => (item={'dest': '/etc/httpd/conf.d',
  'src': 'ssl.conf'})
changed: [serverb.lab.example.com] => (item={'dest': '/var/www/html', 'src':
  'index.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
```

9. Verify that the web server now responds to HTTPS requests, using the self-signed custom certificate to encrypt the connection. The custom certificate expires on August 9, 2021, and the web server response should match the string **Configured for both HTTP and HTTPS.**

```
[student@workstation control-review]$ curl -k -vvv https://serverb.lab.example.com
```

```
* About to connect() to serverb.lab.example.com port 443 (#0)
*   Trying 172.25.250.11...
* Connected to serverb.lab.example.com (172.25.250.11) port 443 (#0)
* Initializing NSS with certpath: sql:/etc/pki/nssdb
* skipping SSL peer certificate verification
* SSL connection using TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
* Server certificate:
...output omitted...
*   start date: Nov 13 15:52:18 2018 GMT
*   expire date: Aug 09 15:52:18 2021 GMT
*   common name: serverb.lab.example.com
...output omitted...
< 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

Run the **lab control-review grade** command on workstation to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

Cleanup

Run the **lab control-review cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab control-review cleanup
```

SUMMARY

In this chapter, you learned:

- Loops are used to iterate over a set of values, a simple list of strings, or a list of hashes/dictionaries.
- Conditionals are used to execute tasks or plays only when certain conditions have been met.
- Conditions are tested with various operators, including string comparisons, mathematical operators, and Boolean values.
- 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 are configured to handle error conditions by ignoring task failure, forcing handlers to be called even if the task failed, mark a task as failed when it succeeded, or override the behavior that causes a task to be marked as changed.
- Blocks are used to group tasks as a unit and execute other tasks depending on whether or not all the tasks in the block succeed.

CHAPTER 6

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 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

After completing this section, students should be able to create, install, edit, and remove files on managed hosts, and manage permissions, ownership, SELinux context, and other characteristics of those files.

DESCRIBING FILES MODULES

Red Hat Ansible Engine ships with a large collection of modules (the "module library") that are developed as part of the upstream Ansible project. To make it easier to organize, document, and manage them, they are organized into groups based on function in the documentation and when installed on a system.

The **Files** modules library includes modules that allow you to accomplish most tasks related to Linux file management, 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 Files Modules

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 a particular line is in a file, or replace an existing line using a backreference 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.

MODULE NAME	MODULE DESCRIPTION
synchronize	A wrapper around the rsync command to make common tasks quick and easy. 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 may still need to call the rsync command directly via a run command module depending on your use case.

AUTOMATION EXAMPLES WITH FILES MODULES

Creating, copying, editing, and removing files on managed hosts are common tasks that you can implement using modules from the **Files** modules library. 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 **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 makes sure that the owning user, group, and permissions of the file are set to specific values.

```
- name: Touch a file and set permissions
  file:
    path: /path/to/file
    owner: user1
    group: group1
    mode: 0640
    state: touch
```

Example outcome:

```
$ ls -l file
-rw-r----- user1 group1 0 Nov 25 08:00 file
```

Modifying File Attributes

You can use the **file** module to ensure 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.

```
$ 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
  file:
    path: /path/to/samba_file
    setype: samba_share_t
```

Example outcome:

```
$ ls -Z samba_file
-rw-r--r--  owner group unconfined_u:object_r:samba_share_t:s0 samba_file
```

**NOTE**

File attribute parameters are available in multiple file management modules. Run the **ansible-doc file** and **ansible-doc copy** commands for additional information.

Making SELinux File Context Changes Persistent

The **file** module acts like **chcon** when setting file contexts. Changes made with that module could be unexpectedly undone by running **restorecon**. After using **file** to set the context, you can use **sefcontext** from the collection of System modules to update the SELinux policy like **semanage fcontext**.

```
- name: SELinux type is persistently set to samba_share_t
  sefcontext:
    target: /path/to/samba_file
    setype: samba_share_t
    state: present
```

Example outcome:

```
$ ls -Z samba_file
-rw-r--r--  owner group unconfined_u:object_r:samba_share_t:s0 samba_file
```

**IMPORTANT**

The **sefcontext** module updates the default context for the target in the SELinux policy, but does not change the context on existing files.

Copying and Editing Files on Managed Hosts

In this example, the **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 **force: yes** is set. That forces the module to overwrite the remote file if it exists but contains different contents from 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
  copy:
    src: file
    dest: /path/to/file
```

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
  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 `blockinfile` module:

```
- name: Add additional lines to a file
  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 `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 `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
  file:
    dest: /path/to/file
    state: absent
```

Retrieving the Status of a File on Managed Hosts

The `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 `stat` module returns a hash/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 `stat` module 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 to make the example output more legible.)

```
- name: Verify the checksum of a file
  stat:
    path: /path/to/file
    checksum_algorithm: md5
  register: result

- 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 `stat` module are documented by [ansible-doc](#), or you can register a variable and display its contents to see what is available:

```
- name: Examine all stat output of /etc/passwd
  hosts: localhost

  tasks:
    - name: stat /etc/passwd
      stat:
        path: /etc/passwd
      register: results

    - name: Display stat results
      debug:
        var: results
```

Synchronizing Files Between the Control Node and Managed Hosts

The `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 `synchronize` module, the "local host" is the host that the `synchronize` task originates on (usually the control node), and the "destination host" is the host that `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
  synchronize:
    src: file
    dest: /path/to/file
```

There are many ways to use the `synchronize` module and its many parameters, including synchronizing directories. Run the `ansible-doc synchronize` command for additional parameters and playbook examples.



REFERENCES

`ansible-doc(1)`, `chmod(1)`, `chown(1)`, `rsync(1)`, `stat(1)` and `touch(1)` man pages

Files modules

https://docs.ansible.com/ansible/2.7/modules/list_of_files_modules.html

► GUIDED EXERCISE

MODIFYING AND COPYING FILES TO HOSTS

In this exercise, you will 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

You should be able to:

- Retrieve files from managed hosts and store them locally by host name.
- Create playbooks that use common file management modules such as `copy`, `file`, `lineinfile`, and `blockinfile`.

Run the `lab file-manage setup` script on workstation to configure the environment for the exercise. The script creates the `file-manage` project directory, and downloads the Ansible configuration file and the host inventory file needed for the exercise.

```
[student@workstation ~]$ lab file-manage setup
```

- 1. As the `student` user on workstation, change to the `/home/student/file-manage` working directory.

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

- 2. Create a playbook called `secure_log_backups.yml` in the current working directory. Configure the playbook to use the `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_log_backups` directory with subdirectories named after the host name of each managed host. Store the backup files in their respective subdirectories.

- 2.1. Create the `secure_log_backups.yml` playbook with initial content:

```
---
- name: Use the fetch module to retrieve secure log files
  hosts: all
  remote_user: root
```

- 2.2. Add a task to the `secure_log_backups.yml` playbook that retrieves the `/var/log/secure` log file from the managed hosts and stores it in the `~/file-manage/secure-backups` directory. If the `~/file-manage/secure-backups` directory does not exist, it is created automatically by the `fetch` module. 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
    fetch:
      src: /var/log/secure
      dest: secure-backups
      flat: no
```

- 2.3. Before running the playbook, run the **ansible-playbook --syntax-check secure_log_backups.yml** command to verify its syntax. Correct any errors before moving to the next step.

```
[student@workstation file-manage]$ ansible-playbook --syntax-check \
> secure_log_backups.yml

playbook: secure_log_backups.yml
[student@workstation file-manage]$
```

- 2.4. Run **ansible-playbook secure_log_backups.yml** to execute the playbook:

```
[student@workstation file-manage]$ ansible-playbook 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
serverb.lab.example.com      : ok=2      changed=1      unreachable=0      failed=0
```

- 2.5. Verify the playbook results:

```
[student@workstation file-manage]$ tree -F -P secure
...output omitted...
└── secure-backups/
    ├── servera.lab.example.com/
    │   └── var/
    │       └── log/
    │           └── secure
    └── serverb.lab.example.com/
        └── var/
            └── log/
                └── secure
...output omitted...
```

- 3. Create the **copy_file.yml** playbook in the current working directory. Configure the playbook to copy the **/home/student/file-manage/files/users.txt** file to managed hosts.

- 3.1. Add the following initial content to the **copy_file.yml** playbook:

Parameters

PARAMETER	VALUES
name	Using the copy module
hosts	all
remote_user	root

```
---
```

```
- name: Using the copy module
  hosts: all
  remote_user: root
```

- 3.2. Add a task to use the copy module to copy the **/home/student/file-manage/files/users.txt** file to all managed hosts. Use the copy module to set the following parameters for the **users.txt** file:

Parameters

PARAMETER	VALUES
src	files/users.txt
dest	/home/devops/users.txt
owner	devops
group	devops
mode	u+rw, g-wx, o-rwx
setype	samba_share_t

```
tasks:
- name: Copy a file to managed hosts and set attributes
  copy:
    src: files/users.txt
    dest: /home/devops/users.txt
    owner: devops
    group: devops
    mode: u+rw, g-wx, o-rwx
```

```
setype: samba_share_t
```

- 3.3. Use the **ansible-playbook --syntax-check copy_file.yml** command to verify the syntax of the **copy_file.yml** playbook.

```
[student@workstation file-manage]$ ansible-playbook --syntax-check copy_file.yml
playbook: copy_file.yml
```

- 3.4. Run the playbook:

```
[student@workstation file-manage]$ ansible-playbook 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
serverb.lab.example.com : ok=2      changed=1      unreachable=0      failed=0
```

- 3.5. Use an ad hoc command to execute the **ls -Z** command as user devops to verify the attributes of the **users.txt** file on the managed hosts.

```
[student@workstation file-manage]$ ansible all -m command -a 'ls -Z' -u devops
servera.lab.example.com | CHANGED | rc=0 >
-rw-r-----. devops devops unconfined_u:object_r:samba_share_t:s0 users.txt

serverb.lab.example.com | CHANGED | rc=0 >
-rw-r-----. devops devops unconfined_u:object_r:samba_share_t:s0 users.txt
```

- ▶ 4. In a previous step, the **samba_share_t** SELinux type field was set for the **users.txt** file. However, it is now determined that default values should be set for the SELinux file context. Create a playbook called **selinux_defaults.yml** in the current working directory. Configure the playbook to use the **file** module to ensure the default SELinux context for user, role, type, and level fields.

- 4.1. Create the **selinux_defaults.yml** playbook:

```
---
- name: Using the file module to ensure SELinux file context
hosts: all
remote_user: root
tasks:
  - name: SELinux file context is set to defaults
    file:
      path: /home/devops/users.txt
      seuser: _default
```

```
serole: _default
setype: _default
selevel: _default
```

- 4.2. Use the **ansible-playbook --syntax-check selinux_defaults.yml** command to verify the syntax of the **selinux_defaults.yml** playbook.

```
[student@workstation file-manage]$ ansible-playbook --syntax-check \
> selinux_defaults.yml

playbook: selinux_defaults.yml
```

- 4.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-playbook 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
serverb.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

- 4.4. Use an ad hoc command to execute the **ls -Z** command as user devops to verify the default file attributes of **unconfined_u:object_r:user_home_t:s0**.

```
[student@workstation file-manage]$ ansible all -m command -a 'ls -Z' -u devops
servera.lab.example.com | CHANGED | rc=0 >>
-rw-r-----. devops devops unconfined_u:object_r:user_home_t:s0 users.txt

serverb.lab.example.com | CHANGED | rc=0 >>
-rw-r-----. devops devops unconfined_u:object_r:user_home_t:s0 users.txt
```

- ▶ 5. Create a playbook called **add_line.yml** in the current working directory. Configure the playbook to use the **lineinfile** module to append the line **This line was added by the lineinfile module.** to the **/home/devops/users.txt** file on all managed hosts.

- 5.1. Create the **add_line.yml** playbook:

```
---
- name: Add text to an existing file
  hosts: all
  remote_user: devops
  tasks:
    - name: Add a single line of text to a file
      lineinfile:
```

```
path: /home/devops/users.txt
line: This line was added by the lineinfile module.
state: present
```

- 5.2. Use **ansible-playbook --syntax-check add_line.yml** command to verify the syntax of the **add_line.yml** playbook.

```
[student@workstation file-manage]$ ansible-playbook --syntax-check add_line.yml

playbook: add_line.yml
```

- 5.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-playbook 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
serverb.lab.example.com : ok=2      changed=1      unreachable=0      failed=0
```

- 5.4. Use the command module with the **cat** option, as the devops user, to verify the content of the **users.txt** file on the managed hosts.

```
[student@workstation file-manage]$ ansible all -m command \
> -a 'cat users.txt' -u devops
serverb.lab.example.com | CHANGED | rc=0 >>
This line was added by the lineinfile module.

servera.lab.example.com | CHANGED | rc=0 >>
This line was added by the lineinfile module.
```

- ▶ 6. Create a playbook called **add_block.yml** in the current working directory. Configure the playbook to use the **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.

- 6.1. Create the **add_block.yml** playbook:

```
---
- name: Add block of text to a file
hosts: all
remote_user: devops
```

```
tasks:
  - name: Add a block of text to an existing file
    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
```

- 6.2. Use the **ansible-playbook --syntax-check add_block.yml** command to verify the syntax of the **add_block.yml** playbook.

```
[student@workstation file-manage]$ ansible-playbook --syntax-check add_block.yml
playbook: add_block.yml
```

- 6.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-playbook 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
serverb.lab.example.com      : ok=2      changed=1      unreachable=0      failed=0
```

- 6.4. Use the command module with the **cat** command to verify the correct content of the **/home/devops/users.txt** file on the managed host.

```
[student@workstation file-manage]$ ansible all -m command \
> -a 'cat users.txt' -u devops
serverb.lab.example.com | CHANGED | rc=0 >>
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

servera.lab.example.com | CHANGED | rc=0 >>
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
```

- 7. Create a playbook called **remove_file.yml** in the current working directory. Configure the playbook to use the **file** module to remove the **/home/devops/users.txt** file from all managed hosts.

- 7.1. Create the **remove_file.yml** playbook:

```
---
- name: Use the file module to remove a file
  hosts: all
  remote_user: devops
  tasks:
    - name: Remove a file from managed hosts
      file:
        path: /home/devops/users.txt
        state: absent
```

- 7.2. Use the **ansible-playbook --syntax-check remove_file.yml** command to verify the syntax of the **remove_file.yml** playbook.

```
[student@workstation file-manage]$ ansible-playbook --syntax-check remove_file.yml
playbook: remove_file.yml
```

- 7.3. Run the playbook:

```
[student@workstation file-manage]$ ansible-playbook 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
serverb.lab.example.com    : ok=2    changed=1    unreachable=0    failed=0
```

- 7.4. Use an ad hoc command to execute the **ls -l** command to confirm that the **users.txt** file no longer exists on the managed hosts.

```
[student@workstation file-manage]$ ansible all -m command -a 'ls -l'
serverb.lab.example.com | CHANGED | rc=0 >>
total 0

servera.lab.example.com | CHANGED | rc=0 >>
total 0
```

Cleanup

On workstation, run the **lab file-manage cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab file-manage cleanup
```

This concludes the guided exercise.

DEPLOYING CUSTOM FILES WITH JINJA2 TEMPLATES

OBJECTIVES

After completing this section, students should be able to deploy files to managed hosts that are customized by using Jinja2 templates.

TEMPLATING FILES

Red Hat Ansible Engine has 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 bit about how to use it.

Using Delimiters

Variables and logic expressions are placed between tags, or delimiters. For example, Jinja2 templates use `{% EXPR %}` for expressions or logic (for example, loops), while `{{ EXPR }}` are used for outputting the results of an expression or a variable to the end user. The latter tag, when rendered, is replaced with a value or values, and are seen by the end user. Use `{# COMMENT #}` syntax to enclose comments that should not appear in the final file.

In the following example the first line includes a comment that will not be 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 on a template.



NOTE

Remember that the facts associated with a managed host can be obtained using the `ansible system_hostname -i inventory_file -m setup` command.

The following example shows how to create a template with variables using two of the facts retrieved by Ansible from managed hosts: `ansible_facts.hostname` and

`ansible_facts.date_time.date`. When the associated playbook is executed, those two facts are replaced by their values in the managed host being configured.

**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 may make it easier for you to remember that it is a template file.

```
Welcome to {{ ansible_facts.hostname }}.  
Today's date is: {{ ansible_facts.date_time.date }}.
```

DEPLOYING JINJA2 TEMPLATES

Jinja2 templates are a powerful tool to customize configuration files to be deployed on the managed hosts. When the Jinja2 template for a configuration file has been created, it can be deployed to the managed hosts using the `template` module, which supports the transfer of a local file on the control node to the managed hosts.

To use the `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  
    template:  
      src: /tmp/j2-template.j2  
      dest: /tmp/dest-config-file.txt
```

**NOTE**

The `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 `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 copying it into place.

For more details, see [ansible-doc template](#).

MANAGING TEMPLATED FILES

To avoid having system administrators modify files deployed 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 in the `ansible_managed` directive. This is not a normal variable but can be used as one in a template. The `ansible_managed` directive is set in the `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 `user` variable is replaced with all the values included in the `users` variable, one value per line.

```
{% for user in users %}
    {{ user }}
{% endfor %}
```

The following example template uses a **for** statement to run through all the values in the `users` variable, replacing `myuser` with each value, except when 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, and assumes a `myhosts` variable has been defined in the inventory file being used. This variable would contain a list of hosts to be managed. With the following **for** statement, all hosts in the `myhosts` group from the inventory would be listed in the file.

```
{% for myhost in groups['myhosts'] %}
{{ myhost }}
{% endfor %}
```

For a more practical example, you can use this 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
      template:
        src: templates/hosts.j2
        dest: /etc/hosts
```

The following 3-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 (for example, to JSON). There are filters 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, to parse them.

```
  {{ output | from_json }}  
  {{ output | from_yaml }}
```

The expressions used with **when** clauses in Ansible Playbooks are Jinja2 expressions. Built-in Ansible filters used to test return values include **failed**, **changed**, **succeeded**, and **skipped**. The following task shows how filters can be used inside of conditional expressions.

```
tasks:  
  ...output omitted...  
  - debug: msg="the execution was aborted"  
    when: returnvalue | failed
```

Figure 6.0: Deploying custom files with Jinja2 templates



REFERENCES

template - Templates a file out to a remote server – Ansible Documentation

https://docs.ansible.com/ansible/2.7/modules/template_module.html

Variables – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html

Filters – 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 will create a simple template file that your playbook will use to install a customized Message of the Day file on each managed host.

OUTCOMES

You should be able to:

- Build a template file.
- Use the template file in a playbook.

Log in to `workstation` as `student` using `student` as the password.

On `workstation`, run the `lab file-template setup` script. This script ensures that Ansible is installed on `workstation`, creates the `/home/student/file-template` directory, and downloads the `ansible.cfg` file into that directory.

```
[student@workstation ~]$ lab file-template setup
```



NOTE

All the files used during this exercise are available for reference on `workstation` in the `/home/student/file-template/files` directory.

- 1. On `workstation`, navigate to the `/home/student/file-template` working directory.

```
[student@workstation ~]$ cd ~/file-template  
[student@workstation file-template]$
```

- 2. Create the `inventory` file in the current working directory. This file configures two groups: `webservers` and `workstations`. Include the `servera.lab.example.com` system in the `webservers` group, and the `workstation.lab.example.com` system in the `workstations` group.

```
[webservers]  
servera.lab.example.com  
  
[workstations]  
workstation.lab.example.com
```

- 3. Create a template for the Message of the Day and include it in the `motd.j2` file in the current working directory. Include the following variables in the template:

- `ansible_hostname`, to retrieve the host name of the managed host.
- `ansible_date_time.date`, for the date of the managed host.
- `system_owner`, for the system owner's email. This variable needs to be defined with an appropriate value in the `vars` section of the playbook template.

```
This is the system {{ ansible_hostname }}.  
Today's date is: {{ ansible_date_time.date }}.  
Only use this system with permission.  
You can ask {{ system_owner }} for access.
```

- ▶ 4. Create a playbook file named `motd.yml` in the current working directory. Define the `system_owner` variable in the `vars` section, and include a task for the `template` module that maps the `motd.j2` Jinja2 template to the remote file `/etc/motd` on the managed hosts. Set the owner and group to root, and the mode to 0644.

```
---  
- hosts: all  
  remote_user: devops  
  become: true  
  vars:  
    system_owner: clyde@example.com  
  tasks:  
    - template:  
        src: motd.j2  
        dest: /etc/motd  
        owner: root  
        group: root  
        mode: 0644
```

- ▶ 5. Before running the playbook, use the `ansible-playbook --syntax-check` command to verify the 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-playbook --syntax-check motd.yml  
  
playbook: motd.yml
```

- ▶ 6. Run the playbook included in the `motd.yml` file.

```
[student@workstation file-template]$ ansible-playbook motd.yml  
PLAY [all] ****  
  
TASK [Gathering Facts] ****  
ok: [servera.lab.example.com]  
ok: [workstation.lab.example.com]  
  
TASK [template] ****  
changed: [servera.lab.example.com]  
changed: [workstation.lab.example.com]
```

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

- 7. Log in to `servera.lab.example.com` as the `devops` user to verify that the MOTD is correctly displayed when logged in. Log off when you have finished.

```
[student@workstation file-template]$ ssh devops@servera.lab.example.com
This is the system servera.
Today's date is: 2017-07-21.
Only use this system with permission.
You can ask clyde@example.com for access.
[devops@servera ~]# exit
Connection to servera.lab.example.com closed.
```

Cleanup

Run the `lab file-template cleanup` command to clean up after the exercise.

```
[student@workstation ~]$ lab file-template cleanup
```

► LAB

DEPLOYING FILES TO MANAGED HOSTS

PERFORMANCE CHECKLIST

In this lab, you will run a playbook that creates a customized file on your managed hosts by using a Jinja2 template.

OUTCOMES

You should be able to:

- Build a template file.
- Use the template file in a playbook.

Log in to workstation as student using **student** as the password.

On workstation, run the **lab file-review setup** script. It ensures that Ansible is installed on workstation, creates the **/home/student/file-review** directory, and downloads the **ansible.cfg** file into that directory. It also downloads the **motd.yml**, **motd.j2**, and **inventory** files into the **/home/student/file-review/files** directory.

```
[student@workstation ~]$ lab file-review setup
```



NOTE

All files used in this exercise are available on workstation in the **/home/student/file-review/files** directory.

1. Create an inventory file named **inventory** 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 the status of the system memory.
3. Create a template for the Message of the Day, named **motd.j2**, in the current working directory. When the **devops** user logs in to **serverb.lab.example.com**, a message should display that shows the system's total memory and current free memory. Use the **ansible_memtotal_mb** and **ansible_memfree_mb** facts to provide the memory information for the message.
4. Create a new playbook file called **motd.yml** in the current directory. Using the **template** module, configure the **motd.j2** Jinja2 template file previously created to map to the file **/etc/motd** on the managed hosts. This file has the **root** user as owner and group, and its permissions are **0644**. Configure the playbook so that it uses the **devops** user, and sets the **become** parameter to **true**.
5. Run the playbook included in the **motd.yml** file.
6. Check that the playbook included in the **motd.yml** file has been executed correctly.

Evaluation

On workstation, run the **lab file-review grade** script to confirm success on this exercise.

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

Cleanup

On workstation, run the **lab file-review cleanup** script to clean up after the lab.

```
[student@workstation ~]$ lab file-review cleanup
```

► SOLUTION

DEPLOYING FILES TO MANAGED HOSTS

PERFORMANCE CHECKLIST

In this lab, you will run a playbook that creates a customized file on your managed hosts by using a Jinja2 template.

OUTCOMES

You should be able to:

- Build a template file.
- Use the template file in a playbook.

Log in to workstation as student using **student** as the password.

On workstation, run the **lab file-review setup** script. It ensures that Ansible is installed on workstation, creates the **/home/student/file-review** directory, and downloads the **ansible.cfg** file into that directory. It also downloads the **motd.yml**, **motd.j2**, and **inventory** files into the **/home/student/file-review/files** directory.

```
[student@workstation ~]$ lab file-review setup
```



NOTE

All files used in this exercise are available on workstation in the **/home/student/file-review/files** directory.

1. Create an inventory file named **inventory** 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/
```

- 1.2. Create the **inventory** file in the current directory. This file configures one group, called **servers**. Include the **serverb.lab.example.com** system in the **servers** group.

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

2. Identify the facts on **serverb.lab.example.com** that show the status of the system memory.

- 2.1. Use the `setup` module to get a list of all the facts for the `serverb.lab.example.com` managed host. Both the `ansible_memfree_mb` and `ansible_memtotal_mb` facts provide information about the free memory and the total memory of the managed host.

```
[student@workstation file-review]$ ansible serverb.lab.example.com -m setup
serverb.lab.example.com | SUCCESS => {
    "ansible_facts": {
...output omitted...
    "ansible_memfree_mb": 157,
...output omitted...
    "ansible_memtotal_mb": 488,
...output omitted...
    },
    "changed": false
}
```

3. Create a template for the Message of the Day, named `motd.j2`, in the current working directory. When the `devops` user logs in to `serverb.lab.example.com`, a message should display that shows the system's total memory and current free memory. Use the `ansible_memtotal_mb` and `ansible_memfree_mb` facts to provide the memory information for the message.
 - 3.1. Create a new file named `motd.j2` in the current directory. Use both the `ansible_memfree_mb` and `ansible_memtotal_mb` fact variables to create a Message of the Day.

```
[student@workstation file-review]$ cat motd.j2
This system's total memory is: {{ ansible_memtotal_mb }} MBs.
The current free memory is: {{ ansible_memfree_mb }} MBs.
```

4. Create a new playbook file called `motd.yml` in the current directory. Using the `template` module, configure the `motd.j2` Jinja2 template file previously created to map to the file `/etc/motd` on the managed hosts. This file has the `root` user as owner and group, and its permissions are `0644`. Configure the playbook so that it uses the `devops` user, and sets the `become` parameter to `true`.
 - 4.1. Create a new playbook file called `motd.yml` in the current directory. Using the `template` module, configure the `motd.j2` Jinja2 template file previously created as the value for the `src` parameter, and `/etc/motd` as the value for the `dest` parameter. Configure the `owner` and `group` parameters to `root`, and the `mode` parameter to be `0644`. Use the `devops` user for the `remote_user` parameter, and configure the `become` parameter to be `true`.

```
[student@workstation file-review]$ cat motd.yml
---
- hosts: all
  remote_user: devops
  become: true
  tasks:
    - template:
        src: motd.j2
        dest: /etc/motd
        owner: root
        group: root
```

```
mode: 0644
```

5. Run the playbook included in the **motd.yml** file.

- 5.1. Before you run the playbook, use the **ansible-playbook --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-review]$ ansible-playbook --syntax-check motd.yml
playbook: motd.yml
```

- 5.2. Run the playbook included in the **motd.yml** file.

```
[student@workstation file-review]$ ansible-playbook motd.yml
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [template] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com      : ok=2      changed=1      unreachable=0      failed=0
```

6. Check that the playbook included in the **motd.yml** file has been executed correctly.

- 6.1. Log in to `serverb.lab.example.com` as the `devops` user, and verify that the MOTD is displayed when logging in. Log off when you have finished.

```
[student@workstation file-review]$ ssh devops@serverb.lab.example.com
This system's total memory is: 488 MBs.
The current free memory is: 162 MBs.
[devops@serverb ~]$ logout
```

Evaluation

On `workstation`, run the `lab file-review grade` script to confirm success on this exercise.

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

Cleanup

On `workstation`, run the `lab file-review cleanup` script to clean up after the lab.

```
[student@workstation ~]$ lab file-review cleanup
```

SUMMARY

In this chapter, you learned:

- The `Files` modules library includes modules that allow you to accomplish most tasks related to file management, such as creating, copying, editing, and modifying permissions and other attributes of 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.
- Jinja2 filters transform template expressions from one kind or format of data into another.

CHAPTER 7

MANAGING LARGE PROJECTS

GOAL

Write playbooks that are optimized for larger, more complex projects.

OBJECTIVES

- Write sophisticated host patterns to efficiently select hosts for a play or ad hoc command.
- Describe what dynamic inventories are, and install and use an existing script as an Ansible dynamic inventory source.
- Tune the number of simultaneous connections that Ansible opens to managed hosts, and how Ansible processes groups of managed hosts through the play's tasks.
- 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)
- Managing Dynamic Inventories (and Guided Exercise)
- Configuring Parallelism (and Guided Exercise)
- Including and Importing Files (and Guided Exercise)

LAB

- Managing Large Projects

SELECTING HOSTS WITH HOST PATTERNS

OBJECTIVES

After completing this section, students should be able to write sophisticated host patterns to efficiently select hosts for a play or ad hoc command.

REFERENCING INVENTORY HOSTS

Host patterns are used to specify the hosts to target by a play or ad hoc command. 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. For an ad hoc command, the host pattern is provided as a command-line argument to the `ansible` command.

Host patterns are important to understand. 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.

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, you will execute an Ansible Playbook, `playbook.yml`, using different host patterns to target different subsets of managed hosts from this example inventory.

Managed Hosts

The most basic host pattern is the name for a single managed host listed in the inventory. This specifies that the host will be the only one in the inventory that will be acted upon by the `ansible` command.

When the playbook runs, the first **Gathering Facts** task should run on all managed hosts that match the host pattern. (After the first task, it is possible a task fails on a managed host causing it to be removed from the play.)

Remember that an IP address can be listed explicitly in the inventory instead of a host name. If it is, it can be used as a host pattern in the same way. If the IP address is not in the inventory, you cannot use it to specify the host even if the IP address resolves to that host name in the 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
---
- hosts: 192.168.2.1
...output omitted...

[student@controlnode ~]$ ansible-playbook 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 and ad hoc commands. You may find that you have to specify the host by IP address for connection purposes, however, because the host cannot have a real DNS host name for some reason.

It is possible to 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 `dummy.example`, and then direct connections using that name to the IP address 192.168.2.1 by creating a `host_vars/dummy.example` file containing the following host variable:

```
ansible_host: 192.168.2.1
```

Groups

You have also already used inventory host groups as host patterns. When a group name is used as a host pattern, it specifies that Ansible will act on the hosts that are members of the group.

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: lab
...output omitted...

[student@controlnode ~]$ ansible-playbook 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
---
- hosts: all
...output omitted...

[student@controlnode ~]$ ansible-playbook 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 matches all managed hosts in the inventory that are not members of any other group:

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: ungrouped
...output omitted...

[student@controlnode ~]$ ansible-playbook playbook.yml

PLAY [Test Host Patterns] ****

TASK [Gathering Facts] ****
ok: [web.example.com]
ok: [data.example.com]
```

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, all hosts in the inventory will match.

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: '*'
...output omitted...

[student@controlnode ~]$ ansible-playbook 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. This can be a problem when using host patterns to run ad hoc commands from the command line with `ansible`. In this case, it is a recommended practice to quote host patterns used on the command line to protect them from unwanted shell expansion.

Likewise, in an Ansible Playbook, you may need to put your host pattern in single quotes to ensure it is parsed correctly if you are using any special wildcards or list characters:

```
---
hosts: '!test1.example.com,development'
```

The asterisk character can also be used like file globbing 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
---
- hosts: '*.example.com'
...output omitted...

[student@controlnode ~]$ ansible-playbook 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
---
- hosts: '192.168.2.*'
...output omitted...

[student@controlnode ~]$ ansible-playbook 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
---
- hosts: 'datacenter*'
...output omitted...

[student@controlnode ~]$ ansible-playbook 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. This can lead to some unexpected matches if you forget this.

For example, given the example inventory, compare the results of specifying the **datacenter*** host pattern from the preceding example with the results of the **data*** host pattern:

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: 'data*'
...output omitted...

[student@controlnode ~]$ ansible-playbook 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 will be targeted:

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: labhost1.example.com,test2.example.com,192.168.2.2
...output omitted...

[student@controlnode ~]$ ansible-playbook 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 will be targeted:

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: lab,datacenter1
```

```
...output omitted...

[student@controlnode ~]$ ansible-playbook 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
---
- hosts: lab,data*,192.168.2.2
...output omitted...

[student@controlnode ~]$ ansible-playbook 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 syntax, especially when working with IPv6 addresses as managed host names. You may see the colon syntax in older 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
---
- hosts: lab,&datacenter1
...output omitted...

[student@controlnode ~]$ ansible-playbook 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, given our test inventory, matches all hosts defined in the **datacenter** group, with the exception of **test2.example.com**:

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: datacenter,!test2.example.com
...output omitted...

[student@controlnode ~]$ ansible-playbook 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 get the same effect.

The final example shows the use of a host pattern that matches all hosts in the test inventory, with the exception of the managed hosts in the **datacenter1** group.

```
[student@controlnode ~]$ cat playbook.yml
---
- hosts: all,!datacenter1
...output omitted...

[student@controlnode ~]$ ansible-playbook 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

Working with Patterns – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/intro_patterns.html

Working with Inventory – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/intro_inventory.html

► GUIDED EXERCISE

SELECTING HOSTS WITH HOST PATTERNS

In this exercise, you will explore how to use host patterns to specify hosts from the inventory for plays or ad hoc commands. You will be provided with several example inventories to explore the host patterns.

OUTCOMES

You should be able to use different host patterns to access various hosts in an inventory.

Log in to `workstation` as student using `student` as the password. Run the `lab projects-host setup` command.

```
[student@workstation ~]$ lab projects-host setup
```

The setup script confirms that Ansible is installed on `workstation` and creates a directory structure for the lab environment.

- 1. On `workstation`, change to the working directory for the exercise, `/home/student/projects-host` and review the contents of the directory.

```
[student@workstation ~]$ cd /home/student/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 example inventory file, `inventory1`. Notice how the inventory is organized. Explore which hosts are in the inventory, which domains are used, and which groups are in that inventory.

```
[student@workstation projects-host]$ cat inventory1  
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

[1b]
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 example inventory file, **inventory2**. Notice how the inventory is organized. Explore which hosts are in the inventory, which domains are used, and which groups are in that inventory.

```
[student@workstation projects-host]$ cat inventory2
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. Lastly, inspect the contents of the playbook, **playbook.yml**. Notice how the playbook uses the debug module to display the name of each managed host.

```
[student@workstation projects-host]$ cat playbook.yml
---
- name: Resolve host patterns
  hosts:
    tasks:
      - name: Display managed host name
        debug:
          msg: "{{ inventory_hostname }}"
```

- 2. Using an ad hoc command, determine if the db1.example.com server is present in the **inventory1** inventory file.

```
[student@workstation projects-host]$ ansible db1.example.com -i inventory1 \
> --list-hosts
hosts (1):
db1.example.com
```

- 3. Using an ad hoc command, reference an IP address contained in the **inventory1** inventory with a host pattern.

```
[student@workstation projects-host]$ ansible 172.25.252.44 -i inventory1 \
> --list-hosts
hosts (1):
172.25.252.44
```

- 4. With an ad hoc command, use the `all` group to list all managed hosts in the `inventory1` inventory file.

```
[student@workstation projects-host]$ ansible all -i inventory1 --list-hosts
hosts (17):
  srv1.example.com
  srv2.example.com
  s1.lab.example.com
  s2.lab.example.com
  jupiter.lab.example.com
  saturn.example.com
  db1.example.com
  db2.example.com
  db3.example.com
  lb1.lab.example.com
  lb2.lab.example.com
  file1.lab.example.com
  web1.lab.example.com
  file2.example.com
  172.25.252.23
  172.25.252.44
  172.25.252.32
```

- 5. With an ad hoc command, use the asterisk (*) character to list all hosts that end in `.example.com` in the `inventory1` inventory file.

```
[student@workstation projects-host]$ ansible '*.example.com' -i inventory1 \
> --list-hosts
hosts (14):
  jupiter.lab.example.com
  saturn.example.com
  db1.example.com
  db2.example.com
  db3.example.com
  lb1.lab.example.com
  lb2.lab.example.com
  file1.lab.example.com
  web1.lab.example.com
  file2.example.com
  srv1.example.com
  srv2.example.com
  s1.lab.example.com
  s2.lab.example.com
```

- 6. As you can see in the output of the previous command, there are 14 hosts in the `*.example.com` domain. Modify the host pattern in the previous ad hoc command so that hosts in the `*.lab.example.com` domain are ignored.

```
[student@workstation projects-host]$ ansible '*.example.com, !*.lab.example.com' \
> -i inventory1 --list-hosts
hosts (7):
  saturn.example.com
  db1.example.com
  db2.example.com
```

```
db3.example.com  
file2.example.com  
srv1.example.com  
srv2.example.com
```

- 7. Using an ad hoc command, without accessing the groups in the **inventory1** inventory file, list these three hosts: `lb1.lab.example.com`, `s1.lab.example.com`, and `db1.example.com`.

```
[student@workstation projects-host]$ ansible lb1.lab.example.com,\  
> s1.lab.example.com,db1.example.com -i inventory1 --list-hosts  
hosts (3):  
    lb1.lab.example.com  
    s1.lab.example.com  
    db1.example.com
```

- 8. Use a wildcard host pattern in an ad hoc command to list hosts that start with a **172.25.** IP address in the **inventory1** inventory file.

```
[student@workstation projects-host]$ ansible '172.25.*' -i inventory1 --list-hosts  
hosts (3):  
    172.25.252.23  
    172.25.252.44  
    172.25.252.32
```

- 9. Use a host pattern in an ad hoc command to list all hosts that start with the letter "s" in the **inventory1** inventory file.

```
[student@workstation projects-host]$ ansible 's*' -i inventory1 --list-hosts  
hosts (7):  
    saturn.example.com  
    srv1.example.com  
    srv2.example.com  
    s1.lab.example.com  
    s2.lab.example.com  
    file2.example.com  
    db2.example.com
```

Notice the `file2.example.com` and `db2.example.com` hosts in the output of the previous 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 in an ad hoc command, list all hosts in the **inventory1** inventory in the `prod` group, or with an IP address beginning with **172**, or that contain `lab` in their name.

```
[student@workstation projects-host]$ ansible 'prod,172*,*lab*' -i inventory1 \  
> --list-hosts  
hosts (11):  
    lb2.lab.example.com  
    db1.example.com  
    jupiter.lab.example.com  
    172.25.252.23
```

```
172.25.252.44  
172.25.252.32  
lb1.lab.example.com  
file1.lab.example.com  
web1.lab.example.com  
s1.lab.example.com  
s2.lab.example.com
```

- 11. Use an ad hoc command to list all hosts that belong to both the db and london groups.

```
[student@workstation projects-host]$ ansible 'db,&london' -i inventory1 \  
> --list-hosts  
hosts (2):  
  db2.example.com  
  db3.example.com
```

- 12. Modify the host pattern supplied as the value to the **hosts** keyword in the **playbook.yml** playbook so that all servers in the london group are targeted. Execute the playbook using the **inventory2** inventory file.

```
[student@workstation projects-host]$ cat playbook.yml  
...output omitted...  
hosts: london  
...output omitted...  
[student@workstation projects-host]$ ansible-playbook -i inventory2 playbook.yml  
...output omitted...  
TASK [Gathering Facts] ****  
ok: [servera.lab.example.com]  
...output omitted...
```

- 13. Modify the host pattern supplied as the value to the **hosts** keyword in the **playbook.yml** playbook so that all servers in the europe nested group are targeted. Execute the playbook using the **inventory2** inventory file.

```
[student@workstation projects-host]$ cat playbook.yml  
...output omitted...  
hosts: europe  
...output omitted...  
[student@workstation projects-host]$ ansible-playbook -i inventory2 playbook.yml  
...output omitted...  
TASK [Gathering Facts] ****  
ok: [servera.lab.example.com]  
ok: [serverb.lab.example.com]  
...output omitted...
```

- 14. Modify the host pattern supplied as the value to the **hosts** keyword in the **playbook.yml** playbook so that all servers that do not belong to any group are targeted. Execute the playbook using the **inventory2** inventory file.

```
[student@workstation projects-host]$ cat playbook.yml  
...output omitted...  
hosts: ungrouped
```

```
...output omitted...
[student@workstation projects-hosts]$ ansible-playbook -i inventory2 playbook.yml
...output omitted...
TASK [Gathering Facts] ****
ok: [workstation.lab.example.com]
...output omitted...
```

Cleanup

On workstation, run the **lab projects-host cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab projects-host cleanup
```

This concludes the guided exercise.

MANAGING DYNAMIC INVENTORIES

OBJECTIVES

After completing this section, students should be able to describe what dynamic inventories are, and install and use an existing script as an Ansible dynamic inventory source.

GENERATING INVENTORIES DYNAMICALLY

The static inventory files you have worked with so far are easy to write, and are convenient for managing small infrastructures. When working with a large number of machines, however, or in an environment where machines come and go very quickly, it can be hard to keep the static inventory files up-to-date.

Most large IT environments have systems that keep track of which hosts are available and how they are organized. For example, there might be an external directory service maintained by a monitoring system such as Zabbix, or on FreeIPA or Active Directory servers. Installation servers such as Cobbler, or management services such as Red Hat Satellite might track deployed bare-metal systems. In a similar way, cloud services such as Amazon Web Services EC2 or an OpenStack deployment, or virtual machine infrastructures based on VMware or Red Hat Virtualization might be sources of information about the instances and virtual machines that come and go.

Ansible supports *dynamic inventory* scripts that retrieve current information from these types of sources whenever Ansible executes, allowing the inventory to be updated in real time. These scripts are executable programs that collect information from some external source and output the inventory in JSON format.

Dynamic inventory scripts are used just like static inventory text files. The location of the inventory is specified either directly in the current **ansible.cfg** file, or using the **-i** option. If the inventory file is executable, it is treated as a dynamic inventory program and Ansible attempts to run it to generate the inventory. If the file is not executable, it is treated as a static inventory.



NOTE

The inventory location can be configured in the **ansible.cfg** configuration file with the **inventory** parameter. By default, it is configured to be **/etc/ansible/hosts**.

CONTRIBUTED SCRIPTS

A number of existing dynamic inventory scripts have been contributed to the Ansible project by the open source community. They are not included in the **ansible** package or officially supported by Red Hat. They are available from the Ansible GitHub site at <https://github.com/ansible/ansible/tree/devel/contrib/inventory>.

Some of the data sources or platforms that are targeted by contributed dynamic inventory scripts include:

- Private cloud platforms, such as Red Hat OpenStack Platform.
- Public cloud platforms, such as Rackspace Cloud, Amazon Web Services EC2, and Google Compute Engine.

- Virtualization platforms, such as Red Hat Virtualization (oVirt), and VMware vSphere.
- Platform-as-a-Service solutions, such as OpenShift Container Platform.
- Life-cycle management tools, such as Foreman (with Red Hat Satellite 6 or stand-alone) and Spacewalk (upstream of Red Hat Satellite 5).
- Hosting providers, such as Digital Ocean and Linode.

Each script might have its own dependencies and requirements in order to function. The contributed scripts are mostly written in Python, but that is not a requirement for dynamic inventory scripts.

WRITING DYNAMIC INVENTORY PROGRAMS

If a dynamic inventory script does not exist for the directory system or infrastructure in use, it is possible to write a custom dynamic inventory program. It can be written in any programming language, and must return inventory information in JSON format when passed appropriate options.

The **ansible-inventory** command can be a helpful tool for learning how to author Ansible inventories in JSON format. You can use the **ansible-inventory** command, available since Ansible 2.4, to view an inventory file in JSON format.

To display the contents of the inventory file in JSON format, run the **ansible-inventory --list** command. You can use the **-i** option to specify the location of the inventory file to process, or just use the default inventory set by the current Ansible configuration.

The following example demonstrates the use of the **ansible-inventory** command to process an INI-style inventory file and output it in JSON format.

```
[student@workstation projects-host]$ cat inventory
workstation1.lab.example.com

[webservers]
web1.lab.example.com
web2.lab.example.com

[databases]
db1.lab.example.com
db2.lab.example.com

[student@workstation projects-host]$ ansible-inventory -i inventory --list
{
    "_meta": {
        "hostvars": {
            "db1.lab.example.com": {},
            "db2.lab.example.com": {},
            "web1.lab.example.com": {},
            "web2.lab.example.com": {},
            "workstation1.lab.example.com": {}
        }
    },
    "all": {
        "children": [
            "databases",
            "ungrouped",
            "webservers"
        ]
    }
}
```

```
{
  "databases": {
    "hosts": [
      "db1.lab.example.com",
      "db2.lab.example.com"
    ]
  },
  "ungrouped": {
    "hosts": [
      "workstation1.lab.example.com"
    ]
  },
  "webservers": {
    "hosts": [
      "web1.lab.example.com",
      "web2.lab.example.com"
    ]
  }
}
```

If you want to write your own dynamic inventory script, more detailed information is available at [Developing Dynamic Inventory Sources](http://docs.ansible.com/ansible/dev_guide/developing_inventory.html) [http://docs.ansible.com/ansible/dev_guide/developing_inventory.html] in the *Ansible Developer Guide*. The following is a brief overview.

The script should start with an appropriate "shebang" line (for example, `#!/usr/bin/python`) and should be executable so that Ansible can run it.

When passed the `--list` option, the script must output a JSON-encoded hash/dictionary of all of the hosts and groups in the inventory to standard output.

In its simplest form, a group can be a list of managed hosts. In this example of the JSON-encoded output from an inventory script, `webservers` is a host group which has `web1.lab.example.com` and `web2.lab.example.com` as managed hosts in the group. The `databases` host group includes the `db1.lab.example.com` and `db2.lab.example.com` hosts as members.

```
[student@workstation ~]$ ./inventoryscript --list
{
  "webservers" : [ "web1.lab.example.com", "web2.lab.example.com" ],
  "databases" : [ "db1.lab.example.com", "db2.lab.example.com" ]
}
```

Alternatively, each group's value can be a JSON hash/dictionary containing a list of each managed host, any child groups, and any group variables that might be set. The next example shows the JSON-encoded output for a more complex dynamic inventory. The `boston` group has two child groups (`backup` and `ipa`), three managed hosts of its own, and a group variable set (`example_host: false`).

```
{
  "webservers" : [
    "web1.demo.example.com",
    "web2.demo.example.com"
  ],
  "boston" : {
    "children" : [
      "backup",
      "ipa"
    ],
    "vars" : {
      "example_host" : false
    }
  }
}
```

```
        "ipa"
    ],
    "vars" : {
        "example_host" : false
    },
    "hosts" : [
        "server1.demo.example.com",
        "server2.demo.example.com",
        "server3.demo.example.com"
    ]
},
"backup" : [
    "server4.demo.example.com"
],
"ipa" : [
    "server5.demo.example.com"
],
"_meta" : {
    "hostvars" : {
        "server5.demo.example.com": {
            "ntpserver": "ntp.demo.example.com",
            "dnsserver": "dns.demo.example.com"
        }
    }
}
}
```

The script should also support the **--host managed-host** option. That option may print a JSON hash/dictionary consisting of variables which should be associated with that host. If it does not, it must print an empty JSON hash/dictionary.

```
[student@workstation ~]$ ./inventoryscript --host server5.demo.example.com
{
    "ntpserver" : "ntp.demo.example.com",
    "dnsserver" : "dns.demo.example.com"
}
```



NOTE

When called with the **--host hostname** option, the script must print a JSON hash/dictionary of the variables for the specified host (potentially an empty JSON hash or dictionary if there are no variables provided).

Optionally, if the **--list** option returns a top-level element called **_meta**, it is possible to return all host variables in one script call, which improves script performance. In that case, **--host** calls are not made.

See Developing Dynamic Inventory Sources [http://docs.ansible.com/ansible/developing_inventory.html] for more information.

MANAGING MULTIPLE INVENTORIES

Ansible supports the use of multiple inventories in the same run. If the location of the inventory is a directory (whether set by the **-i** option, the value of the **inventory** parameter, or in some other way), then all inventory files included in the directory, either static or dynamic, are combined to

determine the inventory. The executable files within that directory are used to retrieve dynamic inventories, and the other files are used as static inventories.

Inventory files should not depend on other inventory files or scripts in order to resolve. For example, if a static inventory file specifies that a particular group should be a child of another group, it also needs to have a placeholder entry for that group, even if all members of that group come from the dynamic inventory. Consider the `cloud-east` group in the following example:

```
[cloud-east]  
  
[servers]  
test.demo.example.com  
  
[servers:children]  
cloud-east
```

This ensures that no matter what the order is in which inventory files are parsed, all of them are internally consistent.



NOTE

The order in which inventory files are parsed is not specified by the documentation. Currently, when multiple inventory files exist, they seem to be parsed in alphabetical order. If one inventory source depends on information from another in order to make sense, whether it works or whether it throws an error may depend on the order in which they are loaded. Therefore, it is important to make sure that all files are self-consistent to avoid unexpected errors.

Ansible ignores files in an inventory directory if they end with certain suffixes. This can be controlled with the `inventory_ignore_extensions` directive in the Ansible configuration file being used. More information is available in the Ansible documentation.



REFERENCES

Working With Dynamic Inventory: Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/intro_dynamic_inventory.html

Developing Dynamic Inventory: Ansible Documentation

https://docs.ansible.com/ansible/2.7/dev_guide/developing_inventory.html

► GUIDED EXERCISE

MANAGING DYNAMIC INVENTORIES

In this exercise, you will install custom scripts that dynamically generate a list of inventory hosts.

OUTCOMES

You should be able to install and use existing dynamic inventory scripts.

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab projects-inventory setup** script. It checks if Ansible is installed on **workstation** and also creates a working directory for this exercise.

```
[student@workstation ~]$ lab projects-inventory setup
```

- 1. On **workstation**, change to the working directory for the exercise, **/home/student/projects-inventory**.

```
[student@workstation ~]$ cd /home/student/projects-inventory
```

- 2. View the contents of the **ansible.cfg** Ansible configuration file in the working directory. The configuration file sets the inventory location to **inventory**.

```
[defaults]
inventory = inventory
```

- 3. Create the **/home/student/projects-inventory/inventory** directory.

```
[student@workstation projects-inventory]$ mkdir inventory
```

- 4. From <http://materials.example.com/labs/projects-inventory/>, download the **inventorya.py**, **inventoryw.py**, and **hosts** files to your **/home/student/projects-inventory/inventory** directory. Both of the files ending in **.py** are scripts that generate dynamic inventories, and the third file is a static inventory.

- The **inventorya.py** script provides the **webservers** group, which includes the **servera.lab.example.com** host.
- The **inventoryw.py** script provides the **workstation.lab.example.com** host.
- The **hosts** static inventory file defines the **servers** group, which is a parent group of the **webservers** group.

```
[student@workstation projects-inventory]$ wget http://materials.example.com/labs/
projects-inventory/inventorya.py -O inventory/inventorya.py
```

```
[student@workstation projects-inventory]$ wget http://materials.example.com/labs/projects-inventory/inventoryw.py -O inventory/inventoryw.py
[student@workstation projects-inventory]$ wget http://materials.example.com/labs/projects-inventory/hosts -O inventory/hosts
```

- ▶ 5. Using the **ansible** command with the **inventorya.py** script as the inventory, list the managed hosts associated with the **webservers** group. It should raise an error relating to the permissions of **inventorya.py**.

```
[student@workstation projects-inventory]$ ansible -i inventory/inventorya.py webservers --list-hosts
[WARNING]: * Failed to parse /home/student/projects-inventory/inventory/inventorya.py with script plugin: problem running /home/student/projects-inventory/inventorya.py --list ([Errno 13] Permission denied)

[WARNING]: * Failed to parse /home/student/projects-inventory/inventory/inventorya.py with ini plugin: /home/student/projects-inventory/inventory/inventorya.py:3: Expected key=value host variable assignment, got: subprocess

[WARNING]: Unable to parse /home/student/projects-inventory/inventory/inventorya.py as an inventory source

[WARNING]: No inventory was parsed, only implicit localhost is available

[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost does not match 'all'

[WARNING]: Could not match supplied host pattern, ignoring: webservers

hosts (0):
```

- ▶ 6. Check the current permissions of the **inventorya.py** script, and change them to 755.

```
[student@workstation projects-inventory]$ ls -la inventory/inventorya.py
-rw-rw-r-- 1 student student 639 Apr 29 14:20 inventory/inventorya.py
[student@workstation projects-inventory]$ chmod 755 inventory/inventorya.py
```

- ▶ 7. Change the permissions of the **inventoryw.py** script to 755.

```
[student@workstation projects-inventory]$ chmod 755 inventory/inventoryw.py
```

- ▶ 8. Check the current output of the **inventorya.py** script using the **--list** parameter. The hosts associated with the **webservers** group are displayed.

```
[student@workstation projects-inventory]$ inventory/inventorya.py --list
{"webservers": {"hosts": ["servera.lab.example.com"], "vars": {}}}
```

- ▶ 9. Check the current output of the **inventoryw.py** script using the **--list** parameter. The **workstation.lab.example.com** host is displayed.

```
[student@workstation projects-inventory]$ inventory/inventoryw.py --list
```

```
{"all": {"hosts": ["workstation.lab.example.com"], "vars": {} } }
```

- ▶ 10. Check the servers group definition in the **/home/student/projects-inventory/inventory/hosts** file. The webservers group defined in the dynamic inventory is configured as a child of the servers group.

```
[student@workstation projects-inventory]$ cat inventory/hosts
[servers:children]
webservers
```

- ▶ 11. Run the following command to verify the list of hosts in the webservers group. It raises an error about the webservers group being undefined.

```
[student@workstation projects-inventory]$ ansible webservers --list-hosts
[WARNING]: * Failed to parse /home/student/projects-inventory/inventory/hosts
with yaml plugin: Syntax Error while loading YAML.  found unexpected ':'  The
error appears to have been in '/home/student/projects-inventory/inventory/hosts':
line 1, column 9, but may be elsewhere in the file depending on the exact syntax
problem.  The offending line appears to be:  [servers:children]           ^ here

[WARNING]: * Failed to parse /home/student/projects-inventory/inventory/hosts
with ini plugin: /home/student/projects-inventory/inventory/hosts:2: Section
[servers:children] includes undefined group: webservers

[WARNING]: Unable to parse /home/student/projects-inventory/inventory/hosts as an
inventory source

hosts (1):
servera.lab.example.com
```

- ▶ 12. To make sure this problem does not happen, the static inventory should have a placeholder entry which defines an empty webservers host group. It is important for the static inventory to define any host group it references, because it is possible that it could dynamically disappear from the external source, which would cause this error.

Edit the **/home/student/projects-inventory/inventory/hosts** file so it contains the following content:

```
[webservers]

[servers:children]
webservers
```



IMPORTANT

If the dynamic inventory script that provides the host group is named so that it sorts before the static inventory referencing it, you might not see this error. However, if the host group ever disappears from the dynamic inventory, and you do not do this, the static inventory will be referencing a missing host group and the error will break the parsing of the inventory.

- 13. Rerun the following command to verify the list of hosts in the **webservers** group. It should work without any errors.

```
[student@workstation projects-inventory]$ ansible webservers --list-hosts  
hosts (1):  
servera.lab.example.com
```

Cleanup

On workstation, run the **lab projects-inventory cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab projects-inventory cleanup
```

This concludes the guided exercise.

CONFIGURING PARALLELISM

OBJECTIVES

After completing this section, students should be able to tune the number of simultaneous connections that Ansible opens to managed hosts, and how Ansible processes groups of managed hosts through the play's tasks.

CONFIGURE PARALLELISM IN ANSIBLE USING FORKS

When Ansible processes a playbook, it runs each play in order. After it has determined the list of hosts for the play, it runs through each task in order. Normally, all hosts must successfully complete a task before any host starts the next task in the play.

In theory, Ansible could simultaneously connect to all hosts in the play for each task. This works fine for small lists of hosts. But if the play targets hundreds of hosts, this can put a heavy load on the control node.

The maximum number of simultaneous connections that Ansible makes is controlled by the `forks` parameter in the Ansible configuration file. It is set to **5** by default.

```
[student@demo ~]$ grep forks ansible.cfg
forks      = 5
```

For example, assume an Ansible control node is configured with the default value of five forks and the play has ten managed hosts. Ansible will execute the first task in the play on the first five managed hosts, followed by a second round of execution of the first task on the other five managed hosts. After the first task has been executed on all the managed hosts, it will then proceed with executing the next task across all the managed hosts in groups of five hosts at a time. It will do this with each task in turn until the play ends.

The default for `forks` is set to be very conservative. If your control node is managing Linux hosts, most tasks will run on the managed hosts and the control node has less load. In this case, you can usually set `forks` to a much higher value, possibly closer to 100, and see performance improvements.

If your playbooks run a lot of code on the control node, you should raise the fork limit judiciously. This is true if you use Ansible to manage network routers and switches, because most of those modules run on the control node and not on the network device. Because of the higher load this places on the control node, its capacity to support increases in the number of forks will be significantly lower than for a control node managing only Linux hosts.

You can also override the setting for `forks` in the Ansible configuration file from the command line. Both the `ansible` and the `ansible-playbook` commands offer the `-f` or `--forks` options to specify the number of forks to use.

MANAGING ROLLING UPDATES

Normally, when Ansible runs a play, it makes sure that all managed hosts have completed each task before starting any hosts on the next task. After all managed hosts have completed all tasks, then any notified handlers are run.

This can lead to undesirable side effects, however. For example, if a play updates a cluster of load balanced web servers, it might need to take each web server out of service while the update takes place. If all the servers are updated in the same play, they could all be out of service at the same time.

One way to avoid this is to use the `serial` keyword to run the hosts through the play in batches. Each batch of hosts will be run through the entire play before the next batch is started.

In the example below, Ansible executes the play on two managed hosts at a time, until all managed hosts have been updated. Ansible begins by executing the tasks in the play on the first two managed hosts. If either or both of those two hosts notified the handler, Ansible runs the handler as needed for those two hosts. When the play execution is complete on these two managed hosts, Ansible repeats the process on the next two managed hosts. Ansible continues to run the play in this way until all managed hosts have been updated.

```
---
- name: Rolling update
  hosts: webservers
  serial: 2
  tasks:
    - name: latest apache httpd package is installed
      yum:
        name: httpd
        state: latest
      notify: restart apache

  handlers:
    - name: restart apache
      service:
        name: httpd
        state: restarted
```

Suppose the `webservers` group in the previous example contains five web servers, which reside behind a load balancer. With the `serial` parameter set to **2**, the play will only run on two web servers at a time. A majority of the five web servers will always be available.

In contrast, in the absence of the `serial` keyword, the play execution and thus handler execution would occur across all five web servers at the same time. This would probably lead to a service outage, because web services would be restarted at the same time on all of the web servers.



IMPORTANT

For certain purposes, each batch of hosts counts as if it were a full play running on a subset of hosts. This means that if an entire batch fails, the play fails, which causes the entire playbook run to fail at that point.

This is useful. In the previous scenario with `serial: 2` set, if something is wrong and the play fails for the first two hosts processed, the playbook will abort and the remaining three hosts will not be run through the play.

The `serial` keyword can also be specified as a percentage. This percentage is applied to the total number of hosts in the play to determine the rolling update batch size. Regardless of the percentage, the number of hosts per pass will always be 1 or greater.



REFERENCES

Rolling Update Batch Size – Delegation, Rolling Updates, and Local Actions – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_delegation.html#rolling-update-batch-size

Ansible Performance Tuning (For Fun and Profit)

<https://www.ansible.com/blog/ansible-performance-tuning>

► GUIDED EXERCISE

CONFIGURING PARALLELISM

In this exercise, you will explore the effects of different serial and forks directives on how a play is processed by Ansible.

OUTCOMES

You should be able to tune parallel and serial executions of a playbook across multiple managed hosts.

Log in to workstation as student using **student** as the password.

On workstation, run the **lab projects-parallelism setup** script. The setup script checks that Ansible is installed on workstation, creates the directory structure and associated files for the lab environment.

```
[student@workstation ~]$ lab projects-parallelism setup
```

- 1. On workstation, as the student user, change to the **~/projects-parallelism** directory.

```
[student@workstation ~]$ cd ~/projects-parallelism
[student@workstation projects-parallelism]$
```

- 2. Examine the contents of the project directory to become familiar with the project files.

- 2.1. Examine the contents of the **ansible.cfg** file. Note that the inventory file is set to **inventory**. Note also that the **forks** parameter is set to **4**.

```
[defaults]
inventory=inventory
remote_user=devops
forks=4
...output omitted...
```

- 2.2. Examine the contents of the **inventory** file. Note that it contains a host group, **webservers**, which contains four hosts.

```
[webservers]
servera.lab.example.com
serverb.lab.example.com
serverc.lab.example.com
```

```
serverd.lab.example.com
```

- 2.3. Examine the contents of the **playbook.yml** file. The playbook executes on the `webservers` host group. It ensures that the latest `httpd` package is installed and that the `httpd` service is enabled and started.

```
---
- name: Update web server
  hosts: webservers

  tasks:
    - name: Lastest httpd package installed
      yum:
        name: httpd
        state: latest
      notify:
        - Restart httpd

  handlers:
    - name: Restart httpd
      service:
        name: httpd
        enabled: yes
        state: restarted
```

- 2.4. Finally, examine the contents of the **remove_apache.yml** file. The playbook executes on the `webservers` host group. It ensures that the `httpd` service is disabled and stopped and then ensures that the `httpd` package is not installed.

```
---
- hosts: webservers
  tasks:
    - service:
        name: httpd
        enabled: no
        state: stopped
    - yum:
        name: httpd
        state: absent
```

- 3. Execute the **playbook.yml** playbook, using the `time` command to determine how long it takes for the playbook to run. Watch the playbook as it runs. Note how Ansible performs each task on all four hosts at the same time.

```
[student@workstation projects-parallelism]$ time ansible-playbook playbook.yml
PLAY [Update apache] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
ok: [serverd.lab.example.com]
ok: [serverb.lab.example.com]
ok: [serverc.lab.example.com]

...output omitted...
```

```
real    0m22.701s
user    0m23.275s
sys     0m2.637s
```

- ▶ 4. Execute the **remove_apache.yml** playbook to stop and disable the `httpd` service and to remove the `httpd` package.

```
[student@workstation projects-parallelism]$ ansible-playbook remove_apache.yml
```

- ▶ 5. Change the value of the forks parameter to **2** in **ansible.cfg**.

```
[defaults]
inventory=inventory
remote_user=devops
forks=2
...output omitted...
```

- ▶ 6. Re-execute the **playbook.yml** playbook, using the **time** command to determine how long it takes for the playbook to run. Watch the playbook as it runs. Note that this time Ansible performs each task on just two hosts and then the other two hosts. Also note how decreasing the number of forks caused the playbook execution to take longer than before.

```
[student@workstation projects-parallelism]$ time ansible-playbook playbook.yml

PLAY [Update apache] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]
ok: [serverc.lab.example.com]
ok: [serverd.lab.example.com]

...output omitted...

real    0m37.853s
user    0m22.414s
sys     0m4.749s
```

- ▶ 7. Execute the **remove_apache.yml** playbook to stop and disable the `httpd` service and to remove the `httpd` package.

```
[student@workstation projects-parallelism]$ ansible-playbook remove_apache.yml
```

- ▶ 8. Add the following `serial` parameter to the play in the **playbook.yml** playbook so that the play only executes on two hosts at a time. The beginning of the playbook should appear as follows:

```
---
- name: Update web server
  hosts: webservers
```

```
serial: 2
```

- ▶ 9. Re-execute the **playbook.yml** playbook. Watch the playbook as it runs. Note how Ansible executes the entire play on just two hosts before re-executing the play on the two remaining hosts.

```
[student@workstation projects-parallelism]$ ansible-playbook playbook.yml

PLAY [Update apache] ****

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

TASK [Latest version of apache installed] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

...output omitted...

PLAY [Update apache] ****

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

TASK [Latest version of apache installed] ****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

...output omitted...
```

- ▶ 10. Execute the **remove_apache.yml** playbook to stop and disable the `httpd` service and to remove the `httpd` package.

```
[student@workstation projects-parallelism]$ ansible-playbook remove_apache.yml
```

- ▶ 11. Set the **serial** parameter in the **playbook.yml** playbook to **3**. The beginning of the playbook should appear as follows:

```
---
- name: Update web server
  hosts: webservers
  serial: 3
```

- ▶ 12. Re-execute the **playbook.yml** playbook. Watch the playbook as it runs. Note how Ansible executes the entire play on just three hosts and then re-executes the play on the one remaining host.

```
[student@workstation projects-parallelism]$ ansible-playbook playbook.yml
```

```
PLAY [Update apache] ****
```

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

TASK [Latest version of apache installed] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

...output omitted...

PLAY [Update apache] ****

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

TASK [Latest version of apache installed] ****
changed: [serverd.lab.example.com]

...output omitted...
```

Cleanup

On workstation, run the **lab projects-parallelism cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab projects-parallelism cleanup
```

This concludes the guided exercise.

INCLUDING AND IMPORTING FILES

After completing this section, students should be able to 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 in a modular way, 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

There are two operations that Ansible can use to bring 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, it lets you have a master playbook that imports one or more additional playbooks into itself.

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, they will be imported and run in order.

A simple example of a master playbook that imports two additional playbooks is shown below:

```
- 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:
    - debug:
        msg: Play 1

- name: Import Playbook
```

```
import_playbook: play2.yml
```

In the preceding example, the **Play 1** runs first, and then 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
  yum:
    name: httpd
    state: latest

- name: Starts the httpd service
  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, there are some effects on how it works.

- When using the `import_tasks` feature, conditional statements such as `when` set on the import are applied to each of the tasks that are imported.
- Loops cannot be used with `import_tasks`.
- If you use a variable to specify the name of the file to import, 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. This has some impacts on how it works.

- When using the `include_tasks` feature, conditional statements such as `when` set on the `include` determine whether or not the tasks are included in the play at all.
- If you run `ansible-playbook --list-tasks` to list the tasks in the playbook, tasks in the included task files are not displayed. The tasks that include the task files are displayed. (The `import_tasks` feature, by comparison, would not list tasks that import task files but instead the individual tasks from the imported task files.)
- You cannot use `ansible-playbook --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 will run.



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, 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 systems manager.
- If a server requires a particular configuration, it can be integrated as a set of tasks executed based on a conditional (that is, including the tasks only if specific criteria are met).
- If a group of servers need to run a particular task or set of tasks, 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 while making 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 Ansible's 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 widen the scope 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
  yum:
    name: httpd
    state: latest
- name: Start the httpd service
  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 web service only.

```
---
- name: Install the {{ package }} package
  yum:
    name: "{{ package }}"
    state: latest
- name: Start the {{ service }} service
  service:
    name: "{{ service }}"
    enabled: true
    state: started
```

Subsequently, when incorporating the task file into a playbook, you 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: service
```

Ansible makes the passed variables available to the tasks imported from the external file.

You can also use the same technique to make play files more reusable. When incorporating a play file into a playbook, you 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. It 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 caused limitations in some cases. In Ansible 2.1 it became possible for `include` to be dynamic or static depending on task settings. This was confusing and error-prone. There were also issues with ensuring that `include` worked correctly in all contexts.

Because of this, `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 older playbooks, but you should avoid using it in new ones.



REFERENCES

Including and Importing – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_reuseIncludes.html

Creating Reusable Playbooks – Ansible Documentation

https://docs.ansible.com/ansible/2.7/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 will include and import playbooks and tasks in a top-level Ansible Playbook.

OUTCOMES

You should be able to include task and playbook files in playbooks.

On **workstation**, run the lab setup script to confirm the environment is ready for the lab to begin. The script creates the working directory, **/home/student/projects-file**, as well as associated project files.

```
[student@workstation ~]$ lab projects-file setup
```

- 1. On **workstation**, as the **student** user, change to the **~/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.

```
[student@workstation projects-file]$ cat tasks/environment.yml  
---  
- name: Install the {{ package }} package  
  yum:  
    name: "{{ package }}"  
    state: latest  
- name: Start the {{ service }} service  
  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.

```
[student@workstation projects-file]$ cat tasks/firewall.yml  
---  
- name: Install the firewall  
  yum:  
    name: "{{ firewall_pkg }}"  
    state: latest
```

```
- name: Start the firewall
  service:
    name: "{{ firewall_svc }}"
    enabled: true
    state: started

- name: Open the port for {{ rule }}
  firewalld:
    service: "{{ rule }}"
    immediate: true
    permanent: true
    state: enabled
```

- 2.3. Review the contents of the **tasks/placeholder.yml** file. The file contains a task for populating a placeholder web content file.

```
[student@workstation projects-file]$ cat tasks/placeholder.yml
---
- name: Create placeholder file
  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. The file contains a play which tests connections to a web service.

```
---
- name: Test web service
  hosts: localhost
  become: no
  tasks:
    - name: connect to internet web server
      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 hosts defined in the **inventory** file. The beginning of the file should look like the following:

```
---
- name: Configure web server
  hosts: servera.lab.example.com
```

- 5. In the **playbook.yml** playbook, define the tasks section with three sets of tasks. Import the first set of tasks from the **tasks/environment.yml** tasks file and define the necessary variables to install the `httpd` package and to enable and start the `httpd` service. Import the second set of tasks from the **tasks/firewall.yml** tasks file and define the necessary variables to install the `firewalld` package to enable and start the `firewalld`

service, and to allow http connections. Import the third task set 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. Import the first set of tasks from **tasks/environment.yml** using the **import_tasks** feature. Define **httpd** as the value for the package and service variables. Define **started** as the value for the **svc_state** variable.

```
- name: Import the environment task file and set the variables
  import_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. Define **firewalld** as the value for the **firewall_pkg** and **firewall_svc** variables. Define **http** as the value for the **rule** variable.

```
- name: Import the firewall task file and set the variables
  import_tasks: tasks/firewall.yml
  vars:
    firewall_pkg: firewalld
    firewall_svc: firewalld
    rule: http
```

- 5.4. Import the last task set from **tasks/placeholder.yml** using the **import_tasks** feature. Define **/var/www/html/index.html** as the value for the **file** variable.

```
- 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 and final play to the **playbook.yml** playbook using 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**. Define **http://servera.lab.example.com** as the value for the **url** variable.

```
- name: Import test play file and set the variable
  import_playbook: plays/test.yml
  vars:
    url: 'http://servera.lab.example.com'
```

- 6.2. Your playbook should look like the following after the changes are complete:

```
---
- name: Configure web server
```

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

tasks:
  - name: Import the environment task file and set the variables
    import_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
  - 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 its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step.

```
[student@workstation projects-file]$ ansible-playbook playbook.yml --syntax-check
playbook: playbook.yml
```

- 8. Execute the **playbook.yml** playbook. The output of the playbook shows the import of the task and play files.

```
[student@workstation projects-file]$ ansible-playbook playbook.yml

PLAY [Configure web server] ****
TASK [Gathering Facts] ****
ok: [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] ****
changed: [servera.lab.example.com]

TASK [Create placeholder file] ****
changed: [servera.lab.example.com]

PLAY [Test web service] ****

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

TASK [connect to internet web server] ****
ok: [localhost]

PLAY RECAP ****
localhost : ok=2    changed=0    unreachable=0    failed=0
servera.lab.example.com : ok=7    changed=1    unreachable=0    failed=0
```

Cleanup

On workstation, run the **lab projects-file cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab projects-file cleanup
```

This concludes the guided exercise.

► LAB

MANAGING LARGE PROJECTS

PERFORMANCE CHECKLIST

In this lab, you will modify a large playbook to be easier to manage by using host patterns, includes, imports, and a dynamic inventory, and you will tune how the playbook is processed by Ansible.

OUTCOMES

You should be able to simplify managed host references in a playbook by specifying host patterns against a dynamic inventory. You should also be able to restructure a playbook so that tasks are imported from external task files and tune the playbook for rolling updates.

Log in as the **student** user on **workstation** and run **lab projects-review setup**. This set-up script ensures that the managed hosts are reachable on the network. It also ensures that the correct Ansible configuration file, inventory file, and playbook are installed on the control node.

```
[student@workstation ~]$ lab projects-review setup
```

You have inherited a playbook from the previous administrator. The playbook is used to configure 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 and also tune it so that future executions use rolling updates to prevent all four web servers from being unavailable at the same time.

1. Simplify the list of managed hosts in the playbook by using a wildcard host pattern.
2. Restructure the playbook so that the first two tasks in the playbook 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 playbook.
3. Restructure the playbook so that the third, fourth, and fifth tasks in the playbook 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 playbook.
4. Because the handler for restarting the **httpd** service could be triggered if there are future changes to the **files/tune.conf** file, implement the rolling update feature in the **playbook.yml** playbook and set the rolling update batch size to be two hosts.
5. Verify the changes to the **playbook.yml** playbook were correctly made and then execute the playbook.

Evaluation

Run the **lab projects-review grade** command from **workstation** to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

Cleanup

On workstation, run the **lab projects-review cleanup** script to clean up the resources created in this lab.

```
[student@workstation ~]$ lab projects-review cleanup
```

This concludes the lab.

► SOLUTION

MANAGING LARGE PROJECTS

PERFORMANCE CHECKLIST

In this lab, you will modify a large playbook to be easier to manage by using host patterns, includes, imports, and a dynamic inventory, and you will tune how the playbook is processed by Ansible.

OUTCOMES

You should be able to simplify managed host references in a playbook by specifying host patterns against a dynamic inventory. You should also be able to restructure a playbook so that tasks are imported from external task files and tune the playbook for rolling updates.

Log in as the **student** user on **workstation** and run **lab projects-review setup**. This set-up script ensures that the managed hosts are reachable on the network. It also ensures that the correct Ansible configuration file, inventory file, and playbook are installed on the control node.

```
[student@workstation ~]$ lab projects-review setup
```

You have inherited a playbook from the previous administrator. The playbook is used to configure 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 and also tune it so that future executions use rolling updates to prevent all four web servers from being unavailable at the same time.

1. Simplify the list of managed hosts in the playbook by using a wildcard host pattern.

- 1.1. Review the **ansible.cfg** configuration file to determine the location of the inventory file. You should see that the inventory is defined as the **inventory** subdirectory and that this subdirectory contains an **inventory.py** dynamic inventory script.

```
[student@workstation ~]$ cd ~/projects-review
[student@workstation projects-review]$ cat ansible.cfg
[defaults]
inventory = inventory
...output omitted...
[student@workstation projects-review]$ ll
total 16
-rw-rw-r--. 1 student student 33 Dec 19 00:48 ansible.cfg
drwxrwxr-x. 2 student student 4096 Dec 18 22:35 files
drwxrwxr-x. 2 student student 4096 Dec 19 01:18 inventory
-rw-rw-r--. 1 student student 959 Dec 18 23:48 playbook.yml
[student@workstation projects-review]$ ll inventory
total 4
```

```
-rwxrwxr-x. 1 student student 612 Dec 19 01:18 inventory.py
```

- 1.2. Make the **inventory/inventory.py** dynamic inventory script executable, and then execute the dynamic inventory script with the **--list** option to display the full list of hosts in the inventory.

```
[student@workstation projects-review]$ chmod 755 inventory/inventory.py
[student@workstation projects-review]$ inventory/inventory.py --list
{"all": {"hosts": ["servera.lab.example.com", "serverb.lab.example.com",
"serverc.lab.example.com", "serverd.lab.example.com",
"workstation.lab.example.com"], "vars": {}}}
```

- 1.3. Verify that the host pattern **server*.lab.example.com** correctly identifies the four managed hosts that are targeted by the **playbook.yml** playbook.

```
[student@workstation projects-review]$ ansible server*.lab.example.com --list-hosts
hosts
  hosts (4):
    serverb.lab.example.com
    serverd.lab.example.com
    servera.lab.example.com
    serverc.lab.example.com
```

- 1.4. Replace the host list in the **playbook.yml** playbook with the **server*.lab.example.com** host pattern.

```
[student@workstation projects-review]$ cat playbook.yml
---
- name: Install and configure web service
  hosts: server*.lab.example.com
...output omitted...
```

2. Restructure the playbook so that the first two tasks in the playbook 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 playbook.

- 2.1. Create the **tasks** subdirectory.

```
[student@workstation projects-review]$ mkdir tasks
```

- 2.2. Place the contents of the first two tasks in the **playbook.yml** playbook into the **tasks/web_tasks.yml** file. The task file should contain the following content:

```
---
- name: Install httpd
  yum:
    name: httpd
    state: latest

- name: Tuning configuration installed
  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 two tasks from the **playbook.yml** playbook and 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 so that the third, fourth, and fifth tasks in the playbook 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 playbook.
- 3.1. Place the contents of the three remaining tasks in the **playbook.yml** playbook into the **tasks/firewall_tasks.yml** file. The task file should contain the following content.

```
---
- name: Install firewalld
yum:
  name: firewalld
  state: latest

- name: Start the firewall
service:
  name: firewalld
  enabled: true
  state: started

- name: Open the port for http
firewalld:
  service: http
  immediate: true
  permanent: true
  state: enabled
```

- 3.2. Remove the remaining three tasks from the **playbook.yml** playbook and put the following lines in their place to import the **tasks/firewall_tasks.yml** task file.

```
- name: Import the firewall_tasks.yml task file
import_tasks: tasks/firewall_tasks.yml
```

4. Because the handler for restarting the httpd service could be triggered if there are future changes to the **files/tune.conf** file, implement the rolling update feature in the **playbook.yml** playbook and set the rolling update batch size to be two hosts.

- 4.1. Add the **serial** parameter to the **playbook.yml** playbook.

```
[student@workstation projects-review]$ cat playbook.yml
---
- name: Install and configure web service
  hosts: server*.lab.example.com
```

```
serial: 2
...output omitted...
```

5. Verify the changes to the **playbook.yml** playbook were correctly made and then execute 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
  serial: 2

  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
      service:
        name: httpd
        state: restarted
```

- 5.2. Execute the playbook with **ansible-playbook --syntax-check** to verify the playbook contains no syntax errors. If any errors are present, correct them before proceeding.

```
[student@workstation projects-review]$ ansible-playbook playbook.yml --syntax-check
playbook: playbook.yml
```

- 5.3. Execute the playbook. The playbook should execute against the host as a rolling update with a batch size of two managed hosts.

```
[student@workstation projects-review]$ ansible-playbook playbook.yml

PLAY [Install and configure web service] ****
TASK [Gathering Facts] ****
ok: [serverd.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Install httpd] ****
changed: [serverd.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Tuning configuration installed] ****
changed: [serverb.lab.example.com]
changed: [serverd.lab.example.com]

TASK [Install firewalld] ****
ok: [serverb.lab.example.com]
ok: [serverd.lab.example.com]
```

```
TASK [Start the firewall] *****
ok: [serverd.lab.example.com]
ok: [serverb.lab.example.com]

TASK [Open the port for http] *****
changed: [serverd.lab.example.com]
changed: [serverb.lab.example.com]

RUNNING HANDLER [restart httpd] *****
changed: [serverb.lab.example.com]
changed: [serverd.lab.example.com]

PLAY [Install and configure web service] *****

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

TASK [Install httpd] *****
changed: [serverc.lab.example.com]
changed: [servera.lab.example.com]

TASK [Tuning configuration installed] *****
changed: [servera.lab.example.com]
changed: [serverc.lab.example.com]

TASK [Install firewalld] *****
ok: [servera.lab.example.com]
ok: [serverc.lab.example.com]

TASK [Start the firewall] *****
ok: [servera.lab.example.com]
ok: [serverc.lab.example.com]

TASK [Open the port for http] *****
changed: [servera.lab.example.com]
changed: [serverc.lab.example.com]

RUNNING HANDLER [restart httpd] *****
changed: [serverc.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com    : ok=7      changed=2      unreachable=0      failed=0
serverb.lab.example.com    : ok=7      changed=3      unreachable=0      failed=0
serverc.lab.example.com    : ok=7      changed=4      unreachable=0      failed=0
serverd.lab.example.com    : ok=7      changed=4      unreachable=0      failed=0
```

Evaluation

Run the **lab projects-review grade** command from *workstation* to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

Cleanup

On workstation, run the **lab projects-review cleanup** script to clean up the resources created in this lab.

```
[student@workstation ~]$ lab projects-review cleanup
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Host patterns are used to specify the managed hosts to be targeted by plays or ad hoc commands.
- Dynamic inventory scripts can be used to generate dynamic lists of managed hosts from directory services or other sources external to Ansible.
- Dynamic inventory scripts must be executable and must return inventory information in JSON format when passed the **--list** option.
- The **forks** parameter in the Ansible configuration file sets the maximum number of parallel connections to managed hosts.
- The **serial** parameter can be used to implement rolling updates across managed hosts by defining the number of managed hosts in each rolling update batch.
- 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.
- Import features are static in their operation, and take effect when the playbook is parsed. Include features are dynamic, and take effect when that part of the playbook is run.

CHAPTER 8

SIMPLIFYING PLAYBOOKS WITH ROLES

GOAL

Use Ansible roles to develop playbooks more quickly and to reuse Ansible code.

OBJECTIVES

- Describe what a role is, how it is structured, and how you can use it in a playbook.
- 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 Ansible Galaxy or other sources, such as a Git repository, and use them in your playbooks.
- Write playbooks that take advantage of Red Hat Enterprise Linux System Roles to perform standard operations.

SECTIONS

- Describing Role Structure (and Quiz)
- Creating Roles (and Guided Exercise)
- Deploying Roles with Ansible Galaxy (and Guided Exercise)
- Reusing Content with System Roles (and Guided Exercise)

LAB

- Simplifying Playbooks with Roles

DESCRIBING ROLE STRUCTURE

OBJECTIVES

After completing this section, students should be able to describe what a role is, how it is structured, and how you can use it in a playbook.

STRUCTURING ANSIBLE PLAYBOOKS WITH ROLES

As you develop more playbooks, you will probably discover that you have many opportunities to reuse code from playbooks that you have already written. Perhaps a play to configure a MySQL database for one application could be repurposed, with different hostnames, passwords, and users, to configure a MySQL database for another application.

But in the real world, that play might be long and complex, with many included or imported files, and with tasks and handlers to manage various situations. Copying all that code into another playbook might be non-trivial work.

Ansible *roles* provide a way for you to make it easier to reuse Ansible code in a generic way. You can package, in a standardized directory structure, all the tasks, variables, files, templates, and other resources needed to provision infrastructure or deploy applications. You can copy that role from project to project simply by copying that role's directory. You can then simply call that role from a play to execute it.

A well-written role will allow you to pass variables to the role from the playbook that adjust its behavior, setting all the site-specific hostnames, IP addresses, user names, secrets, or other locally-specific details you need. For example, a role to deploy a database server might have been written to support variables which set the hostname, database admin user and password, and other parameters that need customization for your installation. The author of the role can also ensure that reasonable default values are set for those variables if you choose not to set them in the play.

Ansible roles have the following benefits:

- Roles group content, allowing easy sharing of code with others
- Roles can be written that define the essential elements of a system type: web server, database server, Git repository, or other purpose
- Roles make larger projects more manageable
- Roles can be developed in parallel by different administrators

In addition to writing, using, reusing, and sharing your own roles, you can get roles from other sources. Some roles are included as part of Red Hat Enterprise Linux, in the *rhel-system-roles* package. You can also get a large number of community-supported roles from the Ansible Galaxy web site. Later in this chapter, you will learn more about these roles.

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 YAML 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
defaults	The main.yml 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.
files	This directory contains static files that are referenced by role tasks.
handlers	The main.yml file in this directory contains the role's handler definitions.
meta	The main.yml file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
tasks	The main.yml file in this directory contains the role's task definitions.
templates	This directory contains Jinja2 templates that are referenced by role tasks.
tests	This directory can contain an inventory and test.yml playbook that can be used to test the role.
vars	The main.yml 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 will have 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. They are referenced in the role YAML file like any other variable: **{{ VAR_NAME }}**. These variables have a high precedence and can not be overridden by

inventory variables. The intent of these variables is that they are used by the internal functioning of the role.

Default variables allow default values to be set for variables that can be used in a play to configure the role or customize its behavior. They 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 variables available. They can be easily 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. They can be used 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. Default variables should be used when it is intended that their values will be overridden.



IMPORTANT

Roles should not have site-specific data in them. They definitely should not contain any secrets like passwords or private keys.

This is because roles are supposed to be generic, reusable, and freely shareable. Site-specific details should not be hard coded into them.

Secrets should be provided to the role through other means. This is one reason 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 PLAYBOOK

Using roles in a playbook is straightforward. The following example shows one way to call Ansible roles.

```
---  
- hosts: remote.example.com  
  roles:  
    - role1  
    - role2
```

For each role specified, the role tasks, role handlers, role variables, and role dependencies will be imported into the playbook, in that order. Any `copy`, `script`, `template`, or `include_tasks/import_tasks` tasks in the role can reference the relevant files, templates, or task files in the role without absolute or relative path names. Ansible looks for them in the role's **files**, **templates**, or **tasks** subdirectories respectively.

When you use a **roles** section to import roles into a play, the roles will run first, before any tasks that you define for that play.

The following example sets values for two role variables of **role2**. **role1** is used in the same way as the previous example. Any **defaults** and **vars** variables are overridden when **role2** is used.

```
---  
- hosts: remote.example.com  
  roles:  
    - role: role1  
    - role: role2  
      var1: val1
```

```
var2: val2
```

Another equivalent YAML syntax which you might see in this case is:

```
---
- 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

Role variables set inline (role parameters), as in the preceding examples, have very high precedence. They will override most other variables.

Be very careful not to reuse the names of any role variables that you set inline anywhere else in your play, since the values of the role variables will override inventory variables and any play **vars**.

CONTROLLING ORDER OF EXECUTION

For each play in a playbook, tasks execute as ordered in the tasks list. After all tasks execute, any notified handlers are executed.

When a role is added to a play, role tasks are added to the beginning of the tasks list. If a second role is included in a play, its tasks list is added after the first role.

Role handlers are added to plays in the same manner that role tasks are added to plays. Each play defines a handlers list. Role handlers are added to the handlers list first, followed by any handlers defined in the **handlers** section of the play.

In certain scenarios, it may be necessary to execute some play tasks before the roles. To support such scenarios, plays can be configured with a **pre_tasks** section. Any task listed in this section executes before any roles are executed. If any of these tasks notify a handler, those handler tasks execute before the roles or normal tasks.

Plays also support a **post_tasks** keyword. These tasks execute after the play's normal tasks, and any handlers they notify, are run.

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:
    - debug:
        msg: 'pre-task'
        notify: my handler
  roles:
    - role1
  tasks:
    - debug:
        msg: 'first task'
        notify: my handler
```

```
post_tasks:
  - debug:
    msg: 'post-task'
    notify: my handler
handlers:
  - name: my handler
    debug:
      msg: Running my handler
```

In the above example, a debug task executes in each section to notify the **my handler** handler. The **my handler** task is executed three times:

- after all the **pre_tasks** tasks execute
- after all role tasks and tasks from the **tasks** section execute
- after all the **post_tasks** execute

Roles can be added to play using an ordinary task, not just by including them in the **roles** section of a play. Use the `include_role` module to dynamically include a role, and use the `import_role` module to statically import a role.

The following playbook demonstrates how a role can be included using a task with the `include_role` module.

```
- name: Execute a role as a task
hosts: remote.example.com
tasks:
  - name: A normal task
    debug:
      msg: 'first task'
  - name: A task to include role2 here
    include_role: role2
```



NOTE

The `include_role` module was added in Ansible 2.3, and the `import_role` module in Ansible 2.4.



REFERENCES

Roles – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_reuse_roles.html

► QUIZ

DESCRIBING ROLE STRUCTURE

Choose the correct answer to the following questions:

► 1. **Which 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. Collection of YAML task files and supporting items arranged in a specific structure for easy sharing, portability, and reuse.

► 2. **Which of the following can be specified in roles?**

- a. Handlers
- b. Tasks
- c. Templates
- d. Variables
- e. All of the above

► 3. **Which file declares role dependencies?**

- a. The Ansible Playbook that uses the role.
- b. The `meta/main.yml` file inside the role hierarchy.
- c. The `meta/main.yml` file in the project directory.
- d. Role dependencies cannot be defined in Ansible.

► 4. **Which file in a role's directory hierarchy should contain the initial 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 answer to the following questions:

► 1. Which 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. Collection of YAML task files and supporting items arranged in a specific structure for easy sharing, portability, and reuse.

► 2. Which of the following can be specified in roles?

- a. Handlers
- b. Tasks
- c. Templates
- d. Variables
- e. All of the above

► 3. Which file declares role dependencies?

- a. The Ansible Playbook that uses the role.
- b. The `meta/main.yml` file inside the role hierarchy.
- c. The `meta/main.yml` file in the project directory.
- d. Role dependencies cannot be defined in Ansible.

► 4. Which file in a role's directory hierarchy should contain the initial 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

After completing this section, students should be able to 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 requires no 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.

Figure 8.0: Creating roles

CREATING THE ROLE DIRECTORY STRUCTURE

By default, Ansible looks for roles in a subdirectory called **roles** in the directory containing your Ansible Playbook. This allows you to store roles with the playbook and other supporting files.

If Ansible cannot find the role there, it looks at the directories specified by the Ansible configuration setting **roles_path**, in order. This variable contains a colon-separated list of directories to search. The default value of this variable is:

```
~/ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles
```

This allows you to install roles on your system that are shared by multiple projects. For example, you could have your own roles installed your home directory in the **~/ansible/roles** subdirectory, and the system can have roles installed for all users in the **/usr/share/ansible/roles** directory.

Each role has its own directory with a standardized directory structure. 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 and examples of how to use it, and any non-Ansible requirements it might have in order to work. 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 can be deployed by the role when it is used. The other subdirectories can contain **main.yml** files that define default variable values, handlers, tasks, role metadata, or variables, depending on the subdirectory they are in.

If a subdirectory exists but is empty, such as **handlers** in this example, it is ignored. If a role does not use a feature, the subdirectory can be omitted altogether. For example, the **vars** subdirectory has been omitted from this example.

Creating a Role Skeleton

You can create all of the subdirectories and files needed for a new role 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 name of the role 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
- 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

Once you have created the directory structure, you must write the content of the role. A good place to start is the **ROLENAME/tasks/main.yml** task file, the main list of tasks run by the role.

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
  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 will be the email address that is written in the `/etc/motd` file of managed hosts that this role is applied to.

```
[user@host ~]$ cat roles/motd/defaults/main.yml
---
system_owner: user@host.example.com
```

Recommended Practices for Role Content Development

Roles allow playbooks to be written in a modular fashion. 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.
- Sensitive information, such as passwords or SSH keys, should not be stored in the role repository. Sensitive values should be parameterized as variables with default values that are not sensitive. Playbooks that use the role are responsible for defining sensitive variables through Ansible Vault variable files, environment variables, or other `ansible-playbook` options.
- Use `ansible-galaxy init` to start your role, and then remove any directories and files that you do not need.
- Create and maintain `README.md` and `meta/main.yml` files to document what your role is for, who wrote it, and how to use it.
- Keep your role focused on a specific purpose or function. Instead of making one role do many things, you might write more than one role.
- Reuse and refactor roles often. Resist creating new roles for edge configurations. If an existing role accomplishes a majority 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 also does not cause problems for existing playbooks.

DEFINING ROLE DEPENDENCIES

Role dependencies allow a role to include other roles as dependencies. For example, a role that defines a documentation server may 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
```

By default, roles are only added as a dependency to a playbook once. If another role also lists it as a dependency it will not be run again. This behavior can be overridden by setting the **allow_duplicates** variable to **yes** in the **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.

USING THE ROLE IN A PLAYBOOK

To access a role, reference it in the **roles:** section of a playbook. The following playbook refers to the **motd** role. Because no variables are specified, the role is applied with its default variable values.

```
[user@host ~]$ cat use-motd-role.yml
---
- name: use motd role playbook
  hosts: remote.example.com
  user: devops
  become: true
  roles:
    - motd
```

When the playbook is executed, tasks performed because of a role can be identified by the role name prefix. The following sample output illustrates this with the **motd** : prefix in the task name:

```
[user@host ~]$ ansible-playbook -i inventory use-motd-role.yml

PLAY [use motd role playbook] ****
TASK [setup] ****
ok: [remote.example.com]

TASK [motd: deliver motd file] ****
changed: [remote.example.com]

PLAY RECAP ****
remote.example.com      : ok=2    changed=1    unreachable=0    failed=0
```

The above scenario assumes that the **motd** role is located in the **roles** directory. Later in the course you will see how to use a role that is remotely located in a version control repository.

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. This helps make the role more generic and reusable in a variety of contexts.

The value of any variable defined in a role's **defaults** directory will be 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**, will replace 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
  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 will 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**, will replace 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
  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.

- Almost any other variable will 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 are some of the variables that can do that. Inventory variables and play **vars** cannot. This is important because it helps keep your play from accidentally messing up the internal functioning of the role.
- However, variables declared inline as role parameters, like the last of the preceding examples, have very high precedence. They can 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.

**REFERENCES****Using Roles – Ansible Documentation**

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_reuse_roles.html#using-roles

Using Variables – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html

► GUIDED EXERCISE

CREATING ROLES

In this exercise, you will create an Ansible role that uses variables, files, templates, tasks, and handlers to deploy a network service, and enable a working firewall.

OUTCOMES

You should be able to create two roles that use variables and parameters: **myvhost** and **myfirewall**.

The **myvhost** role installs and configures the Apache service on a host. A template is provided that will be used for **/etc/httpd/conf.d/vhost.conf: vhost.conf.j2**.

The **myfirewall** role installs, enables, and starts the **firewalld** daemon. It opens the firewall service port specified by the **firewall_service** variable.

From workstation, run the command **lab role-create setup** to prepare the environment for this exercise. This creates the **role-create** working directory, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab role-create setup
```

- 1. Log in to your workstation host as **student**. Change to the **role-create** working 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.

```
[student@workstation role-create]$ mkdir -v roles; cd roles  
mkdir: created directory 'roles'  
[student@workstation roles]$ ansible-galaxy init myvhost  
- myvhost was created successfully  
[student@workstation roles]$ rm -rvf 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 four tasks to ensure:

- The *httpd* package is installed
- The *httpd* service is started and enabled
- The web server configuration file is installed, using a template provided by the role
- The HTML content is downloaded into the virtual host's **DocumentRoot** directory, as specified in the configuration file

- 3.1. Edit the **roles/myvhost/tasks/main.yml** file. Include code to use the **yum** module to install the *httpd* package. The file contents should look like the following:

```
---
```

```
# tasks file for myvhost
```

```
- name: Ensure httpd is installed
  yum:
    name: httpd
    state: latest
```

- 3.2. Add additional code to the **tasks/main.yml** file to use the **service** module to start and enable the *httpd* service.

```
- name: Ensure httpd is started and enabled
  service:
    name: httpd
    state: started
    enabled: true
```

- 3.3. Add another stanza to use the **template** module to create **/etc/httpd/conf.d/vhost.conf** on the managed host. It should call a handler to restart the *httpd* daemon when this file is updated.

```
- name: vhost file is installed
  template:
    src: vhost.conf.j2
    dest: /etc/httpd/conf.d/vhost.conf
    owner: root
    group: root
    mode: 0644
  notify:
    - restart httpd
```

- 3.4. Add another stanza to copy the HTML content from the role to the virtual host **DocumentRoot** directory. Use the **copy** module and include a trailing slash after the source directory name. This causes the module to copy the contents of the **html** directory immediately below the destination directory (similar to **rsync** usage). The **ansible_hostname** variable expands to the short host name of the managed host.

```
- name: HTML content is installed
  copy:
    src: html/
```

```
dest: "/var/www/vhosts/{{ ansible_hostname }}"
```

3.5. Save your changes and exit the **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 **service** module. The file contents should look like the following:

```
---
```

```
# handlers file for myvhost
```

```
- name: restart httpd
  service:
    name: httpd
    state: restarted
```

- 5. Create the HTML content to be served by the web server.

5.1. The role task that called the **copy** module referred to an **html** directory as the **src**. Create this directory below the **files** subdirectory of the role.

```
[student@workstation role-create]$ mkdir -pv roles/myvhost/files/html
mkdir: created directory 'roles/myvhost/files/html'
```

5.2. Create an **index.html** file below that directory with the contents: “simple index”. Be sure to use this string verbatim because the grading script looks for it.

```
[student@workstation role-create]$ echo \
> 'simple index' > roles/myvhost/files/html/index.html
```

- 6. 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/
'vhost.conf.j2' -> 'roles/myvhost/templates/vhost.conf.j2'
```

- 7. Test the **myvhost** role to make sure it works properly.

7.1. Write a playbook that uses the role, called **use-vhost-role.yml**. It should have the following content:

```
---
```

```
- name: Use myvhost role playbook
  hosts: webservers
  pre_tasks:
    - name: pre_tasks message
      debug:
        msg: 'Ensure web server configuration.'
```

```
  roles:
    - myvhost
```

```
post_tasks:
  - name: post_tasks message
    debug:
      msg: 'Web server is configured.'
```

- 7.2. Before running the playbook, verify that its syntax is correct by running **ansible-playbook** with the **--syntax-check**. 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-playbook use-vhost-role.yml \
> --syntax-check

playbook: use-vhost-role.yml
```

- 7.3. Run the playbook. Review the output to confirm that Ansible performed the actions on the web server, `servera`.

```
[student@workstation role-create]$ ansible-playbook 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]

TASK [myvhost : HTML content is installed] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [myvhost : restart httpd] ****
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
```

- 7.4. Run ad hoc commands to confirm that the role worked. The *httpd* package should be installed and the *httpd* service should be running.

```
[student@workstation role-create]$ ansible webservers -a \
> 'systemctl is-active httpd'
servera.lab.example.com | CHANGED | rc=0 >>
active

[student@workstation role-create]$ ansible webservers -a \
> 'systemctl is-enabled httpd'
servera.lab.example.com | CHANGED | rc=0 >>
enabled
```

- 7.5. The Apache configuration should be installed with template variables expanded.

```
[student@workstation role-create]$ ansible webservers -a \
> 'cat /etc/httpd/conf.d/vhost.conf'
servera.lab.example.com | CHANGED | rc=0 >>
# Ansible managed:

<VirtualHost *:80>
    ServerAdmin webmaster@servera.lab.example.com
    ServerName 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/>
        Options +Indexes +FollowSymlinks +Includes
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

- 7.6. The HTML content should be found in a directory called **/var/www/vhosts/servera**. The **index.html** file should contain the string “simple index”.

```
[student@workstation role-create]$ ansible webservers -a \
> 'cat /var/www/vhosts/servera/index.html'
servera.lab.example.com | CHANGED | rc=0 >>
simple index
```

- 7.7. Use a the *uri* module in an ad hoc command to check that the web content is available locally. Set the *return_content* parameter to **true** to have the content of the server's response added to the output. The server content should be the string **"simple index\n"**.

```
[student@workstation role-create]$ ansible webservers -m uri \
> -a 'url=http://localhost return_content=true'
servera.lab.example.com | SUCCESS => {
    "accept_ranges": "bytes",
```

```
"changed": false,  
"connection": "close",  
"content": "simple index\\n",  
...output omitted...  
"status": 200,  
"url": "http://localhost"  
}
```

- 7.8. Use a web browser on `workstation` to check if content is available from `http://servera.lab.example.com`. If a firewall is running on `servera`, you see the following error message:

```
[student@workstation role-create]$ curl -S http://servera.lab.example.com  
curl: (7) Failed connect to servera.lab.example.com:80; No route to host
```

This is because the firewall port for HTTP is not open. If the web content successfully displays, it is because a firewall is not running on `servera`.

- 8. Use the `ansible-galaxy init` command to create the role directory structure for a role called `myfirewall`.

```
[student@workstation role-create]$ cd roles/  
[student@workstation roles]$ ansible-galaxy init myfirewall  
- myfirewall was created successfully  
[student@workstation roles]$ cd ..  
[student@workstation role-create]$
```

- 9. Edit the `main.yml` file in the `tasks` subdirectory of the `myfirewall` role. The role should perform three tasks:

- Install the `firewalld` package.
- Start and enable the `firewalld` service.
- Open a firewall service port.

- 9.1. Use a text editor to create a file called `roles/myfirewall/tasks/main.yml`. Include code to use the `yum` module to install the `firewalld` package. The file contents should look like the following:

```
---  
# tasks file for myfirewall  
  
- name: Ensure firewalld is installed  
  yum:  
    name: firewalld  
    state: latest
```

- 9.2. Add additional code to the `tasks/main.yml` file to use the `service` module to start and enable the `firewalld` service.

```
- name: Ensure firewalld is started and enabled  
  service:
```

```
name: firewalld
state: started
enabled: true
```

- 9.3. Add another task that uses the **firewalld** module to immediately, and persistently, open the service port for all services listed in the **firewall_services** variable. It should look like the following:

```
- name: Ensure firewalld services are enabled
  firewalld:
    state: enabled
    immediate: true
    permanent: true
    service: "{{ item }}"
  loop: "{{ firewall_services }}"
```

- 9.4. Save your changes and exit the **tasks/main.yml** file.

- 10. Create the file that defines the default value for the **firewall_services** variable. It should be structured as an empty list. This causes the role, by default, to not open any ports.

So that others know how to use this variable in their own playbook projects, provide an example definition of the **firewall_services** variable with at least one entry. Be sure that this definition is commented out.

You will override this variable to open the port for the HTTP protocol when you use the role in a later step.

Use a text editor to create a file called **roles/myfirewall/defaults/main.yml** containing the following content:

```
---
# defaults file for myfirewall

# By default, no firewall services are enabled.
#firewall_services: []
#
#To enable, for example, FTP and HTTP services in firewalld,
# use the following definition for "firewall_services"
#firewall_services:
#  - http
#  - ftp
firewall_services: []
```

- 11. Modify the **myvhost** role to include the **myfirewall** role as a dependency, then retest the modified role.

Use a text editor to modify the **roles/myvhost/meta/main.yml** file to make the **myvhost** role depend on the **myfirewall** role. The dependencies variable should be defined as the following:

```
dependencies:
  - role: myfirewall
```

Optionally, update the author, company, description, and license fields of the **galaxy_info** variable. Save the **roles/myvhost/meta/main.yml** file.

- 12. Create a YAML file under the **group_vars** directory to override the **firewall_services** variable for all **webserver** hosts.

The **firewall_services** variable should contain one service, **ssh**. Ports for the HTTP protocols will be opened by the **myfirewall** role.

- 12.1. Create a directory **group_vars/webservers** to hold YAML files that define variables for the **webservers** host group.

```
[student@workstation role-create]$ mkdir -pv group_vars/webservers  
mkdir: created directory 'group_vars'  
mkdir: created directory 'group_vars/webservers'
```

- 12.2. Copy the **defaults/main.yml** file for the **myfirewall** role to the **group_vars/webservers** directory. Name the file after the role, **myfirewall.yml**.

```
[student@workstation role-create]$ cp -v \  
> roles/myfirewall/defaults/main.yml group_vars/webservers/myfirewall.yml  
'roles/myfirewall/defaults/main.yml' -> 'group_vars/webservers/myfirewall.yml'
```

- 12.3. Update the **group_vars/webservers/myfirewall.yml** file with the desired values for the **firewall_services** variable. Also, remove the **#** comment from the file. The resulting file should look like the following:

```
---  
  
firewall_services:  
  - http
```

- 13. Run the playbook again. Confirm the additional **myfirewall** tasks are successfully executed.

```
[student@workstation role-create]$ ansible-playbook 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 [myfirewall : Ensure firewalld is installed] ****  
ok: [servera.lab.example.com]  
  
TASK [myfirewall : Ensure firewalld is started and enabled] ****  
ok: [servera.lab.example.com]  
  
TASK [myfirewall : Ensure firewalld services are enabled] ****  
changed: [servera.lab.example.com] => (item=http)  
  
TASK [myvhost : Ensure httpd is installed] ****
```

```
ok: [servera.lab.example.com]

TASK [myvhost : Ensure httpd is started and enabled] ****
ok: [servera.lab.example.com]

TASK [myvhost : vhost file is installed] ****
ok: [servera.lab.example.com]

TASK [myvhost : HTML content is installed] ****
ok: [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=10    changed=1    unreachable=0    failed=0
```

- 14. Confirm that the web server content is available to remote clients.

```
[student@workstation role-create]$ curl http://servera.lab.example.com
simple index
```

Cleanup

Run the **lab role-create cleanup** command to cleanup the managed host.

```
[student@workstation ~]$ lab role-create cleanup
```

This concludes the guided exercise.

DEPLOYING ROLES WITH ANSIBLE GALAXY

OBJECTIVES

After completing this section, students should be able to select and retrieve roles from Ansible Galaxy or other sources such as a Git repository, and use them in playbooks.

ANSIBLE GALAXY

Ansible Galaxy [<https://galaxy.ansible.com>] is a public library of Ansible content written by a variety of Ansible administrators and users. It contains thousands of Ansible roles and it has a searchable database that helps Ansible users identify roles that might help them accomplish an administrative task. Ansible Galaxy includes links to documentation and videos for new Ansible users and role developers.

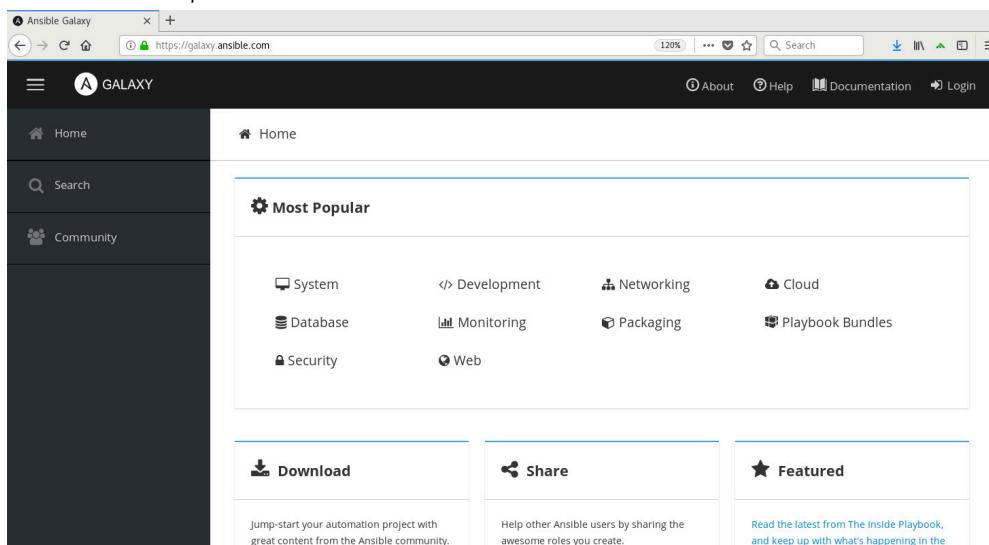


Figure 8.1: Ansible Galaxy home page

In addition, the **ansible-galaxy** command that you use to get and manage roles from Ansible Galaxy can also be used to get and manage roles your projects need from your own Git repositories.

Getting Help with Ansible Galaxy

The Documentation tab on the Ansible Galaxy website home page leads to a page that describes how to use Ansible Galaxy. There is content that describes how to download and use roles from Ansible Galaxy. Instructions on how to develop roles and upload them to Ansible Galaxy are also on that page.

Browsing Ansible Galaxy for Roles

The Search tab on the left side of the Ansible Galaxy website home page gives users access to information about the roles published on Ansible Galaxy. You can search for an Ansible role by its name, using tags, or by 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.

NOTE

Content Scoring [https://galaxy.ansible.com/docs/contributing/content_scoring.html] in the documentation has more information on how roles are scored by Ansible Galaxy.

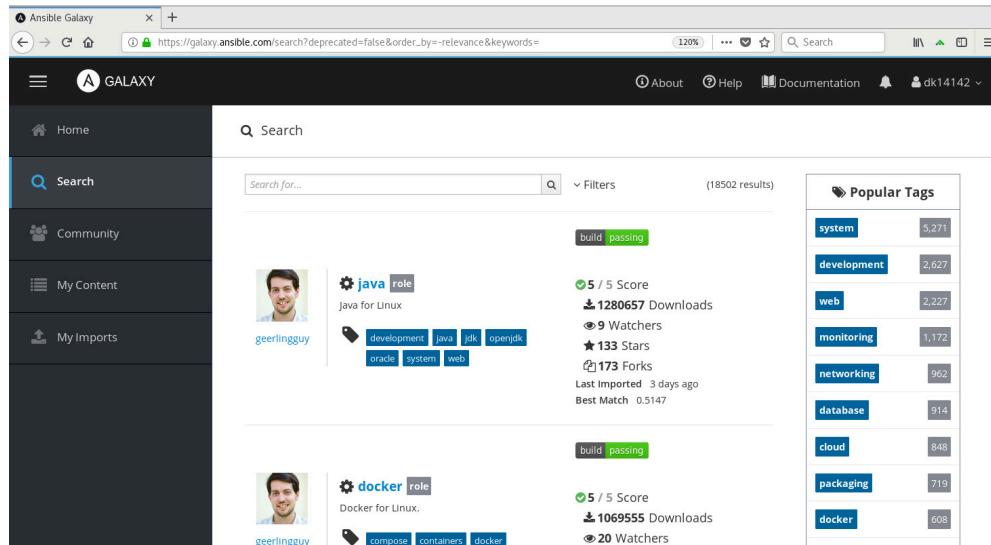


Figure 8.2: Ansible Galaxy search screen

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. Users 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. Notice the first result has a **Best Match** score of **0.9009**.

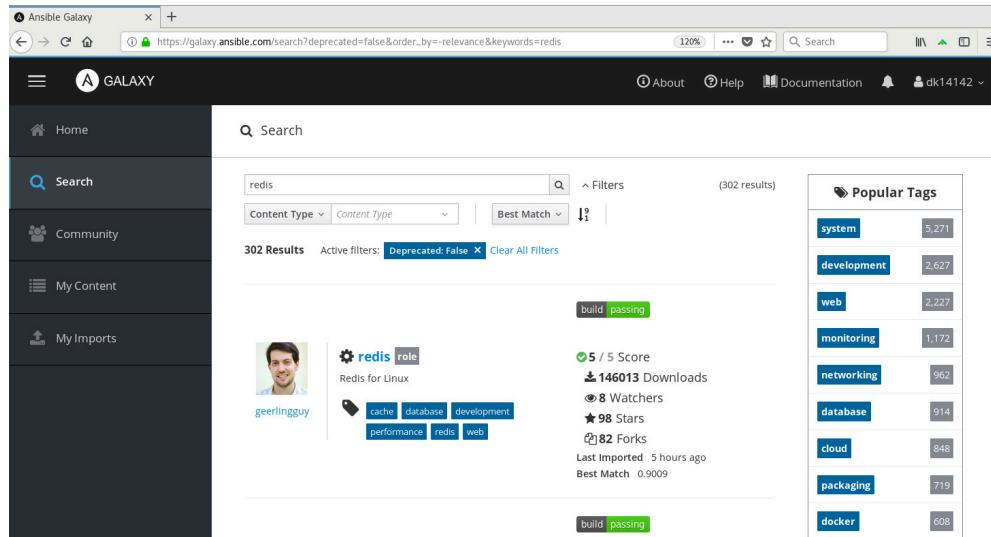


Figure 8.3: Ansible Galaxy search results example

The Filters pulldown menu to the right of the search box allow searches to be performed on keywords, author IDs, 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. Users can use tags to find relevant roles. Possible tag values include **system**, **development**, **web**, **monitoring**, and others. A role can have up to 20 tags in Ansible Galaxy.



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 set by the author for the role.

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.

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** will display 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** in their description, and are available for the Enterprise Linux (**EL**) platform.

```
[user@host ~]$ ansible-galaxy search 'redis' --platforms EL

Found 124 roles matching your search:

Name                                     Description
-----
1it.sudo                                  Ansible role for managing sudoers
AerisCloud.librato                         Install and configure the Librato Agent
AerisCloud.redis                           Installs redis on a server
AlbanAndrieu.java                          Manage Java installation
andrewrothstein.redis                     builds Redis from src and installs
...output omitted...
geerlingguy.php-redis                      PhpRedis support for Linux
geerlingguy.redis                           Redis for Linux
gikoluo.filebeat                          Filebeat for Linux.
...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 geerlingguy.redis

Role: geerlingguy.redis
      description: Redis for Linux
      active: True
...output omitted...
```

```
download_count: 146209
forks_count: 82
github_branch: master
github_repo: ansible-role-redis
github_user: gearlingguy
...output omitted...
license: license (BSD, MIT)
min_ansible_version: 2.4
modified: 2018-11-19T14:53:29.722718Z
open_issues_count: 11
path: [u'/etc/ansible/roles', u'/usr/share/ansible/roles']
role_type: ANS
stargazers_count: 98
...output omitted...
```

Installing Roles from Ansible Galaxy

The **ansible-galaxy install** subcommand downloads a role from Ansible Galaxy, then installs it locally on the control node.

By default, roles are installed into the first directory that is writable in the user's `roles_path`. Based on the default `roles_path` set for Ansible, normally the role will be installed into the user's `~/.ansible/roles` directory. The default `roles_path` might be overridden by your current Ansible configuration file or by the environment variable `ANSIBLE_ROLES_PATH`, which affects the behavior of **ansible-galaxy**.

You can also specify a specific directory to install the role into by using the **-p DIRECTORY** option.

In the following example, **ansible-galaxy** installs the `gearlingguy.redis` role into a playbook project's `roles` directory. The command's current working directory is `/opt/project`.

```
[user@host project]$ ansible-galaxy install gearlingguy.redis -p roles/
- downloading role 'redis', owned by gearlingguy
- downloading role from https://github.com/gearlingguy/...output omitted...
- extracting gearlingguy.redis to /opt/project/roles/gearlingguy.redis
- gearlingguy.redis (1.6.0) was installed successfully
[user@host ~]$ ls roles/
gearlingguy.redis
```

Installing Roles using a Requirements File

You can also use **ansible-galaxy** to install a list of roles based on definitions in a text file. For example, if you have a playbook that needs to have specific roles installed, 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 which enables playbooks to be developed and tested separately from any supporting roles.

For example, a simple `requirements.yml` to install `gearlingguy.redis` might read like this:

```
- src: gearlingguy.redis
version: "1.5.0"
```

The `src` attribute specifies the source of the role, in this case the `gearlingguy.redis` role from Ansible Galaxy. The `version` attribute is optional, and specifies the version of the role to install, in this case **1.5.0**.

**IMPORTANT**

You should specify the version of the role in your **requirements.yml** file, especially for playbooks in production.

If you do not specify a version, you will get the latest version of the role. If the upstream author makes changes to the role that are incompatible with your playbook, it may cause an automation failure or other problems.

To install the roles using a role file, use the **-r REQUIREMENTS-FILE** option:

```
[user@host project]$ ansible-galaxy install -r roles/requirements.yml \
> -p roles
- downloading role 'redis', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-redis/
archive/1.6.0.tar.gz
- extracting geerlingguy.redis to /opt/project/roles/geerlingguy.redis
- geerlingguy.redis (1.6.0) was installed successfully
```

You can use **ansible-galaxy** to install roles that are not in Ansible Galaxy. You can host your own proprietary or internal roles in a private Git repository or on a web server. The following example shows how to configure a requirements file using a variety of remote sources.

```
[user@host project]$ cat roles/requirements.yml
# from Ansible Galaxy, using the latest version
- src: geerlingguy.redis

# from Ansible Galaxy, overriding the name and using a specific version
- src: geerlingguy.redis
  version: "1.5.0"
  name: redis_prod

# from any Git-based repository, using HTTPS
- src: https://gitlab.com/guardianproject-ops/ansible-nginx-acme.git
  scm: git
  version: 56e00a54
  name: nginx-acme

# from any Git-based repository, using SSH
- src: git@gitlab.com:guardianproject-ops/ansible-nginx-acme.git
  scm: git
  version: master
  name: nginx-acme-ssh

# from a role tar ball, given a URL;
#   supports 'http', 'https', or 'file' protocols
- src: file:///opt/local/roles/myrole.tar
  name: myrole
```

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.

If the role is hosted in a source control repository, the **scm** attribute is required. The **ansible-galaxy** command is capable of downloading and installing roles from either a Git-based or

mercurial-based software repository. A Git-based repository requires an **scm** value of **git**, while a role hosted on a mercurial repository requires a value of **hg**. If the role is hosted on Ansible Galaxy or as a tar archive on a web server, the **scm** keyword is omitted.

The **name** keyword is used to override the local name of the role. 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.

To install the roles associated with a playbook project, execute the **ansible-galaxy install** command:

```
[user@host project]$ ansible-galaxy install -r roles/requirements.yml \
> -p roles
- downloading role 'redis', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-redis/
archive/1.6.0.tar.gz
- extracting geerlingguy.redis to /opt/project/roles/geerlingguy.redis
- geerlingguy.redis (1.6.0) was installed successfully
- downloading role 'redis', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-redis/
archive/1.5.0.tar.gz
- extracting redis_prod to /opt/project/roles/redis_prod
- redis_prod (1.5.0) was installed successfully
- extracting nginx-acme to /opt/project/roles/nginx-acme
- nginx-acme (56e00a54) was installed successfully
- extracting nginx-acme-ssh to /opt/project/roles/nginx-acme-ssh
- nginx-acme-ssh (master) was installed successfully
- downloading role from file:///opt/local/roles/myrole.tar
- extracting myrole to /opt/project/roles/myrole
- myrole was installed successfully
```

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 roles that are found locally.

```
[user@host project]$ ansible-galaxy list
- geerlingguy.redis, 1.6.0
- myrole, (unknown version)
- nginx-acme, 56e00a54
- nginx-acme-ssh, master
- redis_prod, 1.5.0
```

A role can be removed locally with the **ansible-galaxy remove** subcommand.

```
[user@host ~]$ ansible-galaxy remove nginx-acme-ssh
- successfully removed nginx-acme-ssh
[user@host ~]$ ansible-galaxy list
- geerlingguy.redis, 1.6.0
- myrole, (unknown version)
- nginx-acme, 56e00a54
- redis_prod, 1.5.0
```

Use downloaded and installed roles in playbooks like any other role. They may be referenced in the **roles** section using their downloaded role name. If a role is not in the project's **roles** directory, the **roles_path** will be checked to see if the role is installed in one of those directories, first match being used. The following **use-role.yml** playbook references the **redis_prod** and **geerlingguy.redis** roles:

```
[user@host project]$ cat use-role.yml
---
- name: use redis_prod for Prod machines
  hosts: redis_prod_servers
  user: devops
  become: true
  roles:
    - redis_prod

- name: use geerlingguy.redis for Dev machines
  hosts: redis_dev_servers
  user: devops
  become: true
  roles:
    - geerlingguy.redis
```

This playbook causes different versions of the **geerlingguy.redis** role to be applied to the production and development servers. In this manner, changes to the role can be systematically tested and integrated before deployment to the production servers. If a recent change to a role causes problems, using version control to develop the role allows you to roll back to a previous, stable version of the role.



REFERENCES

Ansible Galaxy – Ansible Documentation

https://docs.ansible.com/ansible/2.7/reference_appendices/galaxy.html

► GUIDED EXERCISE

DEPLOYING ROLES WITH ANSIBLE GALAXY

In this exercise, you will use Ansible Galaxy to download and install an Ansible role.

OUTCOMES

You should be able to:

- create a roles file to specify role dependencies for a playbook
- install roles specified in a roles file
- list roles using the **ansible-galaxy** command

SCENARIO OVERVIEW

Your organization places custom files in the **/etc/skel** directory on all hosts. As a result, new user accounts are configured with a standardized organization-specific Bash environment.

You will test the development version of the Ansible role responsible for deploying Bash environment skeleton files.

From workstation, run the command **lab role-galaxy setup** to prepare the environment for this exercise. This creates the working directory, **role-galaxy**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab role-galaxy setup
```

- 1. Log in to your workstation host as **student**. Change to the **role-galaxy** working directory.

```
[student@workstation ~]$ cd ~/role-galaxy  
[student@workstation role-galaxy]$
```

- 2. To test the Ansible role that configures skeleton files, add the role specification to a roles file.

Launch your favorite text editor and 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 **master** branch of the repository. Set the local name of the role to **student.bash_env**.

The **roles/requirements.yml** now contains the following content:

```
---  
# requirements.yml
```

```
- src: git@workstation.lab.example.com:student/bash_env
  scm: git
  version: master
  name: student.bash_env
```

- 3. Use the **ansible-galaxy** command to utilize the roles file you just created and install the **student.bash_env** role.

- 3.1. For comparison, display the contents of the **roles** subdirectory before the role is installed.

```
[student@workstation role-galaxy]$ ls roles/
requirements.yml
```

- 3.2. Use Ansible Galaxy to download and install the roles listed in the **roles/requirements.yml** file. Be sure that any downloaded roles are stored in the **roles** subdirectory.

```
[student@workstation role-galaxy]$ ansible-galaxy install -r \
> roles/requirements.yml -p roles
- extracting student.bash_env to /home/student/role-galaxy/roles/student.bash_env
- student.bash_env (master) was installed successfully
```

- 3.3. Display the **roles** subdirectory after the role has been installed. Confirm that it has a new subdirectory called **student.bash_env**, matching the **name** value specified in the YAML file.

```
[student@workstation role-galaxy]$ ls roles/
requirements.yml  student.bash_env
```

- 3.4. Try using the **ansible-galaxy** command, without any options, to list the project roles:

```
[student@workstation role-galaxy]$ ansible-galaxy list
[WARNING]: - the configured path /home/student/.ansible/roles does not exist.
```

Because you used the **-p** option with the **ansible-galaxy install** command, the **student.bash_env** role was not installed in the default location. Use the **-p** option with the **ansible-galaxy list** command to list the downloaded roles:

```
[student@workstation role-galaxy]$ ansible-galaxy list -p roles
- student.bash_env, master
[WARNING]: - the configured path /home/student/.ansible/roles does not exist.
```



NOTE

The **/home/student/.ansible/roles** directory is in your default **roles_path**, but since you have not attempted to install a role without using the **-p** option, **ansible-galaxy** has not yet created the directory.

- ▶ 4. Create a playbook, called **use-bash_env-role.yml**, that uses the **student.bash_env** role. The contents of the playbook should match the following:

```
---
- 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
      user:
        name: student2
        state: absent
        force: yes
        remove: yes

  roles:
    - student.bash_env

  post_tasks:
    - name: Create the test user
      user:
        name: student2
        state: present
        password: "{{ 'redhat' | password_hash('sha512', 'mysecretsalt') }}"
```

To see the effects of the configuration change, a new user account must be created. The **pre_tasks** and **post_tasks** section of the playbook ensure that the **student2** user account is created each time the playbook is executed. After playbook execution, the **student2** account is accessed with a password of **redhat**.

- ▶ 5. Run the 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-playbook 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] ****
ok: [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=3    unreachable=0    failed=0
```

- ▶ 6. Connect to servera as the student2 user using SSH. Observe the custom prompt for the student2 user, and then disconnect from servera.

```
[student@workstation role-galaxy]$ ssh student2@servera
[student2 on servera in ~ dir]$ exit
logout
Connection to servera closed.
[student@workstation role-galaxy]$
```

- ▶ 7. Execute 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 executing 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**. The **roles/requirements.yml** file now contains:

```
---
# requirements.yml

- src: git@workstation.lab.example.com:student/bash_env
  scm: git
  version: dev
  name: student.bash_env
```

- 7.2. Use the **ansible-galaxy install** command to install the role using the updated roles file. Use the **--force** option to overwrite the existing **master** version of the role with the **dev** version of the role.

```
[student@workstation role-galaxy]$ ansible-galaxy install \
> -r requirements.yml --force -p roles
- changing role student.bash_env from master to dev
- extracting student.bash_env to /home/student/role-galaxy/roles/student.bash_env
- student.bash_env (dev) was installed successfully
```

- 7.3. Edit the **use-bash-env-role.yml** file. Add the **prompt_color** variable with a value of **blue** to the **vars** section of the playbook. The file now contains:

```
---
- name: use student.bash_env role playbook
  hosts: devservers
  vars:
    prompt_color: blue
    default_prompt: '[\u on \h in \W]\$ '
  pre_tasks:
```

...output omitted...

7.4. Execute the `use-bash_env-role.yml` playbook.

```
[student@workstation role-galaxy]$ ansible-playbook 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] ****
changed: [servera.lab.example.com]

TASK [student.bash_env : put away .vimrc] ****
okay: [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
```

- 8. Connect again to `servera` as the `student2` using SSH. Observe the error for the `student2` user, and then disconnect from `servera`.

```
[student@workstation role-galaxy]$ ssh student2@servera
-bash: [: missing `]'
-bash-4.2$ exit
logout
Connection to servera closed.
[student@workstation role-galaxy]$
```

A Bash error occurred while parsing the `student2` user's `.bash_profile` file.

- 9. Correct the error in the development version of the `student.bash_env` role, and re-execute the playbook.
- 9.1. Edit the `roles/student.bash_env/templates/_bash_profile.j2` file. Add the missing `]` character to line 4 and save the file. The top of the file is now:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

```
# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin

export PATH
```

Save the file.

9.2. Execute the `use-bash_env-role.yml` playbook.

```
[student@workstation role-galaxy]$ ansible-playbook 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] ****
ok: [servera.lab.example.com]

TASK [student.bash_env : put away .bash_profile] ****
changed: [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
```

9.3. Connect again to servera as the student2 using SSH.

```
[student@workstation role-galaxy]$ ssh student2@servera
[student2 on servera in ~ dir]$ exit
logout
Connection to servera closed.
[student@workstation role-galaxy]$
```

The error message is no longer present. The custom prompt for the student2 user now displays with blue characters.

The steps above demonstrate that the development version of the `student.bash_env` role is defective. Based on testing results, developers will commit necessary fixes back to the development branch of the role. When the development branch passes required quality checks, developers merge features from the development branch into the `master` branch.

Committing role changes to a Git repository is beyond the scope of this course.

**IMPORTANT**

If you are tracking the latest version of a role in your project, periodically reinstall the role to update it. This ensures that your local copy stays current with bug fixes, patches, and other features.

On the other hand, if you are using a third-party role in production, you should specify the version that you want to use in order to avoid breakage due to unexpected changes. If you do this, you should be periodically updating to the latest version of the role in your test environment so that you can adopt improvements and changes in a controlled manner.

Cleanup

Run the **lab role-galaxy cleanup** command to cleanup the managed host.

```
[student@workstation ~]$ lab role-galaxy cleanup
```

This concludes the guided exercise.

REUSING CONTENT WITH SYSTEM ROLES

OBJECTIVES

After completing this section, students should be able to write playbooks that take advantage of Red Hat Enterprise Linux System Roles to perform standard operations.

RED HAT ENTERPRISE LINUX SYSTEM ROLES

Beginning with Red Hat Enterprise Linux 7.4, a number of Ansible roles have been provided with the operating system as part of the *rhel-system-roles* package in the Extras channel. A brief description of each role:

RHEL System Roles

NAME	STATE	ROLE DESCRIPTION
<code>rhel-system-roles.kdump</code>	Fully Supported	Configures the kdump crash recovery service.
<code>rhel-system-roles.network</code>	Fully Supported	Configures network interfaces.
<code>rhel-system-roles.selinux</code>	Fully Supported	Configures and manages SELinux customization, including SELinux mode, file and port contexts, Boolean settings, and SELinux users.
<code>rhel-system-roles.timesync</code>	Fully Supported	Configures time synchronization using Network Time Protocol or Precision Time Protocol.
<code>rhel-system-roles.postfix</code>	Technology Preview	Configures each host as a Mail Transfer Agent using the Postfix service.
<code>rhel-system-roles.firewall</code>	In Development	Configures a host's firewall.
<code>rhel-system-roles.tuned</code>	In Development	Configures the tuned service to tune system performance.

System roles aim to standardize the configuration of Red Hat Enterprise Linux subsystems, over multiple versions of Red Hat Enterprise Linux. Use system roles to configure any Red Hat Enterprise Linux host, version 6.10 and onward.

Simplified Configuration Management

As an example, the recommended time synchronization service for Red Hat Enterprise Linux 7 is the `chronyd` service. In Red Hat Enterprise Linux 6 however, the recommended service is the `ntpd` service. In an environment with a mixture of Red Hat Enterprise Linux 6 and 7 hosts, an administrator must manage the configuration files for both services.

With RHEL System Roles, administrators no longer need to maintain configuration files for both services. Administrators can use `rhel-system-roles.timesync` role to configure time synchronization for both Red Hat Enterprise Linux 6 and 7 hosts. A simplified YAML file containing role variables defines the configuration of time synchronization for both types of hosts.

Support for RHEL System Roles

RHEL System Roles are derived from the open source Linux System Roles project, found on Ansible Galaxy. Unlike Linux System Roles, RHEL System Roles are supported by Red Hat as part of a standard Red Hat Enterprise Linux subscription. RHEL System Roles have the same life cycle support benefits that come with a Red Hat Enterprise Linux subscription.

Every system role is tested and stable. The **Fully Supported** system roles also have stable interfaces. Any **Fully Supported** system role will continue to use the same role variables in future versions. Playbook refactoring due to system role changes should be minimal.

The **Technology Preview** system roles may utilize different role variables in future versions. Integration testing is recommended for playbooks that incorporate any **Technology Preview** role. Playbooks may require refactoring if role variables change in a future version of the role.

Other roles are in development in the upstream Linux System Roles project, but are not yet available through a RHEL subscription. These roles are available through Ansible Galaxy.

INSTALLING RHEL SYSTEM ROLES

The RHEL System Roles are provided by the `rhel-system-roles` package, which is available from the RHEL Extras channel. Install this package on the Ansible control node.

Use the following procedure to install the `rhel-system-roles` package. The procedure assumes the control node is registered to a Red Hat Enterprise Linux subscription and that Ansible Engine is installed. See the section on *Installing Ansible* for more information.

1. Enable the Extras channel.

```
[root@host ~]# subscription-manager repos --enable rhel-7-server-extras-rpms
```

2. Install RHEL System Roles.

```
[root@host ~]# yum install rhel-system-roles
```

After installation, the RHEL System roles are located in the `/usr/share/ansible/roles` directory:

```
[root@host ~]# ls -l /usr/share/ansible/roles/
total 20
...output omitted... linux-system-roles.kdump -> rhel-system-roles.kdump
...output omitted... linux-system-roles.network -> rhel-system-roles.network
...output omitted... linux-system-roles.postfix -> rhel-system-roles.postfix
...output omitted... linux-system-roles.selinux -> rhel-system-roles.selinux
...output omitted... linux-system-roles.timesync -> rhel-system-roles.timesync
...output omitted... rhel-system-roles.kdump
...output omitted... rhel-system-roles.network
...output omitted... rhel-system-roles.postfix
...output omitted... rhel-system-roles.selinux
...output omitted... rhel-system-roles.timesync
```

The corresponding upstream name of each role is linked to the RHEL System Role. This allows a role to be referenced in a playbook by either name.

The default `roles_path` on Red Hat Enterprise Linux includes `/usr/share/ansible/roles` in the path, so Ansible should automatically find those roles when referenced by a playbook.



NOTE

Ansible might not find the system roles if `roles_path` has been overridden in the current Ansible configuration file, if the environment variable `ANSIBLE_ROLES_PATH` is set, or if there is another role of the same name in a directory listed earlier in `roles_path`.

Accessing Documentation for RHEL System Roles

After installation, documentation for the RHEL System Roles is found in the `/usr/share/doc/rhel-system-roles-<version>/` directory. Documentation is organized into subdirectories by subsystem:

```
[root@host ~]# ls -l /usr/share/doc/rhel-system-roles-1.0/
total 4
drwxr-xr-x. ...output omitted... kdump
drwxr-xr-x. ...output omitted... network
drwxr-xr-x. ...output omitted... postfix
drwxr-xr-x. ...output omitted... selinux
drwxr-xr-x. ...output omitted... timesync
```

Each role's documentation directory contains a `README.md` file. The `README.md` file contains a description of the role, along with role usage information.

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 of the role documentation directories contain example playbooks. When using a role for the first time, review any additional example playbooks in the documentation directory.

Role documentation for RHEL System Roles matches the documentation for Linux System Roles. Use a web browser to access role documentation for the upstream roles at the Ansible Galaxy site, <https://galaxy.ansible.com>.

TIME SYNCHRONIZATION ROLE EXAMPLE

Suppose you need to configure NTP time synchronization on your servers. You could write automation yourself to perform each of the necessary tasks. But RHEL System Roles includes a role that can do this, `rhel-system-roles.timesync`.

The role is documented in its `README.md` in the `/usr/share/doc/rhel-system-roles-1.0/timesync` directory. The file describes all of the variables that affect the role's behavior and contains three playbook snippets illustrating different time synchronization configurations.

To manually configure NTP servers, the role has a variable named `timesync_ntp_servers`. It takes 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:

timesync_ntp_servers attributes

ATTRIBUTE	PURPOSE
hostname	The hostname of an NTP server with which to synchronize.
iburst	A Boolean that enables or disables fast initial synchronization. Defaults to no in the role, you should normally set this to yes .

Given this information, the following example is a play that uses the `rhel-system-roles.timesync` role to configure managed hosts to get time from three NTP servers using fast initial synchronization. In addition, a task has been added that uses the `timezone` module to set the hosts' time zone to UTC.

```
- name: Time Synchronization Play
hosts: servers
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
  timezone: UTC

roles:
  - rhel-system-roles.timesync

tasks:
  - name: Set timezone
    timezone:
      name: "{{ timezone }}"
```

**NOTE**

If you want to set a different time zone, you can use the `tzselect` command to look up other valid values. You can also use the `timedatectl` command to check current clock settings.

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 a playbook project with the following structure:

```
[root@host playbook-project]# tree
.
└── ansible.cfg
└── group_vars
   └── servers
      └── timesync.yml①
└── inventory
└── timesync_playbook.yml②
```

- ① Defines the time synchronization variables overriding the role defaults for hosts in group `servers` 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
  timezone: UTC
```

- ② The content of the playbook simplifies to:

```
- name: Time Synchronization Play
  hosts: servers
  roles:
    - rhel-system-roles.timesync
  tasks:
    - name: Set timezone
      timezone:
        name: "{{ timezone }}"
```

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 may 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.

SELINUX ROLE EXAMPLE

As another example, the `rhel-system-roles.selinux` role simplifies management of SELinux configuration settings. It 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 will ensure your desired SELinux configuration is applied.

Among the tasks this role can perform:

- 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 the managed hosts are rebooted in order to completely apply its changes. However, it does not ever reboot hosts itself. This is so that you can control how the reboot is handled. But it means that it is a little more complicated than usual to properly use this role in a play.

The way this works is that the role will set a Boolean variable, `selinux_reboot_required`, to `true` and fail 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: rhel-system-roles.selinux
  rescue:
    - name: Check for failure for other reasons than required reboot
      fail:
        when: not selinux_reboot_required

    - name: Restart managed host
      reboot:

    - name: Reapply SELinux role to complete changes
      include_role:
        name: rhel-system-roles.selinux
```

Configuring the SELinux Role

The variables used to configure the `rhel-system-roles.selinux` role are documented in its `README.md` file. The following examples show some ways to use this role.

The `selinux_state` variable sets the mode 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 hash/dictionary of variables: the name of the Boolean, the `state` (whether it should be `on` or `off`), and whether or not 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_fcontext` variable takes a list of file contexts to persistently set (or remove). It 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 `restorecon`:

```
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 its **README.md** file for more information.



REFERENCES

Red Hat Enterprise Linux (RHEL) System Roles

<https://access.redhat.com/articles/3050101>

Linux System Roles

<https://linux-system-roles.github.io/>

► GUIDED EXERCISE

REUSING CONTENT WITH SYSTEM ROLES

In this exercise, you will use one of the Red Hat Enterprise Linux System Roles in conjunction with a normal task to configure time synchronization and the time zone on your managed hosts.

OUTCOMES

You should be able to:

- Install the Red Hat Enterprise Linux System Roles.
- Find and use the RHEL System Roles documentation.
- Use the `rhel-system-roles.timesync` role in a playbook to configure time synchronization on remote hosts.

SCENARIO OVERVIEW

Your organization maintains two data centers: one in the United States (Chicago) and one in Finland (Helsinki). To aid log analysis of database servers across data centers, ensure the system clock on each host is synchronized using Network Time Protocol. To aid time-of-day activity analysis across data centers, ensure each database server has a time zone set that corresponds to the host's data center location.

Time synchronization has the following requirements:

- Use the NTP server located at `classroom.example.com`. Enable the `iburst` option to accelerate initial time synchronization.
- Use the `chrony` package for time synchronization.

From `workstation`, run the command `lab role-system setup` to prepare the environment for this exercise. This creates the working directory, `role-system`, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab role-system setup
```

- 1. Log in to your `workstation` host as `student`. Change to the `role-system` working directory.

```
[student@workstation ~]$ cd ~/role-system  
[student@workstation role-system]$
```

- 2. Install the Red Hat Enterprise Linux system roles on the control node, `workstation.lab.example.com`. Verify the installed location of the roles on the control node.
- 2.1. Use the `ansible-galaxy` command to verify that no roles are initially available for use in the playbook project.

```
[student@workstation role-system]$ ansible-galaxy list  
[student@workstation role-system]$
```

The **ansible-galaxy** command searches three directories for roles, as indicated by the `roles_path` entry in the **ansible.cfg** file:

- **./roles**
- **/usr/share/ansible/roles**
- **/etc/ansible/roles**

The above output indicates there are no roles in any of these directories.

2.2. Enable the `rhel-7-server-extras-rpms` repository.

```
[student@workstation role-system]$ sudo yum-config-manager \  
> --enable rhel-7-server-extras-rpms
```

2.3. Install the `rhel-system-roles` package.

```
[student@workstation role-system]$ sudo yum install rhel-system-roles
```

Enter **y** when prompted to install the package.

2.4. Use the **ansible-galaxy** command to verify that the system roles are now available.

```
[student@workstation role-system]$ ansible-galaxy list  
- linux-system-roles.kdump, (unknown version)  
- linux-system-roles.network, (unknown version)  
- linux-system-roles.postfix, (unknown version)  
- linux-system-roles.selinux, (unknown version)  
- linux-system-roles.timesync, (unknown version)  
- rhel-system-roles.kdump, (unknown version)  
- rhel-system-roles.network, (unknown version)  
- rhel-system-roles.postfix, (unknown version)  
- rhel-system-roles.selinux, (unknown version)  
- rhel-system-roles.timesync, (unknown version)
```

The roles are located in the **/usr/share/ansible/roles** directory. Any role beginning with **linux-system-roles** is actually a symlink to the corresponding **rhel-system-roles** role.

- ▶ 3. Create a playbook, `configure_time.yml`, with one play that targets the `database_servers` host group. Include the `rhel-system-roles.timesync` role in the `roles` section of the play.

The contents of the `configure_time.yml` now matches:

```
---  
- name: Time Synchronization  
  hosts: database_servers
```

```
roles:
  - 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 to 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.

- Review the *Role Variables* section of the **README.md** file for the **rhel-system-roles.timesync** role.

```
[student@workstation role-system]$ cat \
> /usr/share/doc/rhel-system-roles-1.0/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
```

- 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'
```

- Create a new file **group_vars/all/timesync.yml** using a text editor. Add variable definitions to satisfy the time synchronization requirements. The file now contains:

```
...
#rhel-system-roles.timesync variables for all hosts

timesync_ntp_provider: chrony

timesync_ntp_servers:
  - hostname: classroom.example.com
```

```
iburst: yes
```

- 5. Insert a task to set the time zone for each host. Ensure the task uses the `timezone` module and executes after the `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.

- Review the *Examples* section of the `timezone` module documentation.

```
[student@workstation role-system]$ ansible-doc timezone | grep -A 4 "EXAMPLES"
EXAMPLES:
- name: set timezone to Asia/Tokyo
  timezone:
    name: Asia/Tokyo
```

- Add a task to the `post_tasks` section of the play in the `configure_time.yml` playbook. Model the task after the example from the documentation, but use the `host_timezone` variable for the time zone name.

The `timezone` module documentation also recommends restarting the `cron` service after changing the timezone. Add a `notify` keyword to the task, with an associated value of `restart cron`. The `post_tasks` section of the play matches:

```
post_tasks:
- name: Set timezone
  timezone:
    name: "{{ host_timezone }}"
  notify: restart cron
```

- Add the `restart cron` handler to the **Time Synchronization** play. The complete playbook now contains:

```
---
- name: Time Synchronization
  hosts: database_servers

  roles:
    - rhel-system-roles.timesync

  post_tasks:
    - name: Set timezone
      timezone:
        name: "{{ host_timezone }}"
      notify: restart cron

  handlers:
    - name: restart cron
      service:
        name: cron
        state: restarted
```

- 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 playbook.

```
[student@workstation role-system]$ ansible-playbook configure_time.yml

PLAY [Time Synchronization] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [rhel-system-roles.timesync : Check if only NTP is needed] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

...output omitted...

TASK [rhel-system-roles.timesync : Enable timemaster] ****
skipping: [servera.lab.example.com]
skipping: [serverb.lab.example.com]

RUNNING HANDLER [rhel-system-roles.timesync : restart chronyrd] ****
changed: [servera.lab.example.com]
changed: [serverb.lab.example.com]

TASK [Set timezone] ****
changed: [serverb.lab.example.com]
```

```
changed: [servera.lab.example.com]

RUNNING HANDLER [restart crond] ****
changed: [serverb.lab.example.com]
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=17    changed=6      unreachable=0      failed=0
serverb.lab.example.com      : ok=17    changed=6      unreachable=0      failed=0
```

- 8. Verify the time zone settings of each server. Use an Ansible ad hoc command to see the output of the **date** command on all the database servers.

```
[student@workstation role-system]$ ansible database_servers -m shell -a date
serverb.lab.example.com | CHANGED | rc=0 >>
Tue Nov 27 23:24:27 EET 2018

servera.lab.example.com | CHANGED | rc=0 >>
Tue Nov 27 15:24:27 CST 2018
```

Each server has a time zone setting based on its geographic location.

Cleanup

Run the **lab role-system cleanup** command to cleanup the managed host.

```
[student@workstation ~]$ lab role-system cleanup
```

This concludes the guided exercise.

► LAB

IMPLEMENTING ROLES

PERFORMANCE CHECKLIST

In this lab, you will create Ansible roles that use variables, files, templates, tasks, and handlers.

OUTCOMES

You should be able to:

- Create Ansible roles that use variables, files, templates, tasks, and handlers to configure a development web server.
- Use a role that is hosted in a remote repository in a playbook.
- Use a Red Hat Enterprise Linux system role in a playbook.

SCENARIO OVERVIEW

Your organization must provide a single web server to host development code for all 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 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 using an assigned, nonstandard port.
- SELinux is set to enforcing and targeted.

Your playbook will:

- 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 written by the organization to configure Apache. You should define the dependency using version **v1.4** of the organizational role. The URL of the dependency's repository is: `git@workstation.lab.example.com:infra/apache`

- Use the `rhel-system-roles.selinux` role to configure SELinux for the nonstandard HTTP ports used by your web server. You will be provided with a `selinux.yml` variable file that can be installed as a `group_vars` file to pass the correct settings to the role.

Log in as the **student** user on `workstation` and run `lab role-review setup`. The script creates the project directory, `role-review`, and populates it with an Ansible configuration file, host inventory, and other lab files.

```
[student@workstation ~]$ lab role-review setup
```

1. Log in to your workstation host as **student**. Change to the **role-review** working directory.
2. The host group for the development web server should be **dev_webserver**. Confirm this host group exists in the **inventory** file.
3. Create a playbook named **web_dev_server.yml** with a single play named **Configure Dev Web Server**. Configure the play to target the host group **dev_webserver**. Do not add any roles or tasks to the play yet.

Ensure that the play forces handlers to execute, because you may encounter an error while developing the playbook.
4. Check the syntax of the playbook. Run the playbook. The syntax check should pass and the playbook should run successfully.
5. Make sure that playbook's role dependencies are installed.

The **apache.developer_configs** role that you will create depends on the **infra.apache** role. Create a **roles/requirements.yml** file. It should install 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 that repository automatically. Install the role with the **ansible-galaxy** command.

In addition, install the **rhel-system-roles** package.
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 to be the tasks file for this role.

The **developer.conf.j2** file in the project directory is a Jinja2 template used by the tasks file. Move it to the correct location for template files used by this role.
7. The **apache.developer_configs** role will process a list of users defined in a variable named **web_developers**. The **web_developers.yml** file in the project directory defines the **web_developers** user list variable. Review this file and use it to define the **web_developers** variable for the development web server host group.
8. Add the role **apache.developer_configs** to the play in the **web_dev_server.yml** playbook.
9. Check the syntax of the playbook. 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 have been provided with a variable file, **selinux.yml**, which can be used 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. Review the lecture or the documentation for this role to see how to do this.

Inspect the **selinux.yml** file. Move it to the correct location so that its variables are set for the **dev_webserver** host group.
11. Check syntax of the final playbook. The syntax check should pass.
12. Run the 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

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

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

Cleanup

On workstation, run the **lab role-review cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab role-review cleanup
```

This concludes the lab.

► SOLUTION

IMPLEMENTING ROLES

PERFORMANCE CHECKLIST

In this lab, you will create Ansible roles that use variables, files, templates, tasks, and handlers.

OUTCOMES

You should be able to:

- Create Ansible roles that use variables, files, templates, tasks, and handlers to configure a development web server.
- Use a role that is hosted in a remote repository in a playbook.
- Use a Red Hat Enterprise Linux system role in a playbook.

SCENARIO OVERVIEW

Your organization must provide a single web server to host development code for all 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 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 using an assigned, nonstandard port.
- SELinux is set to enforcing and targeted.

Your playbook will:

- 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 written by the organization to configure Apache. You should define the dependency using version **v1.4** of the organizational role. The URL of the dependency's repository is: `git@workstation.lab.example.com:infra/apache`

- Use the `rhel-system-roles.selinux` role to configure SELinux for the nonstandard HTTP ports used by your web server. You will be provided with a `selinux.yml` variable file that can be installed as a `group_vars` file to pass the correct settings to the role.

Log in as the **student** user on `workstation` and run `lab role-review setup`. The script creates the project directory, `role-review`, and populates it with an Ansible configuration file, host inventory, and other lab files.

```
[student@workstation ~]$ lab role-review setup
```

1. Log in to your workstation host as **student**. Change to the **role-review** working directory.

```
[student@workstation ~]$ cd ~/role-review  
[student@workstation role-review]$
```

2. The host group for the development web server should be **dev_webserver**. Confirm this host group exists in the **inventory** file.

```
[student@workstation role-review]$ cat inventory  
[controlnode]  
workstation.lab.example.com  
  
[dev_webserver]  
servera.lab.example.com
```

3. Create a playbook named **web_dev_server.yml** with a single play named **Configure Dev Web Server**. Configure the play to target the host group **dev_webserver**. Do not add any roles or tasks to the play yet.

Ensure that the play forces handlers to execute, because you may encounter an error while developing the playbook.

Once complete, the **/home/student/role-review/web_dev_server.yml** playbook contains:

```
---  
- name: Configure Dev Web Server  
  hosts: dev_webserver  
  force_handlers: yes
```

4. Check the syntax of the playbook. Run the playbook. The syntax check should pass and the playbook should run successfully.

```
[student@workstation role-review]$ ansible-playbook \  
> --syntax-check web_dev_server.yml  
  
playbook: web_dev_server.yml  
[student@workstation role-review]$ ansible-playbook 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
```

5. Make sure that playbook's role dependencies are installed.

The **apache**.**developer_configs** role that you will create depends on the **infra.apache** role. Create a **roles/requirements.yml** file. It should install 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 that repository automatically. Install the role with the **ansible-galaxy** command.

In addition, install the *rhel-system-roles* package.

- 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.

Once complete, the **roles/requirements.yml** file contains:

```
- 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
- extracting infra.apache to /home/student/role-review/roles/infra.apache
- infra.apache (v1.4) was installed successfully
```

- 5.4. Install the RHEL System Roles package.

```
[student@workstation role-review]$ sudo yum install rhel-system-roles
```

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 to be the tasks file for this role.

The **developer.conf.j2** file in the project directory is a Jinja2 template used by the tasks file. Move it to the correct location for template files used by this role.

- 6.1. Use the **ansible-galaxy init** 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
- apache.developer_configs was created successfully
[student@workstation roles]$ cd ..
```

```
[student@workstation role-review]$
```

- 6.2. Update 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
```

Use **grep** to confirm the correct contents of the dependencies variable:

```
[student@workstation role-review]$ cd roles/apache.developer_configs  
[student@workstation apache.developer_configs]$ grep -A4 \  
> dependencies: meta/main.yml  
dependencies:  
  - name: infra.apache  
    src: git@workstation.lab.example.com:infra/apache  
    scm: git  
    version: v1.4  
[student@workstation apache.developer_configs]$ cd /home/student/role-review  
[student@workstation role-review]$
```

- 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  
'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/  
'developer.conf.j2' -> 'roles/apache.developer_configs/templates/  
developer.conf.j2'
```

7. The apache.developer_configs role will process a list of users defined in a variable named **web_developers**. The **web_developers.yml** file in the project directory defines the **web_developers** user list variable. Review this file and use it to define the **web_developers** variable for the development web server host group.

- 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, user_port is defined for each web developer.

- 7.2. Place the **web_developers.yml** 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
'developers.yml' -> 'group_vars/dev_webserver/developers.yml'
```

8. Add the role apache.developer_configs to the play in the **web_dev_server.yml** playbook.

The edited playbook:

```
---
- name: Configure Dev Web Server
  hosts: dev_webserver
  force_handlers: yes
  roles:
    - apache.developer_configs
```

9. Check the syntax of the playbook. 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-playbook \
> --syntax-check web_dev_server.yml

playbook: web_dev_server.yml
[student@workstation role-review]$ ansible-playbook web_dev_server.yml

PLAY [Configure Dev Web Server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

...output omitted...

TASK [infra.apache : Install a skeleton index.html] ****
skipping: [servera.lab.example.com]

TASK [apache.developer_configs : Create user accounts] ****
changed: [servera.lab.example.com] => (item={u'username': u'jdoe', u'user_port': 9081, u'name': u'John Doe'})
changed: [servera.lab.example.com] => (item={u'username': u'jdoe2', u'user_port': 9082, u'name': u'Jane Doe'})

...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. See \\"systemctl status httpd.service\\" and \\"journalctl -xe\\"
for details.\n"}

NO MORE HOSTS LEFT ****
to retry, use: --limit @/home/student/role-review/web_dev_server.retry

PLAY RECAP ****
servera.lab.example.com    : ok=13    changed=7    unreachable=0    failed=1
```

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 have been provided with a variable file, `selinux.yml`, which can be used 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. Review the lecture or the documentation for this role to see how to do this.

Inspect the `selinux.yml` file. Move it to the correct location so that its variables are set for the `dev_webserver` host group.

- 10.1. The `pre_tasks` section can be added to the end of the play in the `web_dev_server.yml` playbook.

You can look at the block in `/usr/share/doc/rhel-system-roles-1.0/selinux/example-selinux-playbook.yml` for a basic outline of how to apply the role, but Red Hat Ansible Engine 2.7 allows you to replace the complex shell and `wait_for` logic with the `reboot` module.

The `pre_tasks` section should contain:

```
pre_tasks:
  - name: Check SELinux configuration
    block:
      - include_role:
          name: rhel-system-roles.selinux
    rescue:
      # Fail if failed for a different reason than selinux_reboot_required.
      - name: Check for general failure
        fail:
          msg: "SELinux role failed."
          when: not selinux_reboot_required

      - name: Restart managed host
        reboot:
          msg: "Ansible rebooting system for updates."

      - name: Reapply SELinux role to complete changes
        include_role:
```

```
name: 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/
'selinux.yml' -> 'group_vars/dev_webserver/selinux.yml'
```

11. Check syntax of the final playbook. The syntax check should pass.

```
[student@workstation role-review]$ ansible-playbook \
> --syntax-check web_dev_server.yml

playbook: web_dev_server.yml
[student@workstation role-review]$
```

The final **web_dev_server.yml** playbook should read as follows:

```
---
- name: Configure Dev Web Server
  hosts: dev_webserver
  force_handlers: yes
  roles:
    - apache.developer_configs
  pre_tasks:
    - name: Check SELinux configuration
      block:
        - include_role:
            name: rhel-system-roles.selinux
  rescue:
    # Fail if failed for a different reason than selinux_reboot_required.
    - name: Check for general failure
      fail:
        msg: "SELinux role failed."
      when: not selinux_reboot_required

    - name: Restart managed host
      reboot:
        msg: "Ansible rebooting system for updates."
```

```
- name: Reapply SELinux role to complete changes
  include_role:
    name: rhel-system-roles.selinux
```

**NOTE**

Whether **pre_tasks** is at the end of the play or in the "correct" position in terms of execution order in the playbook file does not matter to **ansible-playbook**. It will still run the play's tasks in the correct order.

12. Run the playbook. It should succeed.

```
[student@workstation role-review]$ ansible-playbook web_dev_server.yml

PLAY [Configure Dev Web Server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [include_role : rhel-system-roles.selinux] ****
TASK [rhel-system-roles.selinux : Install SELinux python2 tools] ****
ok: [servera.lab.example.com]

...output omitted...

TASK [infra.apache : Apache Service is started] ****
changed: [servera.lab.example.com]

...output omitted...

TASK [apache.developer_configs : Copy Per-Developer Config files] ****
ok: [servera.lab.example.com] => (item={'username': u'jdoe', 'user_port': 9081,
  'name': u'John Doe'})
ok: [servera.lab.example.com] => (item={'username': u'jdoe2', 'user_port': 9082,
  'name': u'Jane Doe'})

PLAY RECAP ****
servera.lab.example.com      : ok=18    changed=3    unreachable=0    failed=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

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

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

Cleanup

On workstation, run the **lab role-review cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab role-review cleanup
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Roles organize Ansible code in a way that allows reuse and sharing.
- Role variables defined in **defaults/main.yml** are overridden by inventory or play variables. Those in **vars/main.yml** are used internally by the role.
- If you provide a list of roles to the **roles** section of a play, then they run before tasks in the **tasks** section of your play. Any handlers they notify run after the tasks in **tasks** run.
- Tasks in the **pre_tasks** section run and run their notified handlers before the **roles** and **tasks**. Tasks in the **post_tasks** section run and run their notified handlers after the **roles** and **tasks**.
- Recent versions of Ansible allow you to include or import a role as a task with the **include_role** and **import_role** modules.
- Ansible Galaxy [<https://galaxy.ansible.com>] is a public library of Ansible roles written by Ansible users.
- The **ansible-galaxy** command can search for, display information about, install, list, remove, or initialize roles.
- External roles needed by a playbook may be defined in the **roles/requirements.yml** file. The **ansible-galaxy install -r requirements.yml** command uses this file to install the roles on the control node.
- Red Hat Enterprise Linux System Roles are a collection of tested and supported roles intended to help you configure host subsystems across versions of Red Hat Enterprise Linux.
- The default **roles_path** setting automatically finds the RHEL System Roles installed in the **/usr/share/ansible/roles** directory.

CHAPTER 9

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

After completing this section, students should be able to troubleshoot generic issues with a new playbook and repair them.

Figure 9.0: Troubleshooting Ansible Playbooks

LOG FILES FOR ANSIBLE

By default, Red Hat Ansible Engine is not configured to log its output to any log file. It 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 `$ANSIBLE_LOG_PATH` environment variable. If any or both are configured, Ansible stores output from both the `ansible` and `ansible-playbook` commands in the log file configured either through the `ansible.cfg` configuration file or the `$ANSIBLE_LOG_PATH` environment variable.

If Ansible log files are to be kept in the default log file directory, `/var/log`, then the playbooks must be run as `root` or the permissions on `/var/log` must be updated. More frequently, log files are created in the local playbook directory.



NOTE

If you configure Ansible to write log files to `/var/log`, Red Hat recommends that you configure `logrotate` to manage the Ansible log files.

THE DEBUG MODULE

One of the modules available for Ansible, the `debug` module, can help provide better insight into what is happening in the play. This module can display the value for a certain variable at a certain point in the play. This feature is key to debugging 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 of `debug` tasks. The 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-playbook`. The second example displays the value of the `output` variable.

```
- name: Display free memory
  debug:
    msg: "The free memory for this system is {{ ansible_facts['memfree_mb'] }}"
```

```
- name: Display the "output" variable
  debug:
    var: output
    verbosity: 2
```

MANAGING ERRORS

There are several issues than can occur during a playbook run, mainly 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). Those errors are issued by the **ansible-playbook** command at execution time.

Earlier in this course, you learned about the **--syntax-check** option, which checks the YAML syntax for the playbook. It is a good practice to run a syntax check on your playbook before using it or if you are having problems with it.

```
[student@demo ~]$ ansible-playbook play.yml --syntax-check
```

You can also use the **--step** option to step through a playbook one task at a time. The **ansible-playbook --step** command interactively prompts for confirmation that you want each task to run.

```
[student@demo ~]$ ansible-playbook play.yml --step
```

The **--start-at-task** option allows you to start execution of a playbook from a specific task. It takes as an argument the name of the task at which to start.

```
[student@demo ~]$ ansible-playbook play.yml --start-at-task="start httpd service"
```

DEBUGGING WITH ANSIBLE-PLAYBOOK

The output given by a playbook that was run with the **ansible-playbook** command is a good starting point for troubleshooting issues related to hosts managed by Ansible. Consider the following output from a playbook execution:

```
PLAY [Service Deployment] ****
...output omitted...
TASK: [Install a service] ****
ok: [demoservera]
ok: [demoserverb]

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

The previous output shows a **PLAY** header with the name of the play to be executed, followed by one or more **TASK** headers. Each of these headers represents their associated *task* in the playbook, and it is executed in all the managed hosts belonging to the group included in the playbook in the *hosts* parameter.

As each managed host executes each play's tasks, the name of the managed host is displayed under the corresponding **TASK** header, along with the task state on that managed host. Task states can appear as **ok**, **fatal**, **changed**, or **skipping**.

At the bottom of the output for each play, the **PLAY RECAP** section displays the number of tasks executed for each managed host.

As discussed earlier in the course, you can increase the verbosity of the output from **ansible-playbook** by adding one or more **-v** options. The **ansible-playbook -v** command provides additional debugging information, with up to four total levels.

Verbosity Configuration

OPTION	DESCRIPTION
-v	The output data is displayed.
-vv	Both the output and input data are displayed.
-vvv	Includes information about connections to managed hosts.
-vvvv	Includes additional information such scripts that are executed on each remote host, and the user that is executing each script.

RECOMMENDED PRACTICES FOR PLAYBOOK MANAGEMENT

Although the previously discussed tools can help to identify and fix issues in playbooks, when developing those playbooks it is important to keep in mind some recommended practices that can help ease the troubleshooting process. Some recommended practices for playbook development are listed below:

- 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.
- Include 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.



REFERENCES

Configuring Ansible – Ansible Documentation

https://docs.ansible.com/ansible/2.7/installation_guide/intro_configuration.html

debug – Print statements during execution – Ansible Documentation

https://docs.ansible.com/ansible/2.7/modules/debug_module.html

Best Practices – Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_best_practices.html

► GUIDED EXERCISE

TROUBLESHOOTING PLAYBOOKS

In this exercise, you will troubleshoot a playbook that has been given to you that does not work properly.

OUTCOMES

You should be able to:

- Troubleshoot playbooks.

Log in to workstation as **student** using **student** as the password.

On workstation, run the **lab troubleshoot-playbook setup** script. It verifies whether Ansible is installed on workstation. It also creates the **/home/student/troubleshoot-playbook/** directory, and downloads to this directory the **inventory**, **samba.yml**, and **samba.conf.j2** files from <http://materials.example.com/labs/troubleshoot-playbook/>.

```
[student@workstation ~]$ lab troubleshoot-playbook setup
```

- 1. On workstation, change to the **/home/student/troubleshoot-playbook/** directory.

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

- 2. Create a file named **ansible.cfg** in the current directory. It should set the **log_path** parameter to write Ansible logs to the **/home/student/troubleshoot-playbook/ansible.log** file. It should set the **inventory** parameter to use the **/home/student/troubleshoot-playbook/inventory** file deployed by the lab script.

When you are finished, **ansible.cfg** should have the following contents:

```
[defaults]
log_path = /home/student/troubleshoot-playbook/ansible.log
inventory = /home/student/troubleshoot-playbook/inventory
```

- 3. Run the playbook. It will fail with an error.

This playbook would set up a Samba server if everything were correct. However, the run will fail due to missing double quotes on the **random_var** variable definition. Read the error message to see how **ansible-playbook** reports the problem. Notice the variable **random_var** is assigned a value that contains a colon and is not quoted.

```
[student@workstation troubleshoot-playbook]$ ansible-playbook samba.yml
ERROR! Syntax Error while loading YAML.
mapping values are not allowed in this context
```

```
The error appears to have been 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 the error has been properly logged to the **/home/student/troubleshoot-playbook/ansible.log** file.

```
[student@workstation troubleshoot-playbook]$ tail ansible.log  
The error appears to have been 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 playbook and correct the error by adding quotes to the entire value being assigned to **random_var**. The corrected version of **samba.yml** should contain the following content:

```
...output omitted...  
vars:  
  install_state: installed  
  random_var: "This is colon: test"  
...output omitted...
```

- ▶ 6. Check the playbook using the **--syntax-check** option. Another error is issued due to extra white space in the indentation on the last task, **deliver samba config**.

```
[student@workstation troubleshoot-playbook]$ ansible-playbook --syntax-check \  
> samba.yml  
ERROR! Syntax Error while loading YAML.  
      did not find expected key
```

```
The error appears to have been in '/home/student/troubleshoot-playbook/samba.yml':  
line 43, column 4, but may  
be elsewhere in the file depending on the exact syntax problem.
```

The offending line appears to be:

```
- name: deliver samba config  
  ^ here
```

- 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
  firewalld:
    state: enabled
    permanent: true
    immediate: true
    service: samba

- name: deliver samba config
  template:
    src: templates/samba.conf.j2
    dest: /etc/samba/smb.conf
    owner: root
    group: root
    mode: 0644
```

- 8. Run the playbook using the `--syntax-check` option. An error is issued due to the `install_state` variable being used as a parameter in the `install samba` task. It is not quoted.

```
[student@workstation troubleshoot-playbook]$ ansible-playbook --syntax-check \
> samba.yml
ERROR! Syntax Error while loading YAML.
  found unacceptable key (unhashable type: 'AnsibleMapping')
```

The error appears to have been in '/home/student/troubleshoot-playbook/samba.yml':
line 14, column 15, 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 }}"
```

- 9. Edit the playbook and correct the `install samba` task. The reference to the `install_state` variable should be in quotes. The resulting file content should look like the following:

```
...output omitted...
```

```
tasks:  
- name: install samba  
  yum:  
    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-playbook --syntax-check \  
> samba.yml  
  
playbook: samba.yml
```

- 11. Run the playbook. An error, related to SSH, will be issued.

```
[student@workstation troubleshoot-playbook]$ ansible-playbook samba.yml  
PLAY [Install a samba server] *****  
  
TASK [Gathering Facts] *****  
fatal: [servera.lab.exammples.com]: UNREACHABLE! => {"changed": false,  
"msg": "Failed to connect to the host via ssh: ssh: Could not resolve hostname  
servera.lab.exammples.com: Name or service not known\r\n", "unreachable": true}  
      to retry, use: --limit @/home/student/troubleshoot-playbook/samba.retry  
  
PLAY RECAP *****  
servera.lab.exammples.com : ok=0     changed=0    unreachable=1   failed=0
```

- 12. Ensure the managed host `servera.lab.example.com` is running, using the **ping** command.

```
[student@workstation troubleshoot-playbook]$ ping -c3 servera.lab.example.com  
PING servera.lab.example.com (172.25.250.10) 56(84) bytes of data.  
64 bytes from servera.lab.example.com (172.25.250.10): icmp_seq=1 ttl=64  
time=0.247 ms  
64 bytes from servera.lab.example.com (172.25.250.10): icmp_seq=2 ttl=64  
time=0.329 ms  
64 bytes from servera.lab.example.com (172.25.250.10): icmp_seq=3 ttl=64  
time=0.320 ms  
  
--- servera.lab.example.com ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 1999ms  
rtt min/avg/max/mdev = 0.247/0.298/0.329/0.041 ms
```

- 13. Ensure that you can connect to the managed host `servera.lab.example.com` 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-playbook]$ ssh devops@servera.lab.example.com  
Warning: Permanently added 'servera.lab.example.com,172.25.250.10' (ECDSA) to the  
list of known hosts.
```

```
...output omitted...
[devops@servera ~]$ exit
Connection to servera.lab.example.com closed.
```

- ▶ 14. Rerun the 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-playbook]$ ansible-playbook -vvvv samba.yml
ansible-playbook 2.7.1
  config file = /home/student/troubleshoot-playbook/ansible.cfg
  configured module search path = [u'~/home/student/.ansible/plugins/modules', u'~/usr/share/ansible/plugins/modules']
    ansible python module location = /usr/lib/python2.7/site-packages/ansible
    executable location = /usr/bin/ansible-playbook
    python version = 2.7.5 (default, Sep 12 2018, 05:31:16) [GCC 4.8.5 20150623 (Red
    Hat 4.8.5-36)]
Using /home/student/troubleshoot-playbook/ansible.cfg as config file
setting up inventory plugins
Parsed /home/student/troubleshoot-playbook/inventory inventory source with ini
  plugin
Loading callback plugin default of type stdout, v2.0 from /usr/lib/python2.7/site-
packages/ansible/plugins/callback/default.pyc

PLAYBOOK: samba.yml ****
1 plays in samba.yml

PLAY [Install a samba server] ****

TASK [Gathering Facts] ****
task path: /home/student/troubleshoot-playbook/samba.yml:2
<servera.lab.exammples.com> ESTABLISH SSH CONNECTION FOR USER: devops
...output omitted...
fatal: [servera.lab.exammples.com]: UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: OpenSSH_7.4p1, OpenSSL 1.0.2k-
fips 26 Jan 2017\r\ndebug1: Reading configuration data /home/student/.ssh/config\r\ndebug1: /home/student/.ssh/config line 1: Applying options for *\r\ndebug1: Reading configuration data /etc/ssh/ssh_config\r\ndebug1: /etc/ssh/ssh_config
line 58: Applying options for *\r\ndebug1: auto-mux: Trying existing master\r\ndebug1: Control socket \"/home/student/.ansible/cp/d4775f48c9\" does not exist\r\ndebug2: resolving \"servera.lab.exammples.com\" port 22\r\nssh: Could not resolve
hostname servera.lab.exammples.com: Name or service not known\r\n",
    "unreachable": true
}

...output omitted...
PLAY RECAP ****
servera.lab.exammples.com : ok=0      changed=0      unreachable=1      failed=0
```

- ▶ 15. When using the highest level of verbosity with Ansible, the Ansible log file is a better option to check output than the console. Review the output from the previous command in the `/home/student/troubleshoot-playbook/ansible.log` file.

```
[student@workstation troubleshoot-playbook]$ tail ansible.log
```

```
2018-12-17 19:22:50,508 p=18287 u=student | task path: /home/student/troubleshoot-playbook/samba.yml:2
2018-12-17 19:22:50,549 p=18287 u=student | fatal: [servera.lab.exammple.com]: UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: OpenSSH_7.4p1, OpenSSL 1.0.2k-fips 26 Jan 2017\r\ndebug1: Reading configuration data /home/student/.ssh/config\r\ndebug1: /home/student/.ssh/config line 1: Applying options for *\r\ndebug1: Reading configuration data /etc/ssh/ssh_config\r\ndebug1: /etc/ssh/ssh_config line 58: Applying options for *\r\ndebug1: auto-mux: Trying existing master\r\ndebug1: Control socket \"/home/student/.ansible/cp/d4775f48c9\" does not exist\r\ndebug2: resolving \"servera.lab.exammple.com\" port 22\r\nssh: Could not resolve hostname servera.lab.exammple.com: Name or service not known\r\n",
    "unreachable": true
}
2018-12-17 19:22:50,550 p=18287 u=student | to retry, use: --limit @/home/student/troubleshoot-playbook/samba.retry

2018-12-17 19:22:50,550 p=18287 u=student | PLAY RECAP ****
2018-12-17 19:22:50,550 p=18287 u=student | servera.lab.exammple.com : ok=0
changed=0    unreachable=1    failed=0
```

- ▶ 16. Investigate the **inventory** file for errors. Notice the **[samba_servers]** group has misspelled servera.lab.example.com. Correct this error as shown below:

```
...output omitted...
[samba_servers]
servera.lab.example.com
...output omitted...
```

- ▶ 17. 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 **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 working directory, **/home/student/troubleshoot-playbook/**.

```
[student@workstation troubleshoot-playbook]$ ansible-playbook samba.yml

PLAY [Install a samba server] ****
...output omitted...
TASK [debug install_state variable] ****
ok: [servera.lab.example.com] => {
    "msg": "The state for the samba service is installed"
}
...output omitted...
TASK [deliver samba config] ****
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/templates/samba.j2\n\t/home/student/troubleshoot-playbook/samba.j2\nIf you are using a module and expect the file to exist on the remote, see the remote_src option"}
...output omitted...
PLAY RECAP ****
```

```
servera.lab.example.com      : ok=7      changed=3      unreachable=0      failed=1
```

- 18. Edit the playbook, and correct the **src** parameter in the *deliver samba config* task to be **samba.conf.j2**. When you are finished it should look like the following:

```
...output omitted...
- name: deliver samba config
  template:
    src: samba.conf.j2
    dest: /etc/samba/smb.conf
    owner: root
...output omitted...
```

- 19. Run the playbook again. Execute the playbook using the **--step** option. It should run without errors.

```
[student@workstation troubleshoot-playbook]$ ansible-playbook samba.yml --step

PLAY [Install a samba server] ****
Perform task: TASK: Gathering Facts (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: install samba (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: install firewalld (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: debug install_state variable (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: start samba (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: start firewalld (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: configure firewall for samba (N)o/(y)es/(c)ontinue: y
...output omitted...
Perform task: TASK: deliver samba config (N)o/(y)es/(c)ontinue: y
...output omitted...
PLAY RECAP ****
servera.lab.example.com      : ok=8      changed=1      unreachable=0      failed=0
```

Cleanup

On workstation, run the **lab troubleshoot-playbook cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab troubleshoot-playbook cleanup
```

This concludes the guided exercise.

TROUBLESHOOTING ANSIBLE MANAGED HOSTS

OBJECTIVES

After completing this section, students should be able to troubleshoot failures on managed hosts when running a playbook.

USING CHECK MODE AS A TESTING TOOL

You can use the **ansible-playbook --check** command to run smoke tests on a playbook. This option executes the playbook without making changes to the managed hosts' configuration. 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 the changes are not displayed but the module still takes no action.

```
[student@demo ~]$ ansible-playbook --check playbook.yml
```



NOTE

The **ansible-playbook --check** command might not work properly if your tasks use conditionals.

You can also control whether individual tasks run in check mode with the `check_mode` setting. If a task has `check_mode: yes` set, it always runs in check mode, whether or not you passed the `--check` option to **ansible-playbook**. Likewise, if a task has `check_mode: no` set, it always runs normally, even if you pass `--check` to **ansible-playbook**.

The following task is always run in check mode, and does not make changes.

```
tasks:
  - name: task always in check mode
    shell: uname -a
    check_mode: yes
```

The following task is always run normally, even when started with **ansible-playbook --check**.

```
tasks:
  - name: task always runs even in check mode
    shell: uname -a
    check_mode: no
```

This can be useful because you can run most of a playbook normally while testing individual tasks with `check_mode: yes`. Likewise, you can make test runs in check mode more likely to provide reasonable results by running selected tasks that gather facts or set variables for conditionals but do not change the managed hosts with `check_mode: no`.

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 will run even when the playbook is run with **ansible-playbook --check**. Therefore, you cannot trust that the **--check** option will make no changes to managed hosts, without confirming this to be the case by 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 will have to replace that code with **check_mode: no** in Ansible 2.6 and later.

The **ansible-playbook** 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-playbook --check --diff playbook.yml
```

MODULES FOR TESTING

Some modules can provide additional information about the status of a managed host. The following list includes some of the Ansible modules that can be used to test and debug issues on managed hosts.

- The **uri** module provides a way to check that a RESTful API is returning the required content.

```
tasks:
  - uri:
      url: http://api.myapp.com
      return_content: yes
      register: apiresponse

  - fail:
      msg: 'version was not provided'
      when: "'version' not in apiresponse.content"
```

- The **script** module supports executing a script on managed hosts, and fails if the return code for that script is nonzero. The script must exist on the control node and is transferred to and executed on the managed hosts.

```
tasks:
  - script: check_free_memory
```

- The **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 **stat** task will not fail, but its registered variable will report **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
    stat:
      path: /var/run/app.lock
    register: lock

  - name: Fail if the application is running
    fail:
      when: not lock.stat.exists
```

- The **assert** module is an alternative to the **fail** module. The **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 **assert** instead of **fail**.

```
tasks:
  - name: Check if /var/run/app.lock exists
    stat:
      path: /var/run/app.lock
    register: lock

  - name: Fail if the application is running
    assert:
      that:
        - not lock.stat.exists
```

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.

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. You should also confirm that you have the correct SSH keys set up or are providing the correct password for that user.

Make sure that **become** is set properly, and that you are using the correct **become_user** (this is **root** by default). You should confirm that you are entering the correct **sudo** password and that **sudo** on the managed host is configured correctly.

A more subtle problem has to do with inventory settings. For a complex server with multiple network addresses, you may 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 will override 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 host **web4.phx.example.com**:

```
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.

USING AD HOC COMMANDS FOR TESTING

The following examples illustrate some of the checks that can be made on a managed host through the use of ad hoc commands.

You have used the `ping` module to test whether you can connect to managed hosts. Depending on the options you pass, you can also use it to test whether privilege escalation and credentials are correctly configured.

```
[student@demo ~]$ ansible demohost -m ping
demohost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
[student@demo ~]$ ansible demohost -m ping --become
demohost | FAILED! => {
    "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 currently available space on the disks configured in 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 command -a 'df'
```

This example returns the currently available free memory on the `demohost` managed host.

```
[student@demo ~]$ ansible demohost -m command -a 'free -m'
```

THE CORRECT LEVEL OF TESTING

Ansible is designed to ensure that the configuration included in playbooks and performed by its modules is correct. It monitors all modules for reported failures, and stops the playbook immediately if any failure is encountered. This helps ensure that any task performed before the failure has no errors.

Because of this, there is usually no need to check if the result of a task managed by Ansible has been correctly applied on the managed hosts. It makes sense to add some health checks either to playbooks, or run those directly as ad hoc commands, when more direct troubleshooting is required. But, you should be careful about adding too much complexity to your tasks and plays in an effort to double check the tests performed by the modules themselves.



REFERENCES

Check Mode ("Dry Run") -- Ansible Documentation

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_checkmode.html

Testing Strategies -- Ansible Documentation

https://docs.ansible.com/ansible/2.7/reference_appendices/test_strategies.html

► GUIDED EXERCISE

TROUBLESHOOTING ANSIBLE MANAGED HOSTS

In this exercise, you will 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.

Log in to workstation as student using student as the password.

On workstation, run the `lab troubleshoot-host setup` script. It ensures that Ansible is installed on workstation. It also downloads the `inventory`, `mailrelay.yml`, and `postfix-relay-main.conf.j2` files from `http://materials.example.com/troubleshoot-host/` to the `/home/student/troubleshoot-host/` directory.

```
[student@workstation ~]$ lab troubleshoot-host setup
```

- 1. On workstation, change to the `/home/student/troubleshoot-host/` directory.

```
[student@workstation ~]$ cd ~/troubleshoot-host/
```

- 2. Run the `mailrelay.yml` playbook using check mode.

```
[student@workstation troubleshoot-host]$ ansible-playbook 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 [email notification of always_bcc config] ****
fatal: [servera.lab.example.com]: FAILED! => {"msg": "The conditional check
'bcc_state.stdout != 'always_bcc =' failed. The error was: error while
evaluating conditional (bcc_state.stdout != 'always_bcc ='): 'dict object'
has no attribute 'stdout'\n\nThe error appears to have been in '/home/student/
troubleshoot-host/mailrelay.yml': line 42, column 7, but may\nbe elsewhere in the
file depending on the exact syntax problem.\n\nThe offending line appears to be:
\n\n      - name: email notification of always_bcc config\n            ^ here\n"}
...output omitted...
PLAY RECAP ****
```

```
servera.lab.example.com      : ok=6      changed=3      unreachable=0      failed=1
```

The *verify main.cf file exists* task uses the `stat` module. It confirmed that `main.cf` exists on `servera.lab.example.com`.

The *email notification of always_bcc config* task failed. It did not receive output from the *check for always_bcc* task because the playbook was executed using check mode.

- 3. Using an ad hoc command, check the header for the `/etc/postfix/main.cf` file.

```
[student@workstation troubleshoot-host]$ ansible servera.lab.example.com \
> -u devops -b -a "head /etc/postfix/main.cf"
servera.lab.example.com | FAILED | rc=1 >>
head: cannot open '/etc/postfix/main.cf' for reading: No such file or
directorynon-zero return code
```

The command failed because the playbook was executed using check mode. Postfix is not installed on `servera.lab.example.com`.

- 4. Run the playbook again, but without specifying check mode. The error in the *email notification of always_bcc config* task should disappear.

```
[student@workstation troubleshoot-host]$ ansible-playbook mailrelay.yml
PLAY [create mail relay servers] ****
...output omitted...
TASK [check for always_bcc] ****
changed: [servera.lab.example.com]

TASK [email notification of always_bcc config] ****
skipping: [servera.lab.example.com]

RUNNING HANDLER [restart postfix] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=8      changed=5      unreachable=0      failed=0
```

- 5. Using an ad hoc command, display the top of the `/etc/postfix/main.cf` file.

```
[student@workstation troubleshoot-host]$ ansible servera.lab.example.com \
> -u devops -b -a "head /etc/postfix/main.cf"
servera.lab.example.com | SUCCESS | rc=0 >>
# Ansible managed
#
# Global Postfix configuration file. This file lists only a subset
# of all parameters. For the syntax, and for a complete parameter
# list, see the postconf(5) manual page (command: "man 5 postconf").
#
# For common configuration examples, see BASIC_CONFIGURATION_README
# and STANDARD_CONFIGURATION_README. To find these documents, use
# the command "postconf html_directory readme_directory", or go to
```

```
# http://www.postfix.org/.
```

Now it starts with a line that contains the string, “Ansible managed”. This file was updated and is now managed by Ansible.

- 6. Add a task to enable the `smtp` service through the firewall.

```
[student@workstation troubleshoot-host]$ vim mailrelay.yml
...output omitted...
- name: postfix firewalld config
  firewalld:
    state: enabled
    permanent: true
    immediate: true
    service: smtp
...output omitted...
```

- 7. Run the playbook. The `postfix firewalld config` task should have been executed with no errors.

```
[student@workstation troubleshoot-host]$ ansible-playbook 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
```

- 8. Using an ad hoc command, check that the `smtp` service is now configured on the firewall at `servera.lab.example.com`.

```
[student@workstation troubleshoot-host]$ ansible servera.lab.example.com \
> -u devops -b -a "firewall-cmd --list-services"
servera.lab.example.com | CHANGED | rc=0 >>
dhcpv6-client samba smtp ssh
```

- 9. Use `telnet` to test if the SMTP service is listening on port `TCP/25` on `servera.lab.example.com`. 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 '^>'.
220 servera.lab.example.com ESMTP Postfix
quit
221 2.0.0 Bye
Connection closed by foreign host.
```

Cleanup

On workstation, run the `lab troubleshoot-host cleanup` script to clean up this exercise.

```
[student@workstation ~]$ lab troubleshoot-host cleanup
```

This concludes the guided exercise.

► LAB

TROUBLESHOOTING ANSIBLE

PERFORMANCE CHECKLIST

In this lab, you will troubleshoot problems that occur when you try to run a playbook that has been provided to you.

OUTCOMES

You should be able to:

- Troubleshoot playbooks.
- Troubleshoot managed hosts.

Log in to workstation as student using student as the password. Run the **lab troubleshoot-review setup** command.

```
[student@workstation ~]$ lab troubleshoot-review setup
```

This script verifies that Ansible is installed on **workstation**, and creates the **~student/troubleshoot-review/** directory, and the **html** subdirectory in it. It also downloads from <http://materials.example.com/labs/troubleshoot-review/> the **ansible.cfg**, **inventory-lab**, **secure-web.yml**, and **vhosts.conf** files to the **/home/student/troubleshoot-review/** directory, and the **index.html** file to the **/home/student/troubleshoot-review/html/** directory.

1. From the **~/troubleshoot-review** directory, check the syntax of the **secure-web.yml** playbook. This playbook contains one play that sets up Apache HTTPD with TLS/SSL for hosts in the group **webservers**. Fix the issue that is reported.
2. Check the syntax of the **secure-web.yml** playbook again. Fix the issue that is reported.
3. Check the syntax of the **secure-web.yml** playbook a third time. Fix the issue that is reported.
4. Check 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 **serverb.lab.example.com**. Fix this problem.
6. Run the **secure-web.yml** playbook again. Ansible is not able to authenticate as the **devops** remote user on the managed host. Fix this issue.
7. Run the **secure-web.yml** playbook a third time. Fix the issue that is reported.
8. 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.

Evaluation

On **workstation**, run the **lab troubleshoot-review grade** script to confirm success on this exercise.

```
[student@workstation troubleshoot-review]$ lab troubleshoot-review grade
```

Cleanup

On workstation, run the **lab troubleshoot-review cleanup** script to clean up this lab.

```
[student@workstation troubleshoot-review]$ lab troubleshoot-review cleanup
```

► SOLUTION

TROUBLESHOOTING ANSIBLE

PERFORMANCE CHECKLIST

In this lab, you will troubleshoot problems that occur when you try to run a playbook that has been provided to you.

OUTCOMES

You should be able to:

- Troubleshoot playbooks.
- Troubleshoot managed hosts.

Log in to workstation as student using student as the password. Run the **lab troubleshoot-review setup** command.

```
[student@workstation ~]$ lab troubleshoot-review setup
```

This script verifies that Ansible is installed on workstation, and creates the **~student/troubleshoot-review/** directory, and the **html** subdirectory in it. It also downloads from <http://materials.example.com/labs/troubleshoot-review/> the **ansible.cfg**, **inventory-lab**, **secure-web.yml**, and **vhosts.conf** files to the **/home/student/troubleshoot-review/** directory, and the **index.html** file to the **/home/student/troubleshoot-review/html/** directory.

1. From the **~/troubleshoot-review** directory, check the syntax of the **secure-web.yml** playbook. This playbook contains one play that sets up Apache HTTPD with TLS/SSL for hosts in the group **webservers**. Fix the issue that is reported.
 - 1.1. On workstation, change to the **/home/student/troubleshoot-review** project directory.

```
[student@workstation ~]$ cd ~/troubleshoot-review/
```

- 1.2. Check 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-playbook --syntax-check \
> secure-web.yml
```

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

The error appears to have been in '/home/student/Ansible-course/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. Correct the syntax issue in the definition of the `random_var` variable by adding double quotes 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. Check the syntax of the `secure-web.yml` playbook again. Fix the issue that is reported.
- 2.1. Check the syntax of `secure-web.yml` using `ansible-playbook --syntax-check` again.

```
[student@workstation troubleshoot-review]$ ansible-playbook --syntax-check \  
> secure-web.yml
```

```
ERROR! Syntax Error while loading YAML.  
      did not find expected '-' indicator
```

The error appears to have been in '/home/student/Ansible-course/troubleshoot-review/secure-web.yml': line 43, 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  
    service:  
      name: httpd  
      state: started  
      enabled: yes  
    tags:  
      - services  
  
  - name: deliver content  
    copy:  
      dest: /var/www/vhosts/serverb-secure  
      src: html/
```

...output omitted...

3. Check the syntax of the **secure-web.yml** playbook a third time. Fix the issue that is reported.

3.1. Check the syntax of the **secure-web.yml** playbook.

```
[student@workstation troubleshoot-review]$ ansible-playbook --syntax-check \
> secure-web.yml
```

```
ERROR! Syntax Error while loading YAML.
  found unacceptable key (unhashable type: 'AnsibleMapping')
```

The error appears to have been in '/home/student/Ansible-course/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:

```
yum:
  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. Add double quotes to **{{ item }}**. The resulting change should appear as follows:

...output omitted...

```
- name: install web server packages
  yum:
    name: "{{ item }}"
    state: latest
  notify:
    - restart services
  tags:
    - packages
  loop:
    - httpd
    - mod_ssl
    - crypto-utils
...output omitted...
```

4. Check the syntax of the **secure-web.yml** playbook a fourth time. It should not show any syntax errors.

```
[student@workstation troubleshoot-review]$ ansible-playbook --syntax-check \
> secure-web.yml

playbook: secure-web.yml
```

5. Run the **secure-web.yml** playbook. Ansible is not able to connect to `serverb.lab.example.com`. Fix this problem.

- 5.1. Run the **secure-web.yml** playbook. This will fail with an error.

```
[student@workstation troubleshoot-review]$ ansible-playbook 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: Warning: Permanently added
  'serverc.lab.example.com,172.25.250.12' (ECDSA) to the list of known hosts.\r
  \nPermission denied (publickey,gssapi-keyex,gssapi-with-mic,password).\r\n",
  "unreachable": true}
    to retry, use: --limit @/home/student/troubleshoot-review/secure-web.retry

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

- 5.2. Run the **secure-web.yml** playbook again, adding the **-vvvv** parameter to increase the verbosity of the output.

Notice that Ansible appears to be connecting to `serverc.lab.example.com` instead of `serverb.lab.example.com`.

```
[student@workstation troubleshoot-review]$ ansible-playbook secure-web.yml -vvvv
...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 -vvv -C -o ControlMaster=auto
-o ControlPersist=60s -o KbdInteractiveAuthentication=no -o
PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o
PasswordAuthentication=no -o User=students -o ConnectTimeout=10 -o ControlPath=/home/student/.ansible/cp/bc0c05136a serverc.lab.example.com '/bin/sh -c '""'"echo
~students && sleep 0'""'"
...output omitted...
```

- 5.3. Correct the line in the **inventory-lab** file. Delete the `ansible_host` host variable so the file appears as shown below:

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

6. Run the **secure-web.yml** playbook again. Ansible is not able to authenticate as the devops remote user on the managed host. Fix this issue.

6.1. Run the **secure-web.yml** playbook.

```
[student@workstation troubleshoot-review]$ ansible-playbook secure-web.yml -vvvv
...output omitted...
TASK [Gathering Facts] ****
task path: /home/student/troubleshoot-review/secure-web.yml:3
<serverb.lab.example.com> ESTABLISH SSH CONNECTION FOR USER: students
<serverb.lab.example.com> EXEC ssh -C -vvv -o ControlMaster=auto
-o ControlPersist=60s -o Port=22 -o KbdInteractiveAuthentication=no
-o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey
-o PasswordAuthentication=no -o User=students -o ConnectTimeout=10
-o ControlPath=/home/student/.ansible/cp/ansible-ssh-%C -tt
serverb.lab.example.com '/bin/sh -c """( umask 22 && mkdir -p """
echo $HOME/.ansible/tmp/ansible-tmp-1460241127.16-3182613343880 `" &&
echo "` echo $HOME/.ansible/tmp/ansible-tmp-1460241127.16-3182613343880
`" )"""
...output omitted...
fatal: [serverb.lab.example.com]: UNREACHABLE! => {
...output omitted...
```

- 6.2. Edit the **secure-web.yml** playbook to make sure 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...
```

7. Run the **secure-web.yml** playbook a third time. Fix the issue that is reported.

7.1. Run the **secure-web.yml** playbook.

```
[student@workstation troubleshoot-review]$ ansible-playbook secure-web.yml -vvvv
...output omitted...
failed: [serverb.lab.example.com] => (item=[u'httpd', u'mod_ssl',
u'crypto-utils']) => {"changed": true, "failed": true, "invocation":
{"module_args": {"conf_file": null, "disable_gpg_check": false, "disablerepo": null,
"enablerepo": null, "exclude": null, "install_repoquery": true,
"list": null, "name": ["httpd", "mod_ssl", "crypto-utils"], "state": "latest",
"update_cache": false}, "module_name": "yum"}, "item": ["httpd", "mod_ssl",
"crypto-utils"], "msg": "You need to be root to perform this command.\n",
"rc": 1, "results": ["Loaded plugins: langpacks, search-disabled-repos\n"]}
...output omitted...
```

- 7.2. Edit the play to make sure that it has `become: true` or `become: yes` 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...
```

8. 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.

8.1. Run the **secure-web.yml** playbook.

```
[student@workstation troubleshoot-review]$ ansible-playbook secure-web.yml
PLAY [create secure web service] ****
...output omitted...
TASK [install web server packages] ****
changed: [serverb.lab.example.com] => (item=[u'httpd', u'mod_ssl', u'crypto-utils'])
...output omitted...
TASK [httpd_conf_syntax variable] ****
ok: [serverb.lab.example.com] => {
    "msg": "The httpd_conf_syntax variable value is {'stderr_lines': [u'Syntax OK'], u'changed': True, u'end': u'2018-12-17 23:31:53.191871', 'failed': False, u'stdout': u'', u'cmd': [u'/sbin/httpd', u'-t'], u'rc': 0, u'start': u'2018-12-17 23:31:53.149759', u'stderr': u'Syntax OK', u'delta': u'0:00:00.042112', 'stdout_lines': [], 'failed_when_result': False}"
}
...output omitted...
RUNNING HANDLER [restart services] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com      : ok=10    changed=7    unreachable=0    failed=0
```

- 8.2. Use an ad hoc command to determine the state of the `httpd` service on `serverb.lab.example.com`. The `httpd` service should now be running on `serverb.lab.example.com`.

```
[student@workstation troubleshoot-review]$ ansible all -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 Tue 2016-05-03 19:43:39 CEST; 12s ago
...output omitted...
```

Evaluation

On `workstation`, run the `lab troubleshoot-review grade` script to confirm success on this exercise.

```
[student@workstation troubleshoot-review]$ lab troubleshoot-review grade
```

Cleanup

On workstation, run the **lab troubleshoot-review cleanup** script to clean up this lab.

```
[student@workstation troubleshoot-review]$ lab troubleshoot-review cleanup
```

SUMMARY

In this chapter, you learned:

- Ansible provides built-in logging. This feature is not enabled by default.
- The `log_path` parameter in the **default** section of the `ansible.cfg` configuration file specifies the location of the log file to which all Ansible output is redirected.
- The `debug` module provides additional debugging information while running a playbook (for example, current value for a variable).
- The `-v` option of the `ansible-playbook` command provides several levels of output verbosity. This is useful for debugging Ansible tasks when running a playbook.
- The `--check` option enables Ansible modules with check mode support to display the changes to be performed, instead of applying those changes to the managed hosts.
- Additional checks can be executed on the managed hosts using ad hoc commands.
- There is no need to double check the configuration performed by Ansible as long as the playbook completes successfully.

CHAPTER 10

INSTALLING AND ACCESSING ANSIBLE TOWER

GOAL

Explain what Red Hat Ansible Tower is and demonstrate a basic ability to navigate and use its web UI.

OBJECTIVES

- Describe the architecture, use cases, and installation requirements of Ansible Tower.
- Install Red Hat Ansible Tower in a single-server configuration.
- Navigate and describe the Ansible Tower web UI, and successfully launch a job using the demo job template, project, credential, and inventory.

SECTIONS

- Explaining the Red Hat Ansible Tower Architecture (and Quiz)
- Installing Red Hat Ansible Tower (and Guided Exercise)
- Navigating Red Hat Ansible Tower (and Guided Exercise)

QUIZ

Installing and Accessing Red Hat Ansible Tower

EXPLAINING THE RED HAT ANSIBLE TOWER ARCHITECTURE

OBJECTIVES

After completing this section, students should be able to describe the architecture, use cases, and installation requirements of Red Hat Ansible Tower.

WHY RED HAT ANSIBLE TOWER?

As an enterprise's experience with Ansible matures, it often finds additional opportunities for leveraging Ansible to simplify and improve IT operations. The same Ansible Playbooks used by operations teams to deploy production systems can also be used to deploy identical systems in earlier stages of the software development life cycle. When automated with Ansible, complex production support tasks typically handled by skilled engineers can easily be delegated to and resolved by entry-level technicians.

However, sharing an existing Ansible infrastructure to scale IT automation across an enterprise can present some challenges. While properly written Ansible Playbooks can be used across teams, Ansible does not provide any facilities for managing their shared access. Additionally, though playbooks may allow for the delegation of complex tasks, their execution may require highly privileged and guarded administrator credentials.

IT teams often vary in their preferred tool sets. While some may prefer the direct execution of playbooks, other teams may wish to trigger playbook execution from existing continuous integration and delivery tool suites. In addition, those that traditionally work with GUI-based tools may find Ansible's pure command-line interface intimidating and awkward.

Red Hat Ansible Tower overcomes many of these problems by providing a framework for running and managing Ansible efficiently on an enterprise scale. Ansible Tower eases the administration involved with sharing an Ansible infrastructure while maintaining organization security by introducing features such as a centralized web UI for playbook management, *role-based access control (RBAC)*, and centralized logging and auditing. Its REST API ensures that Ansible Tower integrates easily with an enterprise's existing workflows and tool sets. The Ansible Tower API and notification features make it particularly easy to associate Ansible Playbooks with other tools such as Jenkins, CloudForms, or Red Hat Satellite, to enable continuous integration and deployment. It provides mechanisms to enable centralized use and control of machine credentials and other secrets without exposing them to end users of Ansible Tower.

RED HAT ANSIBLE TOWER ARCHITECTURE

Ansible Tower is a Django web application designed to run on a Linux server as an on-premise, self-hosted solution which layers on top of an enterprise's existing Ansible infrastructure.

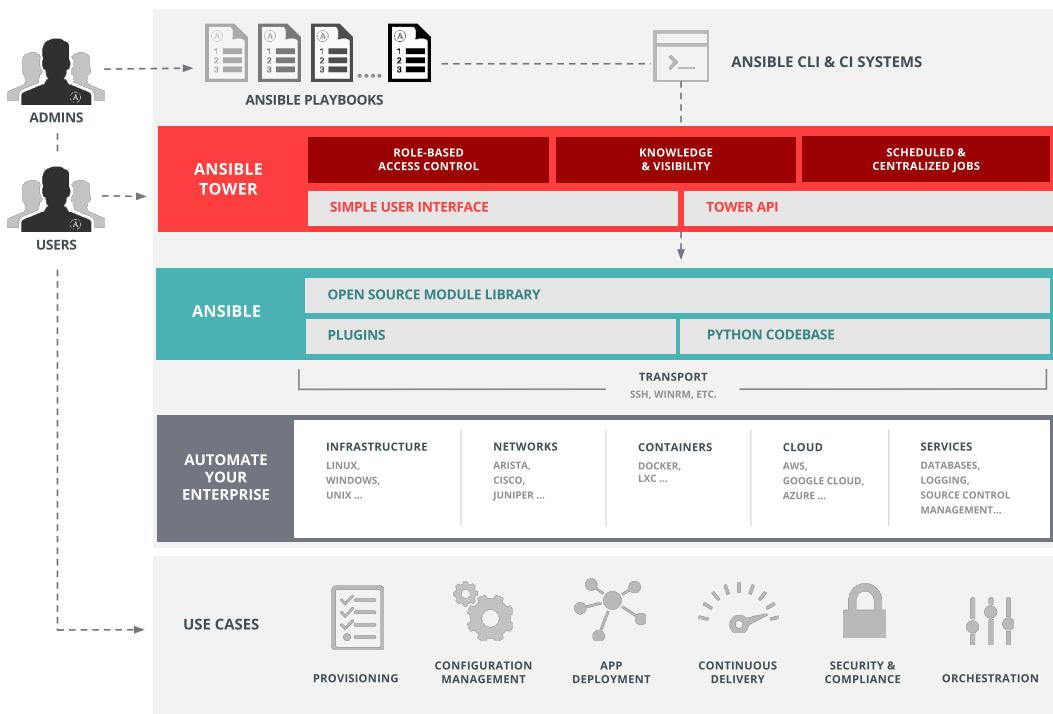


Figure 10.1: Ansible Tower architecture

Users interact with their enterprise's underlying Ansible infrastructure using either the Ansible Tower web UI or RESTful API. The Ansible Tower web UI is a graphical interface which performs its actions by executing calls against the Ansible Tower API. Any action available through the Ansible Tower web UI can therefore also be performed using the Ansible Tower RESTful API. The RESTful API is essential for those users looking to integrate Ansible with existing software tools and processes.

Ansible Tower stores its data in a PostgreSQL back-end database and makes use of the RabbitMQ messaging system. Versions of Ansible Tower prior to version 3.0 also relied on a MongoDB database. This dependency has since been removed, and data is now stored solely in a PostgreSQL database.

Depending on the needs of the enterprise, Ansible Tower can be implemented using one of the following architectures.

Single Machine with Integrated Database

All Ansible Tower components, the web front-end, RESTful API back end, and PostgreSQL database, reside on a single machine. This is the standard architecture.

Single Machine with Remote Database

The Ansible Tower web UI and RESTful API back end are installed on a single machine, and the PostgreSQL database is installed on another server on the same network. The remote database can be hosted on a server with an existing PostgreSQL instance outside the management of Ansible Tower. Another option is to have the Ansible Tower installer create a PostgreSQL instance on the remote server, managed by Ansible Tower, and populate it with the Ansible Tower database.

High Availability Multimachine Cluster

Earlier Ansible Tower versions offered a redundant, active-passive architecture consisting of a single active node and one or more inactive nodes. Starting with Red Hat Ansible Tower 3.1, this architecture is now replaced by an active-active, high-availability cluster consisting of multiple active Ansible Tower nodes.

Each node in the cluster hosts the Ansible Tower web UI and RESTful API back end, and can receive and process requests. In this cluster architecture, the PostgreSQL database is hosted on a remote server. The remote database can reside either on a server with an existing PostgreSQL instance outside the management of Ansible Tower, or on a server with a PostgreSQL instance created by the installer, and managed by Ansible Tower.

OpenShift Pod with Remote Database

In this architecture, Red Hat Ansible Tower operates as a container-based cluster running on Red Hat OpenShift. The cluster runs on an OpenShift pod, which contains four containers to run the Ansible Tower components. OpenShift adds or removes pods to scale Ansible Tower up and down. The installation procedure for this architecture is different from the other architectures.



NOTE

This course focuses on the most straightforward architecture to deploy; that is, a single Red Hat Ansible Tower server with an integrated database.

RED HAT ANSIBLE TOWER FEATURES

Two types of license are available for Ansible Tower: basic and enterprise. An enterprise license offers access to all Ansible Tower features. A basic license offers access to only a subset of the Ansible Tower features and does not include many enterprise-level options, such as logging aggregation, and clustering.

The following are some of the many features offered by Ansible Tower for controlling, securing, and managing Ansible in an enterprise environment.

Visual Dashboard

The Ansible Tower web UI displays a Dashboard which provides a summary view of an enterprise's entire Ansible environment. The Ansible Tower Dashboard allows administrators to easily see the current status of hosts and inventories, as well as the results of recent job executions.

Role-based Access Control (RBAC)

Ansible Tower uses a Role-based Access Control (RBAC) system which maintains security while streamlining user access management. It simplifies the delegation of user access to Ansible Tower objects such as Organizations, Projects, and Inventories.

Graphical Inventory Management

You can use the Ansible Tower web UI to create inventory groups and add inventory hosts. You can also update inventories from an external inventory source such as public cloud providers, local virtualization environments, and an organization's custom *configuration management database (CMDB)*.

Job Scheduling

You can use Ansible Tower to schedule playbook execution and updates from external data sources either on a one-time basis or recurring at regular intervals. This allows routine tasks to be performed unattended and is especially useful for tasks such as backup routines, which are ideally executed during operational off-hours.

Real-time and Historical Job Status Reporting

When you initiate a playbook execution in Ansible Tower, the web UI displays the playbook's output and execution results in real time. The results of previously executed jobs and scheduled job runs are also available in Ansible Tower.

User-triggered Automation

Ansible simplifies IT automation and Ansible Tower takes it a step further by enabling user self-service. The Ansible Tower streamlined web UI, coupled with the flexibility of its RBAC system, allows administrators to reduce complex tasks to simple easy-to-use routines.

Remote Command Execution

Ansible Tower makes the on-demand flexibility of Ansible ad hoc commands available through its remote command execution feature. User permissions for remote command execution are enforced using the Ansible Tower RBAC system.

Credential Management

Ansible Tower centrally manages authentication credentials. This means that you can run Ansible plays on managed hosts, synchronize information from dynamic inventory sources, and import Ansible project content from version control systems. It encrypts the passwords or keys provided so that they cannot be retrieved by Ansible Tower users. Users can be granted the ability to use or replace these credentials without actually exposing them to the user.

Centralized Logging and Auditing

Ansible Tower logs all playbook and remote command execution. This provides the ability to audit when each job was executed and by whom. In addition, Ansible Tower offers the ability to integrate its log data into third-party logging aggregation solutions, such as Splunk and Sumologic.

Integrated Notifications

Ansible Tower notifies you when its job executions succeed or fail. Ansible Tower can deliver notifications using many different applications, including email, Slack, and HipChat.

Multiplaybook Workflows

Complex operations often involve the serial execution of multiple playbooks. Ansible Tower multiplaybook workflows allow users to chain together multiple playbooks to facilitate the execution of complex routines involving provisioning, configuration, deployment, and orchestration. An intuitive workflow editor also helps to simplify the modeling of multiplaybook workflows.

RESTful API

The Ansible Tower RESTful API exposes every Ansible Tower feature available through the web UI. The API's browsable format makes it self-documenting and simplifies the lookup of API usage information.



REFERENCES

For more information, refer to the *Ansible Tower Administration Guide* at
<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► QUIZ

EXPLAINING THE RED HAT ANSIBLE TOWER ARCHITECTURE

Choose the correct answers to the following questions:

- ▶ **1. Which three of the following features are provided by Ansible Tower? (Choose three.)**
 - a. Role-based access control
 - b. Playbook creator wizard
 - c. Centralized logging
 - d. RESTful API
- ▶ **2. Which two of the following enhancements are additions to Ansible provided by Ansible Tower? (Choose two.)**
 - a. Playbook development
 - b. Remote execution
 - c. Multiplay workflows
 - d. Monitoring
 - e. Version control
 - f. Graphical inventory management
- ▶ **3. Which of the following access methods are supported in Ansible Tower? (Choose two.)**
 - a. RESTful API
 - b. Python API
 - c. Ansible Tower CLI
 - d. Ansible CLI
 - e. Web UI Dashboard
- ▶ **4. Which three of the following architectures are supported by the current Ansible Tower installer? (Choose three.)**
 - a. Single machine with integrated database
 - b. Single machine with an external database hosted on a separate server on the network
 - c. Active/active redundancy multimachine cluster with an external database
 - d. Active/passive redundancy multimachine cluster with an external database
- ▶ **5. Which two of the following features are not provided by a basic Ansible Tower license? (Choose two.)**
 - a. Tower Dashboard
 - b. Job scheduling
 - c. Clustering
 - d. Role-based access control
 - e. Logging aggregation

► SOLUTION

EXPLAINING THE RED HAT ANSIBLE TOWER ARCHITECTURE

Choose the correct answers to the following questions:

► 1. **Which three of the following features are provided by Ansible Tower? (Choose three.)**

- a. Role-based access control
- b. Playbook creator wizard
- c. Centralized logging
- d. RESTful API

► 2. **Which two of the following enhancements are additions to Ansible provided by Ansible Tower? (Choose two.)**

- a. Playbook development
- b. Remote execution
- c. Multiplay workflows
- d. Monitoring
- e. Version control
- f. Graphical inventory management

► 3. **Which of the following access methods are supported in Ansible Tower? (Choose two.)**

- a. RESTful API
- b. Python API
- c. Ansible Tower CLI
- d. Ansible CLI
- e. Web UI Dashboard

► 4. **Which three of the following architectures are supported by the current Ansible Tower installer? (Choose three.)**

- a. Single machine with integrated database
- b. Single machine with an external database hosted on a separate server on the network
- c. Active/active redundancy multimachine cluster with an external database
- d. Active/passive redundancy multimachine cluster with an external database

► 5. **Which two of the following features are not provided by a basic Ansible Tower license? (Choose two.)**

- a. Tower Dashboard
- b. Job scheduling
- c. Clustering
- d. Role-based access control
- e. Logging aggregation

INSTALLING RED HAT ANSIBLE TOWER

OBJECTIVES

After completing this section, students should be able to install Red Hat Ansible Tower in a single-server configuration.

Figure 10.1: Installing Red Hat Ansible Tower

INSTALLATION REQUIREMENTS

Ansible Tower can be installed and is supported on 64-bit x86_64 versions of Red Hat Enterprise Linux 7, CentOS 7, and Ubuntu 16.04 LTS. The following are the requirements for installing Ansible Tower on a Red Hat Enterprise Linux 7 system. Details may vary slightly for other supported operating systems.

Operating System

For Red Hat Enterprise Linux installations, the Ansible Tower 3.3 server is supported on systems running Red Hat Enterprise Linux 7.4 or later on the 64-bit x86_64 processor architecture.

Web Browser

The latest supported version of Mozilla Firefox or Google Chrome is required for connecting to the Ansible Tower web UI. Other HTML5-compliant web browsers may work but are not fully tested or supported.

Memory

A minimum of 4 GB of RAM is required on the Ansible Tower host.

The actual memory requirement depends on the maximum number of hosts that Ansible Tower is expected to configure in parallel. This is managed by the `forks` configuration parameter in the job template or system configuration. Red Hat recommends that 100 MB of memory be available for each additional fork, and 2 GB for the Ansible Tower services.

Disk Storage

At least 20 GB of hard disk space is required for Ansible Tower, and 10 GB of this space must be available for the `/var` directory.



NOTE

If you are installing in an Amazon EC2 instance, use at least the `m4.large` instance type; use `m4.xlarge` if you are managing more than 100 hosts.

Red Hat Ansible Engine

Installation of Ansible Tower is performed by executing a shell script that runs an Ansible Playbook. Earlier versions of Ansible Tower required that the latest stable version of Red Hat Ansible Engine was installed prior to installation, but the current installation program automatically attempts to install Red Hat Ansible Engine and its dependencies if they are not already present.

Red Hat Enterprise Linux 7 users must enable the **extras** repository.

For Ansible Tower to work correctly, the latest stable version of Ansible should be installed using your distribution's package manager (such as **yum**). The installation is ideally performed by the Ansible Tower installation program.

SELinux

Ansible Tower supports the **targeted** SELinux policy, which can be set to enforcing mode, permissive, or disabled. Other SELinux policies are not supported.

Managed Hosts

The installation requirements above apply to the Ansible Tower server, not to the machines it manages with Ansible. Those systems should meet the requirements for machines that are managed with the version of Ansible that is installed on the Ansible Tower server.



NOTE

If you are deploying Ansible Tower as a pod to a Red Hat OpenShift cluster, the cluster will need 6 GB of memory and 3 CPU cores per pod.

For more information on the container-based installation process, see OpenShift Deployment and Configuration [https://docs.ansible.com/ansible-tower/latest/html/administration/openshift_configuration.html] in the *Ansible Tower Administration Guide*.

RED HAT ANSIBLE TOWER LICENSING AND SUPPORT

Administrators interested in evaluating Ansible Tower can obtain a trial license at no cost. Instructions on how to get started are available at <https://www.ansible.com/tower-trial>.

Administrators interested in progressing beyond trial licensing can choose from three types of Red Hat Ansible Tower subscriptions:

- Self-support

Targeted at small deployments, this includes a basic Ansible Tower subscription, with software maintenance and upgrades but no technical support or service level agreement (SLA). Some "enterprise" features of Ansible Tower are not included. Versions supporting up to 250 managed nodes are available. Larger deployments should consider the enterprise subscriptions.

- Standard

The Standard edition includes an enterprise Ansible Tower subscription with entitlement to all Ansible Tower features and 8x5 technical support. Pricing is based on the number of nodes that are managed.

- Premium

The Premium edition also includes an enterprise Ansible Tower subscription with software maintenance and upgrades and all Ansible Tower features, but with entitlement to 24x7 technical support. Pricing is based on the number of nodes managed.

Detailed information on Ansible Tower pricing and support is available at <http://www.ansible.com>.

Licenses for Ansible Tower Components

Ansible Tower makes use of various software components, some of which may be open source. Licenses for each of these specific components are provided under the `/usr/share/doc/ansible-tower` directory.

ANSIBLE TOWER INSTALLERS

Two different installation packages are available for Ansible Tower.

The standard **setup** Ansible Tower installation program can be downloaded from <http://releases.ansible.com/ansible-tower/setup/>. The latest version of Ansible Tower for Red Hat Enterprise Linux 7 is always located at <https://releases.ansible.com/ansible-tower/setup/ansible-tower-setup-latest.tar.gz>. This archive is smaller, but requires internet connectivity to download Ansible Tower packages from various package repositories.

A different, bundled installer for RHEL 7 is available at <http://releases.ansible.com/ansible-tower/setup-bundle/ansible-tower-setup-bundle-latest.el7.tar.gz>. This archive includes an initial set of RPM packages for Ansible Tower so that it may be installed on systems disconnected from the internet. Those systems still need to be able to get software packages for Red Hat Enterprise Linux 7 and the Red Hat Enterprise Linux 7 Extras channel from reachable sources. This may be preferred by administrators in higher security environments. This installation method is not currently available for Ubuntu.

INSTALLING ANSIBLE TOWER

The following procedure applies to the bundled installer to install Ansible Tower on a single Red Hat Enterprise Linux 7.4 or later system with access to the Red Hat Enterprise Linux 7 Extras repository. The exercise after this section will go over this in more detail.

1. As the **root** user, download the Ansible Tower setup bundle to the system.
2. Extract the Ansible Tower setup bundle and change into the directory containing the extracted contents.

```
[root@towerhost ~]# tar xzf ansible-tower-setup-bundle-latest.el7.tar.gz  
[root@towerhost ~]# cd ansible-tower-setup-bundle-3.3.1-1.el7
```

3. Edit the **inventory** file to set passwords for the Ansible Tower admin account (`admin_password`), the PostgreSQL database user account (`pg_password`), and the RabbitMQ messaging user account (`rabbitmq_password`).

```
[tower]  
localhost ansible_connection=local  
  
[database]  
  
[all:vars]  
admin_password='myadminpassword'  
  
pg_host=''  
pg_port=''  
  
pg_database='awx'  
pg_username='awx'  
pg_password='somedatabasepassword'
```

```
rabbitmq_port=5672
rabbitmq_vhost=tower
rabbitmq_username=tower
rabbitmq_password='and-a-messaging-password'
rabbitmq_cookie=cookiemonster

# Needs to be true for fqdns and ip addresses
rabbitmq_use_long_name=false
```

**WARNING**

No restrictions apply to these passwords but you should set them to something secure. The `admin` user has full control of the Ansible Tower server, and the ports for PostgreSQL and the RabbitMQ service are exposed to external hosts by default.

- Run the `setup.sh` script to start the Ansible Tower installer.

```
[root@towerhost ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh
[warn] Will install bundled Ansible
...output omitted...
The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2017-02-27-10:52:44.log
```

- When the installer finishes successfully, connect to the Ansible Tower system with a web browser. You should be redirected to an HTTPS login page.
The web browser may generate a warning regarding a self-signed HTTPS certificate presented by the Ansible Tower website. Replacing the default self-signed HTTPS certificate for the Ansible Tower web UI with a properly CA-signed certificate is discussed later in this course.
- Log in to the Ansible Tower web UI as the Ansible Tower administrator with the `admin` account and the password you set in the installer's `inventory` file.
- When you log in to the Ansible Tower web UI for the first time, you are prompted to enter a license and accept the end-user license agreement. Enter the Ansible Tower license provided by Red Hat and accept the end-user license agreement.
The Ansible Tower Dashboard should now display. The next section provides an orientation to the Ansible Tower interface in more detail.

**REFERENCES****[Ansible Tower Quick Installation Guide](#)**

<https://docs.ansible.com/ansible-tower/latest/html/quickinstall/>

[Ansible Tower Installation and Reference Guide](#)

<https://docs.ansible.com/ansible-tower/latest/html/installandreference/>

[OpenShift Deployment and Configuration](#)

https://docs.ansible.com/ansible-tower/latest/html/administration/openshift_configuration.html

[Ansible Tower User Guide](#)

<https://docs.ansible.com/ansible-tower/latest/html/userguide/>

[Ansible Tower Administration Guide](#)

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

INSTALLING RED HAT ANSIBLE TOWER

In this exercise, you will install a single-node Red Hat Ansible Tower instance.

OUTCOMES

You should be able to successfully install Ansible Tower and configure it with a license, resulting in a running Ansible Tower server.

BEFORE YOU BEGIN

Log in to workstation as student using student as the password. Run **lab tower-install setup** to configure `tower.lab.example.com` with a Yum repository containing package dependencies from the Red Hat Enterprise Linux and Extras repositories required for Ansible Tower installation.

```
[student@workstation ~]$ lab tower-install setup
```

- 1. Download the Ansible Tower setup bundle to the `tower` system.

- 1.1. Log in to the `tower` system as the `root` user.

```
[student@workstation ~]$ ssh root@tower
```

- 1.2. Use the `curl` command to download the Ansible Tower setup bundle.

```
[root@tower ~]# curl -O -J \
> http://content.example.com/tower3.3/x86_64/dvd/setup-bundle/ansible-tower-setup-
bundle.el7.tar.gz
```

- 2. Extract the Ansible Tower setup bundle. Change to the directory which contains the extracted contents.

```
[root@tower ~]# tar xzf ansible-tower-setup-bundle.el7.tar.gz
[root@tower ~]# cd ansible-tower-setup-bundle-3.3.1-1.el7
```

- 3. Edit the `inventory` file and set the passwords for the Ansible Tower administrator account, database user account, and messaging user account to `redhat`. This file is used by the Ansible Tower installer playbook.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# vi inventory
admin_password='redhat'
pg_password='redhat'
rabbitmq_password='redhat'
```

- ▶ 4. Run the Ansible Tower installer by executing the **setup.sh** script. The script may take up to 15 minutes to complete. Ignore the errors in the script output because they are related to verification checks performed by the installer playbook.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh  
[warn] Will install bundled Ansible  
...output omitted...  
The setup process completed successfully.  
Setup log saved to /var/log/tower/setup-2018-10-18-10:52:44.log
```

- ▶ 5. When the installer finishes successfully, exit the console session on the **tower** system.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# exit
```

- ▶ 6. Launch the Firefox web browser on **workstation** and navigate to Ansible Tower at <https://tower.lab.example.com>. Firefox warns you that the Ansible Tower server's security certificate is not secure. Add and confirm the security exception for the self-signed certificate.
- ▶ 7. Log in to the Ansible Tower web UI as **admin** using **redhat** as the password.
- ▶ 8. When you log in to the Ansible Tower web UI for the first time, you have to enter a license and accept the end-user license agreement.
Upload the Ansible Tower license and accept the end-user license agreement.
- 8.1. On **workstation**, download the Ansible Tower license provided at <http://materials.example.com/tower/install/Ansible-Tower-license.txt>.
 - 8.2. In the Ansible Tower web UI, click **BROWSE** and then select the license file that you downloaded earlier.
 - 8.3. Select the check box next to **I agree to the End User License Agreement**.
 - 8.4. Click **SUBMIT** to submit the license and accept the license agreement.

This concludes the guided exercise.

NAVIGATING RED HAT ANSIBLE TOWER

OBJECTIVES

After completing this section, students should be able to navigate and describe the Ansible Tower web UI, and successfully launch a job using the demo job template, project, credentials, and inventory.

Figure 10.1: Navigating Red Hat Ansible Tower

USING ANSIBLE TOWER

This section provides an overview of how to use the Ansible Tower web UI to launch a job with an example Ansible Playbook, an inventory, and access credentials for the machines in the inventory. It also provides an orientation to the web UI itself.

The basic idea is that Ansible Tower is configured with a number of Ansible *projects* that contain playbooks. It is also configured with a number of Ansible *inventories* and the necessary machine *credentials* to log in to inventory hosts and escalate privileges. A *job template* is set up by an administrator and specifies which playbook from which project should be run on the hosts in a particular inventory using particular machine credentials. A *job* is performed when a user runs a playbook on an inventory by launching a job template.

THE ANSIBLE TOWER DASHBOARD

When you log in to the web UI you see the Ansible Tower Dashboard, the main control center for Ansible Tower.

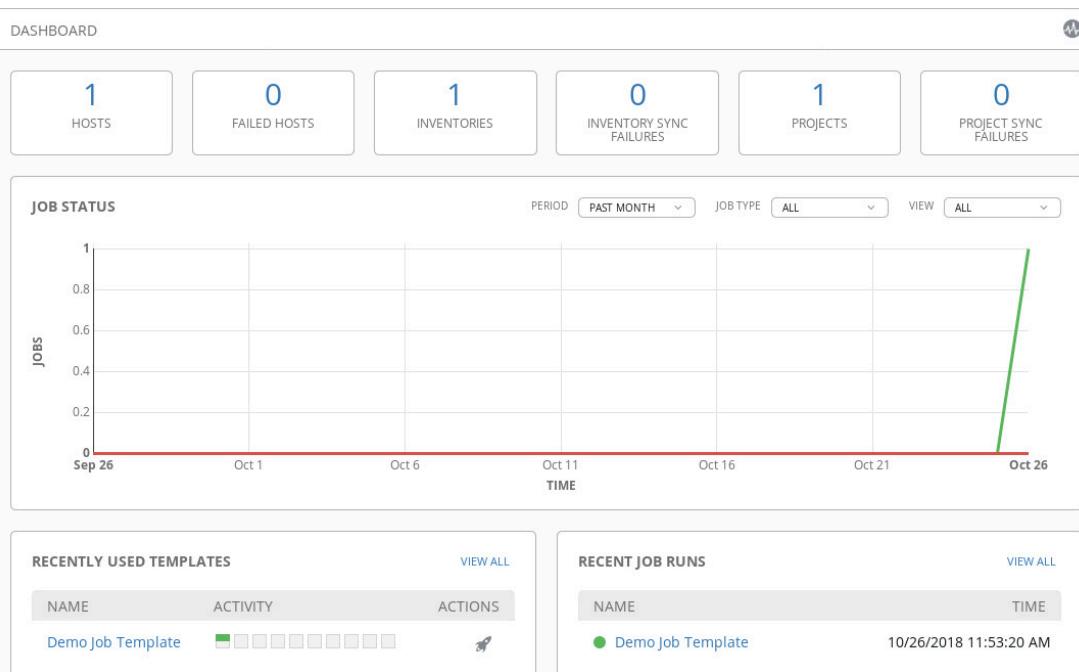


Figure 10.2: The Ansible Tower Dashboard

The Dashboard is composed of four reporting sections:

Summary

Across the top of the Dashboard is a summary report of the status of managed hosts, inventories, and Ansible projects. You can click a cell in the summary section to view the detailed Dashboard page for the reported metric.

Job Status

A *job* is an attempted run of a playbook by Ansible Tower. This section provides a graphical display of the number of successful and failed jobs over time. This graph can be adjusted in several ways:

- Use the PERIOD menu to change the time window for the plotted graph.
- Use the JOB TYPE menu to select which job types to include on the graph.
- Use the VIEW menu to choose between graphing the status of all jobs, only failed jobs, or only successful jobs.

Recently Used Templates

This section displays a list of job templates that were recently used for the execution of jobs.

- For each job template used, the results of each associated job run are indicated under the ACTIVITY column by a small colored square, with green indicating success and red indicating failure. You can mouse over a colored square to display the job ID number and when it was run. You can click a square to open the Job Details view for that job. This is discussed in more detail later in this section.
- Under the ACTIONS column are controls for the use of the job template.
- You can click the VIEW ALL link to display all job templates, not just those that have recently been used for job execution.

Recent Job Runs

This section displays a list of recently executed jobs and the date and time they were executed. Each job run is preceded by a colored dot which represents the outcome of the run. A green dot represents a successful run; a red dot represents a failed run. You can click a job run to display the Job Details view for that job.

NAVIGATION BAR

In the left portion of the Ansible Tower web UI is a collection of navigation links to commonly used Ansible Tower resources. The Ansible Tower icon takes you to the Ansible Tower Dashboard. The other links provide access to the administrative page for each of the Ansible Tower resources. These are described below.

Jobs

A job represents a single run of an Ansible Playbook by Ansible Tower against an inventory of hosts.

Templates

A template defines parameters to use to launch a job (to run an Ansible Playbook) with Ansible Tower.

Credentials

Use this interface to manage credentials. Credentials are authentication data used by Ansible Tower to log in to managed hosts to run plays, decrypt Ansible Vault files, synchronize inventory data from external sources, download updated project materials from version control systems, and similar tasks.

Projects

In Ansible Tower, a project represents a collection of related Ansible Playbooks.

Inventories

Inventories contain a collection of hosts to be managed.

Inventory Scripts

Use this interface to manage scripts that generate and update dynamic inventories from external sources, such as cloud providers and configuration management databases (CMDBs).

Organizations

Use this interface to manage organization entities within Ansible Tower. An organization represents a logical collection of other Ansible Tower resources, such as Teams, Projects, and Inventories. Organizations are often used for departmental separation within an enterprise. An organization is the highest level at which the Ansible Tower role-based access control system can be applied.

Users

Use this interface to manage Ansible Tower users. Users are granted access to Ansible Tower and then assigned roles which determine their access to Ansible Tower resources.

Teams

Use this interface to administer Ansible Tower teams. Teams represent a group of Ansible Tower users. Like users, teams can also be assigned roles for access to Ansible Tower resources.

Notifications

Use this interface to manage notification templates. These templates define the set of configuration parameters needed to generate notifications using a variety of message delivery tools, such as email, IRC, and HipChat.

Management Jobs

Use this interface to manage system jobs, which clean up old data from Ansible Tower operations. This includes job history, and activity streams. You might need to do this to control the amount of storage used for this information on your Ansible Tower server, or to comply with your organization's data retention requirements.

ADMINISTRATION TOOL LINKS

The upper-right portion of the Ansible Tower web UI contains links to various Ansible Tower administration tools.



Figure 10.3: Administration tool links

Account Configuration

The current user account name is displayed as a link. You can click the account name to access the account configuration page.

About

The information icon displays the installed version of Ansible Tower, and the version of Ansible it is using.

View Documentation

You can click the book icon to display the Ansible Tower documentation web site in a new window.

Log Out

Use the power icon to log out of the Ansible Tower web UI.

TOWER SETTINGS

Click **Settings** in the left navigation bar to access the Ansible Tower Settings page. Use the buttons at the top of the page to switch between different categories of settings.

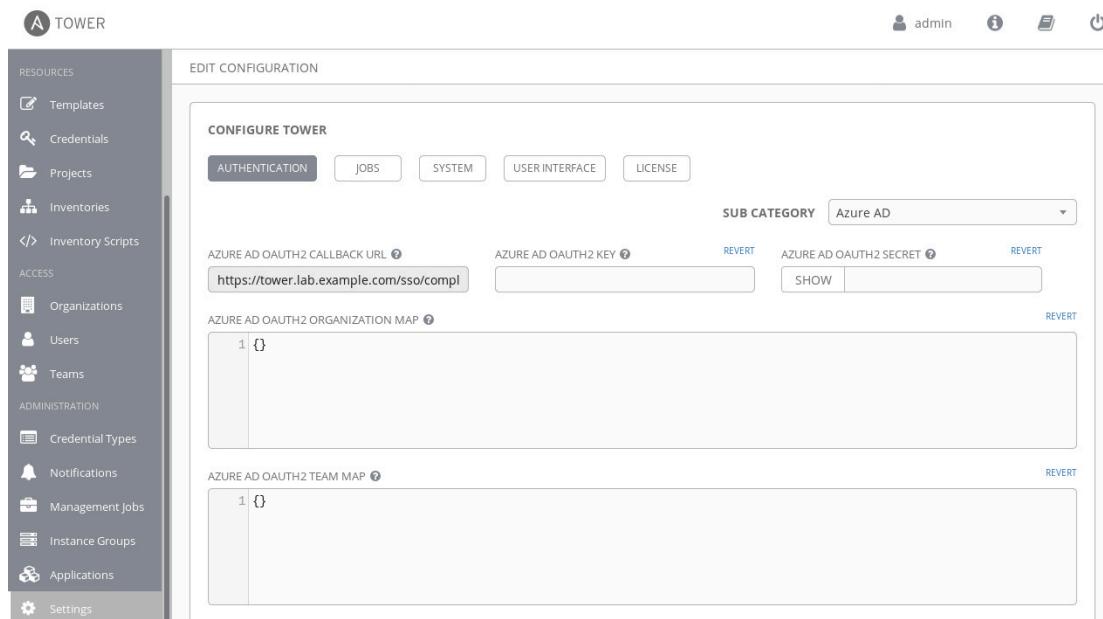


Figure 10.4: The Settings page in Ansible Tower

The different categories available on the **Settings** page are described below.

Authentication

The Authentication category contains settings used to configure simplified authentication to user accounts in Ansible Tower using third-party login information, such as LDAP, Azure Active Directory, GitHub, or Google OAuth2. Select the type of login information to configure from the **SUB CATEGORY** menu.

Jobs

The Jobs category contains advanced settings used to configure job execution. You can use this to control how many scheduled jobs may be set up by users, the Ansible modules allowed for ad hoc jobs launched by Ansible Tower, and timeouts for project updates, fact caching, and job runs.

System

The System category contains advanced settings that you can use to configure log aggregation, activity stream settings, and other miscellaneous Ansible Tower options.

User Interface

The User Interface category allows you to configure analytics reporting, and to set a custom logo or custom login messages for the Ansible Tower server.

License

This interface provides details of the installed license and can also be used to perform administrative licensing tasks such as license installation and upgrade.

GENERAL CONTROLS

In addition to the navigational and administrative controls previously outlined, some additional controls are used throughout the Ansible Tower web UI.

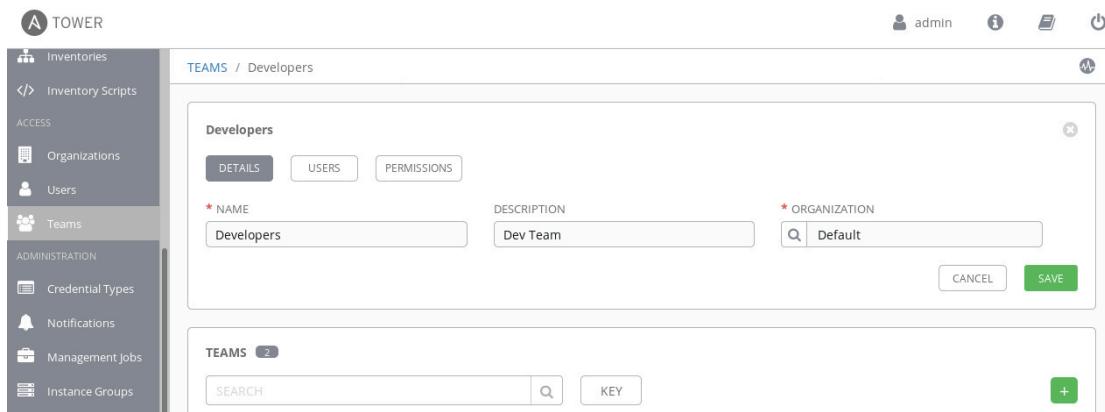


Figure 10.5: Breadcrumb Navigation and activity streams links in teams

Breadcrumb Navigation Links

As you navigate through the Ansible Tower web UI a "breadcrumb" trail is created in the upper-left corner of the page. This clearly identifies the path to each page and also provides a quick way to return to previous pages.

Activity Streams

Under the Logout icon in the upper-right corner of most pages in the Ansible Tower web UI you can see the View Activity Stream icon. It appears as a circular icon containing a small graph. Click this icon to display a report of activities related to the current page. For example, if you click the View Activity Stream icon on the Projects page, a list of project-related events is displayed, including the event time and the user who initiated it.

Search Fields

Search fields exist throughout the Ansible Tower web UI which can be used to search or filter through data sets.

Click the Key button next to each search field to display a guide which describes the correct syntax of the specific criteria for each field. You can also use the Key button to define the options provided by other input fields within Ansible Tower.

CONFIGURING JOB TEMPLATES

Ansible Tower is preconfigured with a demonstration job template. You can use this template as an example of how a job template is constructed and to test the operation of Ansible Tower. The following discussion examines the components that make up this example. The exercise that follows this section provides a more detailed hands-on walk-through.

1. Under Inventories in the left navigation bar, an inventory called Demo Inventory has been configured. This is a static inventory with no host groups and one host, called localhost. Clicking on that host in the inventory reveals that it has the inventory variable `ansible_connection: local` set.

Figure 10.6: Details of Demo Inventory

2. Under **Credentials** in the left navigation bar, a machine credential named **Demo Credential** has been created. This credential contains information that could be used to authenticate access to machines in an inventory.
3. Under **Projects** in the left navigation bar, a project called **Demo Project** has been configured. This project is configured to get Ansible project materials, including playbooks, from a local directory on the Ansible Tower system.

Figure 10.7: Details of Demo Project

4. Under **Templates** in the left navigation bar, a template called **Demo Job Template** has been configured. This job template is configured as a normal playbook run (**Job Type** is **Run**), using the **hello_world.yml** playbook from **Demo Project**.

This job template runs on the machines in **Demo Inventory**, using **Demo Credential** to authenticate to those machines. Privilege escalation is not enabled because **hello_world.yml** does not need it. Were it needed, the **Demo Credential** option would need to provide the necessary information. Notice that because the job template is not using

privilege escalation and is running only on machines using `ansible_connection: local`, there is very little information needed in `Demo Credential`.

No extra variables are set (analogous to the `-e` or `--extra-vars` option of an `ansible-playbook` command).

The screenshot shows the Ansible Tower web interface. On the left, a navigation sidebar lists 'TOWER' at the top, followed by 'VIEWS' (Dashboard, jobs, Schedules, My View), 'RESOURCES' (Templates, Credentials, Projects, Inventories, Inventory Scripts, Organizations, Users, Teams), and 'ADMINISTRATION'. The 'TEMPLATES' section is selected. In the main content area, the title 'TEMPLATES / Demo Job Template' is shown above a form for 'Demo Job Template'. The form has tabs for 'DETAILS' (selected), 'PERMISSIONS', 'NOTIFICATIONS', 'COMPLETED JOBS', 'SCHEDULES', and 'ADD SURVEY'. The 'DETAILS' tab contains fields for 'NAME' (Demo Job Template), 'DESCRIPTION', 'JOB TYPE' (Run), 'INVENTORY' (Demo Inventory), 'PROJECT' (Demo Project), 'PLAYBOOK' (hello_world.yml), 'CREDENTIAL' (Demo Credential), 'FORKS' (DEFAULT), 'LIMIT', 'VERBOSITY' (0 (Normal)), 'JOB TAGS', 'SKIP TAGS', 'LABELS', 'INSTANCE GROUPS', 'OPTIONS' (Enable Privilege Escalation, Allow Provisioning Callbacks), and 'SHOW CHANGES' (OFF). At the bottom right of the form is a large blue 'SAVE' button.

Figure 10.8: Demo Job Template

LAUNCHING A JOB

After you have created and configured a job template, you can use it to repeatedly launch jobs using the same parameters. A job template is like a saved `ansible-playbook` command complete with options and arguments. When you launch a job template, it is like repeating an `ansible-playbook` command from your shell history.

The following discussion examines what happens when you use `Demo Job Template` to launch a job. This is covered in more detail in an upcoming exercise.

- Under `Templates` in the left navigation bar, the `Demo Job Template` should be listed. Across from the name of the job template is a rocket icon ("Start a job using this template"). Clicking that icon launches the job using the settings in the job template.

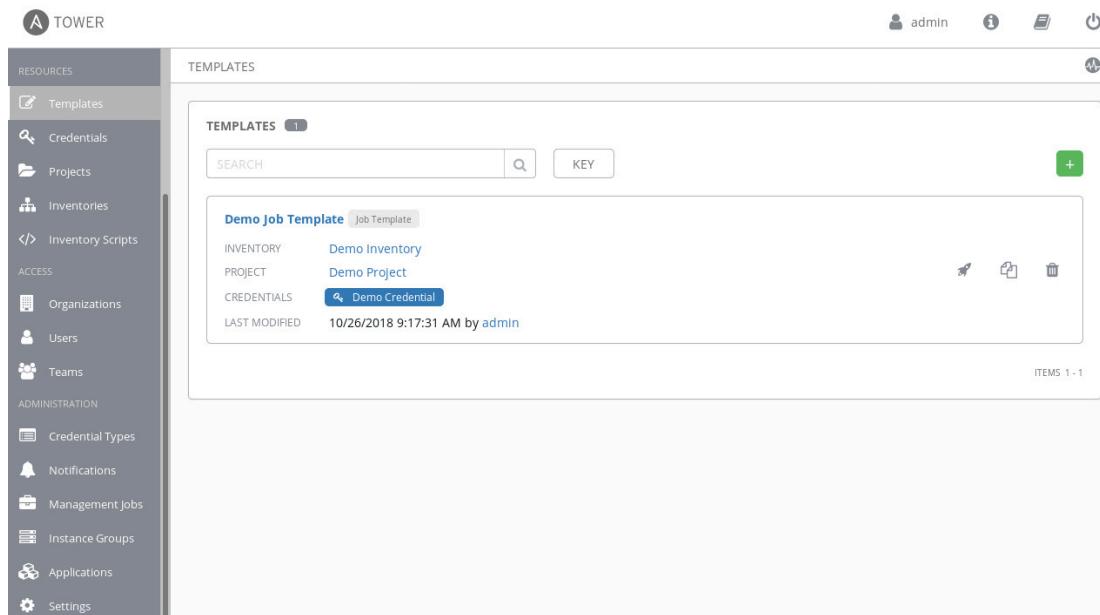


Figure 10.9: Launching a job

2. As the job runs, a new status page opens that provides real-time information about the progress of the job. The DETAILS pane provides basic information about the job being run: when it was launched, who launched it, what job template, project, and inventory were used, and so on. While running, the job status is **Pending**.
3. As the job executes, the status page also includes the output from the job run in a job output pane. The job run output resembles the output generated when you use the **ansible-playbook** command to run an Ansible Playbook. This can be used for troubleshooting purposes. In the following example, the play and tasks in the playbook ran successfully.

```

PLAY [Hello World Sample] ****
*****
2 TASK [Gathering Facts] ****
*****
4 ok: [localhost]
6
7 TASK [Hello Message] ****
*****
8 ok: [localhost] => {
9   "msg": "Hello World!"
10 }

```

Figure 10.10: Example job output

4. After the job completes, the Ansible Tower Dashboard provides a link to the status page for the job run under RECENT JOB RUNS. The other statistics on the Ansible Tower Dashboard are also updated.
5. Under **Jobs** in the left navigation bar, all the jobs that have run on Ansible Tower are listed. You can click the name of the job listed on this page to display the status page logged for that job.

The screenshot shows the Ansible Tower interface. On the left is a sidebar with icons for Dashboard, Jobs (which is selected), Schedules, My View, Templates, Credentials, Projects, Inventories, and Inventory Scripts. The main area is titled 'JOBS' and shows one job entry: '1 - Demo Job Template' (Playbook Run). Below the job title, it says 'STARTED 10/26/2018 11:53:09 AM' and 'FINISHED 10/26/2018 11:53:20 AM'. It was 'LAUNCHED BY admin'. The 'JOB TEMPLATE' is 'Demo Job Template', 'INVENTORY' is 'Demo Inventory', 'PROJECT' is 'Demo Project', and 'CREDENTIALS' is 'Demo Credential'. There are edit and delete icons next to the job entry. At the bottom right of the main area, it says 'ITEMS 1 - 1'.

Figure 10.11: Example jobs page



REFERENCES

Ansible Tower Quick Setup Guide

<https://docs.ansible.com/ansible-tower/latest/html/quickstart/>

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide/>

Ansible Tower Administration Guide

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

ACCESSING RED HAT ANSIBLE TOWER

In this exercise, you will navigate through the Ansible Tower web UI and launch a job.

OUTCOMES

You should be able to browse through and interact with the project, inventory, credential, and template pages in the Ansible Tower web UI to successfully launch a job.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the `student` user on `workstation` and run `lab tower-webui setup`. This setup script configures the `tower` virtual machine for the exercise.

```
[student@workstation ~]$ lab tower-webui setup
```

- ▶ 1. Log in to the Ansible Tower web UI running on `tower` as `admin` using `redhat` as the password.
- ▶ 2. Identify the project that was created by the script and determine its source.
 - 2.1. Click **Projects** in the left navigation bar to display the list of projects. You should see a project named **Demo Project**, which was created during the Ansible Tower installation.
 - 2.2. Click the **Demo Project** link to view the details of the project.
 - 2.3. Look at the values of the **SCM TYPE**, **PROJECT BASE PATH**, and **PLAYBOOK DIRECTORY** fields to determine the origin of the project. You should see that **Demo Project** was obtained from the `/var/lib/awx/projects/_4_demo_project` directory on the Ansible Tower system.
- ▶ 3. Browse the inventory that was created during the Ansible Tower installation and determine its managed hosts.
 - 3.1. Click **Inventories** in the left navigation bar to display the list of known inventories. You should see an inventory named **Demo Inventory**, which was created during the Ansible Tower installation.
 - 3.2. Click the **Demo Inventory** link to view the details of the inventory. Click **HOSTS** and notice that the inventory includes just one host: `localhost`.
- ▶ 4. View the details of the credential that was created during the Ansible Tower installation.
 - 4.1. Click **Credentials** in the left navigation bar to display the list of credentials.

- 4.2. Click the **Demo Credential** link to view the details of the credential. You should see that the credential is a machine credential that uses the username **admin**.
- ▶ 5. Identify the job template that was created during the Ansible Tower installation. Click **Templates** in the left navigation bar to display the list of existing job templates. You should see a job template named **Demo Job Template**, which was created during the Ansible Tower installation.
- ▶ 6. Determine the inventory, project, and credential used by the job template, **Demo Job Template**.
 - 6.1. Click the **Demo Job Template** link to view the details of the job template.
 - 6.2. Look at the **INVENTORY** field. You should see that the job template uses the **Demo Inventory** inventory.
 - 6.3. Look at the **PROJECT** field. You should see that the job template uses the **Demo Project** project.
 - 6.4. Look at the **PLAYBOOK** field. You should see that the job template executes a **hello_world.yml** playbook.
 - 6.5. Look at the **CREDENTIAL** field. You should see that the job template uses the **Demo Credential** credential.
- ▶ 7. Launch a job using the **Demo Job Template** job template.
 - 7.1. Click **Templates** in the left navigation bar to exit the template details view.
 - 7.2. Click the rocket icon in the same row as the **Demo Job Template** template. This launches a job using the parameters configured in the **Demo Job Template** template and redirects you to the job details page. As the job executes, the details of the job execution and its output are displayed.
- ▶ 8. Determine the outcome of the job execution.
 - 8.1. When the job has completed successfully, the **STATUS** value changes to **Successful**.
 - 8.2. Review the output of the job execution to determine which tasks were executed. You should see that the **msg** module was used to successfully display a **Hello World!** message.
- ▶ 9. Review the changes to the Dashboard reflecting the job execution.
 - 9.1. Click **TOWER** in the upper-left corner of the page to return to the Dashboard.
 - 9.2. Review the **JOB STATUS** graph. The green line on the graph indicates the number of recent successful job executions.
 - 9.3. Review the **RECENT JOB RUNS** section. This section provides a list of the jobs recently executed as well as their results. The **Demo Job Template** entry indicates that the job template was used to execute a job. The green dot at the beginning of the entry indicates the successful completion of the executed job.

This concludes the guided exercise.

► QUIZ

INSTALLING AND ACCESSING RED HAT ANSIBLE TOWER

Choose the correct answers to the following questions:

- ▶ **1. Which three of the following are components of the Red Hat Ansible Tower 3.3 architecture? (Choose three.)**
 - a. Django web application
 - b. MongoDB database
 - c. PostgreSQL database
 - d. RabbitMQ messaging system
- ▶ **2. Which three of the following statistics are reported by the Ansible Tower Dashboard? (Choose three.)**
 - a. Number of inventories
 - b. Number of credentials stored
 - c. Status of executed jobs
 - d. Number of hosts managed
- ▶ **3. Which of the following is not a requirement for a Red Hat Ansible Tower 3.3 installation?**
 - a. An existing installation of Ansible
 - b. SELinux targeted policy in enforcing, permissive, or disabled mode
 - c. A minimum of 4 GB of RAM for a server-based installation
 - d. A minimum of 20 GB of hard disk space
- ▶ **4. Which two of the following statements regarding the bundled Red Hat Ansible Tower installer are incorrect? (Choose two.)**
 - a. Includes additional software dependencies needed by Ansible Tower
 - b. Requires access to third-party repositories to meet software dependencies
 - c. Requires access to Red Hat Enterprise Linux 7 Extras software channel to meet software dependencies
 - d. Supports installations on systems running Ubuntu
- ▶ **5. Which of the following resources cannot be accessed directly through the left navigation bar?**
 - a. Projects
 - b. Inventories
 - c. Activity Stream
 - d. Templates

► SOLUTION

INSTALLING AND ACCESSING RED HAT ANSIBLE TOWER

Choose the correct answers to the following questions:

- ▶ **1. Which three of the following are components of the Red Hat Ansible Tower 3.3 architecture? (Choose three.)**
 - a. Django web application
 - b. MongoDB database
 - c. PostgreSQL database
 - d. RabbitMQ messaging system
- ▶ **2. Which three of the following statistics are reported by the Ansible Tower Dashboard? (Choose three.)**
 - a. Number of inventories
 - b. Number of credentials stored
 - c. Status of executed jobs
 - d. Number of hosts managed
- ▶ **3. Which of the following is not a requirement for a Red Hat Ansible Tower 3.3 installation?**
 - a. An existing installation of Ansible
 - b. SELinux targeted policy in enforcing, permissive, or disabled mode
 - c. A minimum of 4 GB of RAM for a server-based installation
 - d. A minimum of 20 GB of hard disk space
- ▶ **4. Which two of the following statements regarding the bundled Red Hat Ansible Tower installer are incorrect? (Choose two.)**
 - a. Includes additional software dependencies needed by Ansible Tower
 - b. Requires access to third-party repositories to meet software dependencies
 - c. Requires access to Red Hat Enterprise Linux 7 Extras software channel to meet software dependencies
 - d. Supports installations on systems running Ubuntu
- ▶ **5. Which of the following resources cannot be accessed directly through the left navigation bar?**
 - a. Projects
 - b. Inventories
 - c. Activity Stream
 - d. Templates

SUMMARY

In this chapter, you learned:

- Red Hat Ansible Tower is a centralized management solution for Ansible projects.
- Ansible Tower is offered with two installer options. The standard installer downloads packages from network repositories. The bundled installer includes third-party software dependencies.
- Job templates are prepared commands to execute Ansible Playbooks. Important components of a job template include an inventory, a machine credential, an Ansible project, and a playbook.
- A job is launched from a job template, and represents a single run of a playbook on an inventory of machines.
- The Ansible Tower Dashboard provides summaries of the status of hosts, inventories, projects, and executed jobs.
- The navigation bar at the left side of the Ansible Tower web UI provides access to commonly used Ansible Tower resources.

CHAPTER 11

MANAGING ACCESS WITH USERS AND TEAMS

GOAL

Create user accounts and organize them into teams in Red Hat Ansible Tower. Assign them permissions to administer and access resources in the Ansible Tower service.

OBJECTIVES

- Create new users in the web user interface and explain different types of users in Ansible Tower
- Create new teams in the web user interface, assign users to them, and explain the different roles that can be assigned to users
- Creating and Managing Ansible Tower Users (and Guided Exercise)
- Managing Users Efficiently with Teams (and Guided Exercise)

SECTIONS

LAB

Managing Access with Users and Teams

CREATING AND MANAGING ANSIBLE TOWER USERS

OBJECTIVES

After completing this section, students should be able to create new users in the web UI and explain the different types of users in Ansible Tower.

Figure 11.0: Creating Ansible Tower users

ROLE-BASED ACCESS CONTROLS

Different people using an Ansible Tower installation need to have different levels of access. Some may simply need to run existing job templates against a preconfigured inventory of machines. Others need to be able to modify particular inventories, job templates, and playbooks. Still others may need access to change anything in the Tower installation.

The Tower interface has a built-in administrative user, `admin`, that has superuser access to the entire Tower configuration. Setting up user accounts for each person makes it easier to manage individual access to Inventories, Credentials, Projects, and Job Templates.

Users are assigned *roles* granting permissions that define who can see, change, or delete an object in Tower. This is managed with Role-Based Access Controls (RBAC). Roles can be managed collectively by granting them to a Team, which is a collection of users. All users in a team inherit the team's roles.

Roles determine whether users and teams can see, use, change, or delete objects such as Inventories and Projects.

ORGANIZATIONS

An Ansible Tower Organization is a logical collection of Teams, Projects, and Inventories. All Users must belong to an Organization.

One of the benefits of implementing Tower is the ability to share Ansible roles and playbooks across departmental or functional boundaries within an enterprise. For example, an operations group of an organization may already have Ansible roles for provisioning production web, database, and application servers, which the developers group should use to provision servers to prepare their development environment. Tower makes it easier for different users and groups to use existing roles and playbooks.

However, for very large deployments it can be useful to categorize large numbers of Users, Teams, Projects, and Inventories under one umbrella Organization. It may be that certain departments do not deploy to certain inventories of hosts, or run certain playbooks. By using Organizations, a collection of Users and Teams can be configured to work with only the Tower resources that they're expected to use.

As part of the Tower installation, a Default Organization is created. The following is the procedure for creating additional Organizations using the Tower web interface.

- Log in to the Tower web interface as the Tower `admin` user.
- Click on the **Organizations** link located on the left navigation bar.
- Click on the **+** button to create a new Organization.

- In the provided fields, enter a name for the new Organization and, if desired, an optional description.

The screenshot shows a modal dialog titled "NEW ORGANIZATION". At the top, there are three tabs: "DETAILS" (which is selected and highlighted in dark grey), "USERS", and "NOTIFICATIONS". Below the tabs are three input fields: a required field labeled "* NAME" containing a placeholder "Organization Name", an optional "DESCRIPTION" field containing a placeholder "Description", and a "INSTANCE GROUPS" search field with a magnifying glass icon. At the bottom right of the dialog are two buttons: "CANCEL" and "SAVE".

Figure 11.1: Fill the info of the new Organization

- Click SAVE to finalize the creation of the new Organization.



IMPORTANT

Users of an Ansible Tower that has a Self-Support subscription only have the Default Organization available and are not able to add additional Organizations. Users of Self-Support subscriptions should not delete the default Organization and try to add a new Organization to avoid breaking their Tower configuration.

For Self-Support deployments of Ansible Tower, assign all objects that need an Organization to the Default Organization.

USER TYPES

By default, the Ansible Tower installer creates an **admin** user account with full control of the Tower installation. Using the special **admin** account, a Tower administrator can log in to the Tower web interface and create additional users.

The three *user types* in Tower are:

System Administrator

The System Administrator user type (also known as *Superuser*) provides unrestricted access to perform any action within the entire Tower installation. **System Administrator** is a special *singleton role*, which has read-write permission on all objects in all Organizations on the Tower.

The **admin** user created by the installer also has the System Administrator singleton role and should therefore only be used by the Tower administrator.

System Auditor

The System Auditor user type also has a special singleton role, which has read-only access to the entire Tower installation.

Normal User

Normal User is the standard user type. It initially has no special roles assigned, and starts with minimal access. It is not assigned any singleton roles, and is only assigned roles associated with the Organization of which the user is a member.

CREATING USERS

A user logged in as the Tower **admin** can create new users by executing the following procedure:

- Click on the **Users** link located on the left navigation bar.
- Click on the **+** button to create a new user.

- Enter the first name, last name, and email address for the new user into the FIRST NAME, LAST NAME, and EMAIL fields, respectively.
- In the USERNAME field, specify a unique username for the new user.
- Click the magnifying glass icon next to the ORGANIZATION field to display a list of Organizations within Tower. Select an Organization from the list and click SAVE.
- Enter the desired password for the new user into the PASSWORD and CONFIRM PASSWORD fields.
- Select a user type.
- Click SAVE to finish.

The screenshot shows the 'CREATE USER' dialog box. At the top, there are tabs for 'DETAILS' (which is selected), 'ORGANIZATIONS', 'TEAMS', and 'PERMISSIONS'. The 'DETAILS' tab contains fields for 'FIRST NAME', 'LAST NAME', 'EMAIL', 'USERNAME', 'CONFIRM PASSWORD', and 'USER TYPE' (set to 'Normal User'). To the right of the 'EMAIL' and 'USERNAME' fields are search icons. Below these fields is a dropdown for 'USER TYPE'. At the bottom right of the dialog are 'CANCEL' and 'SAVE' buttons.

Figure 11.2: Fill the info for the new user

EDITING USERS

Use the Edit User screen to edit the properties of the newly created user by executing the following procedure:

- Click the Users link on the left navigation bar.
- Click on the link for the user to edit.
- Make the changes to the desired fields.
- Click SAVE to finish.

ORGANIZATION ROLES

Newly created users inherit specific roles from their Organization, based on their user type. You can assign additional roles to a user after creation to grant permissions to view, use, or change other Tower objects. An Organization is itself one of these objects. There are four roles that users can be assigned on an Organization:

Admin

When assigned the Admin role on an Organization, a user gains the ability to manage all aspects of that Organization, including reading and changing the Organization, and adding and removing Users and Teams from the Organization.

A number of related administrative roles exist that grant more limited access than Admin:

Project Admin

Can create, read, update and delete any project in the Organization. In conjunction with the Inventory Admin permission, this allows users to create Job Templates.

Inventory Admin

Can create, read, update and delete any inventory in the Organization. In conjunction with the Job Template Admin and Project Admin roles, this allows the user full control over job templates within the organization.

Credential Admin

Can create, read, update and delete shared credentials.

Notification Admin

Can be assigned notifications.

Workflow Admin

Can create workflows within the Organization.

Job Template Admin

Can make changes to nonsensitive fields on Job Templates. To make changes to fields that impact job runs, the user also needs the Admin role on the Job Template, the Use role on the related project, and the Use role in the related inventory.

Auditor

When assigned the Auditor role on an Organization, a user gains read-only access to the Organization.

Member

When assigned the Member role on an Organization, a user gains read permission to the Organization. The Organization Member role only provides a user the ability to view the list of users who are members of the Organization and their assigned Organization roles.

Unlike the Organization Admin and Auditor roles, the Member role does not provide users permissions to any of the resources the Organization contains, such as Teams, Credentials, Projects, Inventories, Job Templates, Work Templates, and Notifications.

Read

When assigned the Read role on an Organization, a user gains read permission to the Organization only. The Organization Read role only provides a user with the ability to view the list of users who are members of the Organization and their assigned Organization roles.

Unlike the Organization Admin and Auditor roles, the Read role does not inherit roles on any of the resources the Organization contains, such as Teams, Credentials, Projects, Inventories, Job Templates, Work Templates, and Notifications.

The Organization object cannot be assigned roles on Tower resources. Therefore, a user that has the Member role on an Organization only has access to the Organization object and inherits no other permissions as a result of this role. Consequently, a user that has the Member role on an Organization is the equivalent of a user that has the Read role on an Organization.

Execute

When assigned the Execute role on an Organization, a user gains permission to execute Job Templates and Workflow Job Templates in the Organization.

**IMPORTANT**

Users with the System Administrator singleton role inherit the Admin role on every Organization within Tower.

Users with the System Auditor singleton role inherit the Auditor role on every Organization within Tower.

A user created with the Normal User user type is assigned a Member role on the Organization they were assigned to at the time of the user's creation in Tower. Other roles can be added later, including additional Member roles on other Organizations.

Managing User Organization Roles

The Edit User screen only allows changes to user type and the Organization a user belongs to. Since the user types designated to users determine their inherited Organization roles, modifying a user's Organization and user type can impact their Organization role.

However, full administration of a user's role in an Organization requires the following procedure:

- Log in to the Tower web interface as the Tower admin or any user with the Admin role on the Organization being modified.
- Click on the **Organizations** link on the left navigation bar.
- Click on the **USERS** link under the organization being managed.

A list of users who have been granted roles to the Organization appears.

If a user does not appear on the list and needs a role in the Organization, or if a user exists and needs additional Organization roles, use the following procedure:

- Click the **+** button.

The screenshot shows two overlapping interface sections. The top section is titled 'Default' and contains a 'USERS' tab. It lists two users: 'admin' with a 'SYSTEM ADMINISTRATOR' role and 'jdoe' with a 'MEMBER' role. Below this is a 'ROLE' section. The bottom section is titled 'ORGANIZATIONS' and lists several categories: 'Default', 'USERS', 'INVENTORIES', 'JOB TEMPLATES', 'TEAMS', 'PROJECTS', and 'ADMINS'. Each category has a small icon and a count of items (e.g., 1 for USERS, 1 for INVENTORIES, etc.). A green '+' button is visible in both sections.

Figure 11.3: Select the user to assign Organization roles to

- In the ADD USERS screen, check the box next to the desired user.
- Click the **SELECT ROLES** drop-down and select the desired Organization role for the user. This step can be repeated multiple times to add multiple roles for a user.

NOTE

For a list of each role's definition, click the KEY button.

DEFAULT | ADD USERS

- 1 Please select Users from the list below.

SEARCH	FIRST NAME	LAST NAME	KEY
USERNAME	FIRST NAME	LAST NAME	
<input type="checkbox"/> admin			
<input type="checkbox"/> sam	Sam	Simons	
<input checked="" type="checkbox"/> simon	Simon	Stephens	
<input checked="" type="checkbox"/> sylvia	Sylvia	Simons	
ITEMS 1 - 4			

2 Please assign roles to the selected users/teams

Sylvia Simons <small>USER</small>	<input type="checkbox"/> Admin	X
Simon Stephens <small>USER</small>	<input type="checkbox"/> Member	X

CANCEL SAVE

Figure 11.4: Set the roles to assign on Organization

- Click SAVE to assign roles to the user for the Organization.
- Click the X preceding to the role to remove existing roles from a user.

**REFERENCES**

Ansible Tower User Guide

<http://docs.ansible.com/ansible-tower/latest/html/userguide>

► GUIDED EXERCISE

CREATING AND MANAGING ANSIBLE TOWER USERS

In this exercise, you will create user accounts of different types and explore the different levels of access of those account types.

OUTCOMES

You should be able to create user accounts and describe the access provided by different account types.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

- ▶ 1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
- ▶ 2. Create a user, `sam`, as a Normal User.
 - 2.1. Go to Users in the left navigation bar.
 - 2.2. Click the `+` button to add a new user.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Sam
LAST NAME	Simons
ORGANIZATION	Default
EMAIL	sam@lab.example.com
USERNAME	sam
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

- 2.4. Click `SAVE` to create the new User.

- ▶ 3. Verify the permissions for the newly created `sam` user.
- 3.1. Click **PERMISSIONS** to see the user's permissions.
 - 3.2. As you can see, `sam` is simply a Member of the **Default Organization**.
 - 3.3. Click the **Log Out** icon in the top right corner to log out and then log back in as `sam` with password of `redhat123`.
 - 3.4. In the left navigation bar, you can see the user's access is limited.
 - 3.5. Click **Users** in the left navigation bar to manage user permission. As you can see, it is not possible for `sam` to add new users.
 - 3.6. Click the **Log Out** icon to log out of the Tower web interface.
- ▶ 4. Create a user, `sylvia`, as a System Auditor.
- 4.1. Log in to the Tower web interface as the `admin` user with the `redhat` password.
 - 4.2. Click **Users** in the left navigation bar to manage users.
 - 4.3. Click the **+** button to add a new user.
 - 4.4. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Sylvia
LAST NAME	Simons
ORGANIZATION	Default
EMAIL	<code>sylvia@lab.example.com</code>
USERNAME	<code>sylvia</code>
PASSWORD	<code>redhat123</code>
CONFIRM PASSWORD	<code>redhat123</code>
USER TYPE	System Auditor

- 4.5. Click **SAVE** to create the new user.

► 5. Verify the permissions for **sylvia**.

- 5.1. Click **PERMISSIONS** to see the user's permissions.
- 5.2. As you can see, **sylvia** is a Member of the **Default** Organization and has the role of a System Auditor.
- 5.3. Log out and log back in as **sylvia** with password of **redhat123**.
- 5.4. You can see the user has access to all the elements in the left navigation bar.
- 5.5. Click **Users** in the left navigation bar to manage Users. As you can see, it is not possible for **sylvia** to add new users.
- 5.6. Click the **Log Out** icon to log out of the Tower web interface.

► 6. Create a user, **simon**, as a System Administrator.

- 6.1. Log in to the Tower web interface as the **admin** user with the **redhat** password.
- 6.2. Click **Users** in the left navigation bar to manage users.
- 6.3. Click the **+** button to add a new user.
- 6.4. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Simon
LAST NAME	Stephens
ORGANIZATION	Default
EMAIL	simon@lab.example.com
USERNAME	simon
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	System Administrator

- 6.5. Click **SAVE** to create the new user.

► 7. Verify the permissions for **simon**.

- 7.1. Click **PERMISSIONS** to see the user's permissions. This time the permissions explicitly say "SYSTEM ADMINISTRATORS HAVE ACCESS TO ALL PERMISSIONS".
- 7.2. Log out and log back in as **simon** with password of **redhat123**.
- 7.3. You can see, the user has access to all the elements in the left navigation bar.
- 7.4. Click **Users** in the left navigation bar to manage users. As you can see, **simon** has the rights to create new users.
- 7.5. Click the **Log Out** icon to log out of the Tower web interface.

This concludes the guided exercise.

MANAGING USERS EFFICIENTLY WITH TEAMS

OBJECTIVES

After completing this section, students should be able to create new teams in the web user interface, assign users to them, and explain the different roles that can be assigned to users.

TEAMS

Teams are groups of users. Teams make managing roles on Tower objects such as Inventories, Projects, and Job Templates, more efficient than managing them for each user separately. Users, who are members of a Team, inherit the roles assigned to that Team.

Rather than assigning the same roles to multiple users, a Tower administrator can assign roles to the Team representing a group of users.

In Ansible Tower, users exist as objects at a Tower level. A user can have roles in multiple Organizations. A Team belongs to exactly one Organization, but a Tower System Administrator can assign the Team roles on resources belonging to other Organizations.



IMPORTANT

A Team belongs to a particular Organization, but it cannot be assigned roles on an Organization object, like individual users can. To manage an Organization object, roles must be assigned directly to specific users.

Teams can be assigned roles on resources which belong to a particular Organization, such as Projects, Inventories, or Job Templates.

CREATING TEAMS

Teams are created within each Organization using the following procedure.

- Log in to the Tower web interface as the Tower admin user or as a user assigned the admin role for the Organization in which you intend to create the new Team.
- Click the Teams link on the left navigation bar.
- Click the + button.
- On the New Team screen, enter a name for the new Team in the NAME field.
- If desired, enter a description in the DESCRIPTION field.
- For the required ORGANIZATION field, click the magnifying glass icon to get a list of the Organizations within Tower. In the SELECT ORGANIZATION screen, select the Organization to create the Team in, and then click SAVE.
- Click SAVE to finalize the creation of the new Team.

TEAM ROLES

You can add users to the newly created Team. The role assigned to the User for a specific Team defines its relationship with it.

A user can be assigned one or more of the following Team roles:

admin

The Team **admin** role grants users full control of the Team. Users with this Team role can manage the Team's users and their associated Team roles. Users with Team **admin** roles can also manage the Team's roles on resources for which the Team has been assigned **admin** role.

member

The Team **member** role allows users to inherit roles on Tower resources granted to the Team. It also grants users the ability to see the Team's users and associated Team roles.

read

The Team **read** role gives users the ability to see the Team's users and their associated Team roles. A user assigned a Team **read** role does not inherit roles that the Team has been granted on Tower resources.



IMPORTANT

Users with Team **admin** roles can only manage the Team's roles on a resource when the resource also grants the Team **admin** role on itself. Just because a Team **admin** can manage Team membership, it does not imply that the Team **admin** has any rights to manage roles on objects to which the team has access.

For example, for a user to grant a Team the **use** role for a Project, the user needs to have the **admin** role for both the Team and the Project.

ADDING USERS TO A TEAM

After Team creation is complete, you can add users to the Team. To add users to a Team in an Organization, execute the following procedure:

- Log in to the Tower web interface as the Tower **admin** user or a user assigned the **admin** role for the Organization to which the team belongs.
- Click the **Organizations** link on the left navigation bar.
- Click the **TEAMS** link under the Organization which the Team belongs to.
- On the **Teams** screen, click the name of the Team to add the user to.
- On the Team details screen, click the **USERS** button.
- Click the **+** button.

The screenshot displays two main sections of the Ansible Tower interface. The top section, titled 'Developers', lists three users: 'admin' (SYSTEM ADMINISTRATOR), 'simon' (SYSTEM ADMINISTRATOR), and 'sylvia' (SYSTEM AUDITOR). The bottom section, titled 'TEAMS', shows a single team named 'Developers' associated with the 'Default' organization. Both sections feature standard UI elements like search, key generation, and an add button.

Figure 11.5: Select the Team to Assign roles to a user

- On the ADD USERS screen, select one or more users to add to Team. Then, click the SELECT ROLES drop-down menu next to each user and select the Team role to assign to the user.



NOTE

For a list of each role's definition, click the Key button.

- Click SAVE to save your edits, adding the user to the Team.



REFERENCES

You can find further information in the Teams section of the *Ansible Tower User Guide*
<http://docs.ansible.com/ansible-tower/latest/html/userguide>

► GUIDED EXERCISE

MANAGING USERS EFFICIENTLY WITH TEAMS

In this exercise, you will organize users into Teams and explore the access provided by different Team roles.

OUTCOMES

You should be able to organize users into Teams and explain the different roles that users can have on a Team.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Perform the following steps using the Default Organization.

► 1. Create a new Team called Developers.

- 1.1. Log in to the Tower web interface as the `admin` user with the `redhat` password.
- 1.2. Click **Teams** in the left navigation bar to manage Teams.
- 1.3. Click the **+** button to add a new Team.
- 1.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Developers
DESCRIPTION	Dev Team
ORGANIZATION	Default

- 1.5. Click **SAVE** to create the new Team.

► 2. Create a user which will be the administrator for the newly created Developers Team.

- 2.1. Click **Users** in the left navigation bar to manage users.
- 2.2. Click **+** to add a new user.
- 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Daniel
LAST NAME	George

FIELD	VALUE
EMAIL	daniel@lab.example.com
USERNAME	daniel
ORGANIZATION	Default
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

**NOTE**

We are creating an administrator of the Developers Team, not a System Administrator. This is why the User Type is set here to **Normal User**.

- 2.4. Click **SAVE** to create the new user.
- ▶ 3. Assign the **Admin** role on the Developers Team to the **daniel** user.
 - 3.1. Click **Teams** in the left navigation bar to manage Teams.
 - 3.2. Click the link for the Developers Team created previously.
 - 3.3. Click the **USERS** button to manage the users assigned to the Team.
 - 3.4. Click **+** to add a new user to the Team.
 - 3.5. In the first section of the screen, check the box next to **daniel** to select this user. This adds **Daniel George** to the list of selected users in the second section below.
 - 3.6. In the second section, click on the empty field next to the user and assign the **Admin** role by selecting it from the drop-down list.
 - 3.7. Click **SAVE**.
- ▶ 4. Create a user, **donnie**, to be granted **Read** role to the Developers Team.
 - 4.1. Click **Users** in the left navigation bar to manage users.
 - 4.2. Click **+** to add a new user.
 - 4.3. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Donnie
LAST NAME	Jameson
EMAIL	donnie@lab.example.com
USERNAME	donnie

FIELD	VALUE
ORGANIZATION	Default
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

4.4. Click **SAVE** to create the new user.

► **5.** Assign the **donnie** user the **Read** role for the **Developers** Team.

- 5.1. Click **Teams** in the left navigation bar to manage Teams.
- 5.2. Click the link for the **Developers** Team you created previously.
- 5.3. Click the **USERS** button to manage the Team's users.
- 5.4. Click **+** to add a new User to the Team.
- 5.5. In the first section of the screen, check the box next to **donnie** to select this user. This adds **Donnie Jameson** to the list of selected users in the second section below.
- 5.6. In the second section, click on the empty field next to the user and assign the **Read** role by selecting it from the drop-down list.
- 5.7. Click **SAVE**.

► **6.** Create a user, **david**, to be granted **Member** role to the **Developers** Team.

- 6.1. Click **Users** in the left navigation bar to manage users.
- 6.2. Click **+** to add a new user.
- 6.3. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	David
LAST NAME	Jacobs
EMAIL	david@lab.example.com
USERNAME	david
ORGANIZATION	Default
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

6.4. Click **SAVE** to create the new user.

- **7.** Assign the **Member** role on the Developers Team to **david**.
- 7.1. Click **Teams** in the left navigation bar to manage Teams.
 - 7.2. Click the link for the **Developers** Team you created previously.
 - 7.3. Click the **USERS** button to manage the Team's users.
 - 7.4. Click **+** to add a new user to the Team.
 - 7.5. In the first section on the screen, check the box next to **david** to select this user. This adds **David Jacobs** to the list of selected users in the second section below.
 - 7.6. In the second section, click on the empty field next to the user and assign the **Member** role by selecting it from the drop-down list.
 - 7.7. Click **SAVE** and click the **Log Out** icon to exit the Tower web interface.
- **8.** Verify the permissions for the **daniel** user.
- 8.1. Log in to the Tower web interface as **daniel** with a password of **redhat123**.
 - 8.2. Click **Teams** in the left navigation bar to manage Teams.
 - 8.3. Click the link for the **Developers** Team.
 - 8.4. Click the **USERS** button to manage the users assigned to the Team.
 - 8.5. Click the **+** button. As you can see, **daniel** can add and manage role assignments for himself in this Team.
 - 8.6. Click the **Log Out** icon to exit the Tower web interface.
- **9.** Verify the permissions for the **donnie** user.
- 9.1. Log in to the Tower web interface as **donnie** with a password of **redhat123**.
 - 9.2. Click **Teams** in the left navigation bar to manage Teams.
 - 9.3. Click the link for the **Developers** Team you created previously. The user, **donnie**, can view the settings for the Team that she has been assigned the **Read** role to.
 - 9.4. Click the **USERS** button and you should see that **donnie** sees all users that are part of the **Developers** Team (including the System Auditor and System Administrators) but has no rights to manage users or user roles in that Team.
 - 9.5. Click the **Log Out** icon to exit the Tower web interface.

► 10. Verify the permissions for the `david` user.

- 10.1. Log in to the Tower web interface as `david` with a password of **redhat123**.
- 10.2. Click **Teams** in the left navigation bar to manage Teams.
- 10.3. Click the link for the **Developers** Team you created previously.
- 10.4. Click the **USERS** button and you should see that, like `donnie`, `david` sees all users that are part of the **Developers** Team, including the System Auditor and System Administrators, but has no rights to manage users or user roles in that Team.
- 10.5. Click the **Log Out** icon to exit the Tower web interface.

This concludes the guided exercise.

► LAB

MANAGING ACCESS WITH USERS AND TEAMS

PERFORMANCE CHECKLIST

In this lab, you will create users and organize them into teams.

OUTCOMES

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

1. Add two users as Normal Users to the Default organization with the following information:

	USER 1	USER 2
FIRST NAME	Oliver	Ophelia
LAST NAME	Stone	Dunham
ORGANIZATION	Default	Default
EMAIL	<code>oliver@lab.example.com</code>	<code>ophelia@lab.example.com</code>
USERNAME	<code>oliver</code>	<code>ophelia</code>
PASSWORD	<code>redhat123</code>	<code>redhat123</code>
CONFIRM PASSWORD	<code>redhat123</code>	<code>redhat123</code>
USER TYPE	Normal User	Normal User

2. Create a Team called Operations in the Default Organization.
3. Add Oliver as a Member to the Operations Team. Add Ophelia as an Admin to the Operations Team.
4. Click the Log Out icon to log out of the Tower web interface.
5. Run the command `lab org-review grade` on workstation to grade your exercise.

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

This concludes the lab.

► SOLUTION

MANAGING ACCESS WITH USERS AND TEAMS

PERFORMANCE CHECKLIST

In this lab, you will create users and organize them into teams.

OUTCOMES

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

1. Add two users as Normal Users to the Default organization with the following information:

	USER 1	USER 2
FIRST NAME	Oliver	Ophelia
LAST NAME	Stone	Dunham
ORGANIZATION	Default	Default
EMAIL	oliver@lab.example.com	ophelia@lab.example.com
USERNAME	oliver	ophelia
PASSWORD	redhat123	redhat123
CONFIRM PASSWORD	redhat123	redhat123
USER TYPE	Normal User	Normal User

1. Log in to the Tower web interface as the `admin` user with the `redhat` password.
2. Click `Users` in the left navigation bar to manage users.
3. Click the `+` button to add a new user.
4. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Oliver
LAST NAME	Stone
ORGANIZATION	Default
EMAIL	oliver@lab.example.com
USERNAME	oliver

FIELD	VALUE
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

- 1.5. Click SAVE to create the new user.
- 1.6. Scroll down the next screen and click the + button to add another user.
- 1.7. On the next screen, fill in the details as follows:

FIELD	VALUE
FIRST NAME	Ophelia
LAST NAME	Dunham
ORGANIZATION	Default
EMAIL	ophelia@lab.example.com
USERNAME	ophelia
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

- 1.8. Click SAVE to create the new user.
2. Create a Team called Operations in the Default Organization.
 - 2.1. Click Teams in the left navigation bar to manage Teams.
 - 2.2. Click the + button to add a new team.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Operations
DESCRIPTION	Ops Team
ORGANIZATION	Default

- 2.4. Click SAVE to create the Operations Team.

3. Add Oliver as a Member to the Operations Team. Add Ophelia as an Admin to the Operations Team.
 - 3.1. On the Operations team editor screen, click USERS to manage the Team's users.
 - 3.2. Click the + button to add a new user to the Team.
 - 3.3. In the first section of the screen, check the boxes next to oliver and ophelia to select those users. This adds Oliver Stone and Ophelia Dunham to the list of selected users in the second section below.
 - 3.4. In the second section, assign the Member role to Oliver Stone and the Admin role to Ophelia Dunham by selecting them from the drop-down lists.
 - 3.5. Click SAVE.
 - 3.6. You can verify on the next screen that oliver has been assigned the role of Member and that ophelia has been assigned the role of Admin.
4. Click the Log Out icon to log out of the Tower web interface.
5. Run the command **lab org-review grade** on workstation to grade your exercise.

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

SUMMARY

In this chapter, you learned:

- Organizations are logical collections of users, teams, projects, and inventories.
- Users can be created as one of three user types: System Administrator, System Auditor, and Normal User.
- System Administrator and System Auditor are singleton roles that grant read-write and read-only access to all Tower objects, respectively.
- Users can be assigned one of four organization roles: **admin**, **auditor**, **member**, and **read**.
- A team is a group of users, and you can use a team to make it easier to assign particular roles on Tower resources to a set of users.
- Users can be assigned one of three team roles: **admin**, **member**, and **read**.
- The roles assigned to users control what access users have on objects.

CHAPTER 12

MANAGING INVENTORIES AND CREDENTIALS

GOAL

Create inventories of machines to manage, and set up credentials necessary for Red Hat Ansible Tower to log in and run Ansible jobs on those systems.

OBJECTIVES

- Create a static inventory of managed hosts, using the web UI.
- Create a machine credential for inventory hosts to allow Ansible Tower to run jobs on the inventory hosts using SSH.

SECTIONS

- Creating a Static Inventory (and Guided Exercise)
- Creating Machine Credentials for Access to Inventory Hosts (and Guided Exercise)

LAB

- Managing Inventories and Credentials

CREATING A STATIC INVENTORY

OBJECTIVES

After completing this section, students should be able to create a static inventory of managed hosts, using the web UI.

Figure 12.0: Creating a static inventory

ANSIBLE INVENTORY

Ansible Playbooks run against an *inventory* of hosts and groups of hosts. Ansible can select parts or all of the inventory as target systems to manage through task execution. Without Red Hat Ansible Tower, Ansible normally defines this inventory either as a static file or dynamically from external sources through a script.

This section of the course focuses on static inventory configuration and management using the Red Hat Ansible Tower web interface. As a review, look at the following Ansible static inventory file in INI format. It defines four hosts. The host `london1.example.com` is a member of no groups. The hosts `raleigh1.example.com` and `atlanta1.example.com` are members of the host group `southeast`. The host `boston1.example.com` is a member of the host group `northeast`. The host group `east` is a group of groups, including the `southeast` and `northeast` host groups.

```
london1.example.com

[southeast]
raleigh1.example.com
atlanta1.example.com

[northeast]
boston1.example.com

[east:children]
southeast
northeast
```

Two implicit groups also exist. The group `all` includes all the hosts in the inventory. The group `ungrouped` includes all hosts that are not in a group other than `all`, which in the previous example would be only `london1.example.com`.

Inventories are managed as objects in Ansible Tower, and can be configured in two different ways. This section looks at how to set up a static inventory and use RBAC to control access to it by using the Tower web interface.

CREATING AN INVENTORY IN ANSIBLE TOWER

Inventories are managed as objects in Ansible Tower. Each organization may have many inventories available. When job templates are created, they can be configured to use a specific inventory belonging to the organization. Which users on the Tower can use an inventory object is determined by the roles they have in the inventory.

This section looks at how to use Tower's web user interface to create a new inventory object from nothing, using the preceding sample inventory file as a model. Later sections cover other methods of configuring an inventory.

The following procedure illustrates how the example Ansible inventory previously shown can be implemented as a static inventory resource named **Mail Servers** assigned to the **Default** organization.

1. Log in as a user assigned the **Admin** role for the **Default** organization.
2. Click the Inventories quick navigation link on the left of the screen.
3. Create a new inventory. On the INVENTORIES screen, click the + button. Choose **Inventory** from the drop-down menu.
On the NEW INVENTORY screen, the NAME of the inventory and its ORGANIZATION are required. For this example, use **Mail Servers** for NAME and click the magnifying glass icon to select the **Default** organization. Click **SAVE**.
4. Add the host **london1.example.com** to the inventory.
On the MAIL SERVERS detail screen, click the HOSTS button.
On the HOSTS detail screen, click the + button. Enter **london1.example.com** for the HOST NAME, and then click **SAVE**.
5. Add the **east** group to the inventory.
Click the Inventories quick navigation link on the left of the screen. On the INVENTORIES screen, click the **Mail Servers** link.
On the MAIL SERVERS inventory detail screen, click the GROUPS button. Click the + button. On the CREATE GROUP screen, enter **east** in the NAME field. Click the **SAVE** button on the CREATE GROUP screen to finalize the addition of the group to the inventory.

Figure 12.1: New host group

6. Add the **northeast** and **southeast** groups as child groups of the **east** group just added to the inventory.
On the MAIL SERVERS inventory detail screen, click the GROUPS button. Click the **east** host group.
On the **east** group detail screen, click the GROUPS button.

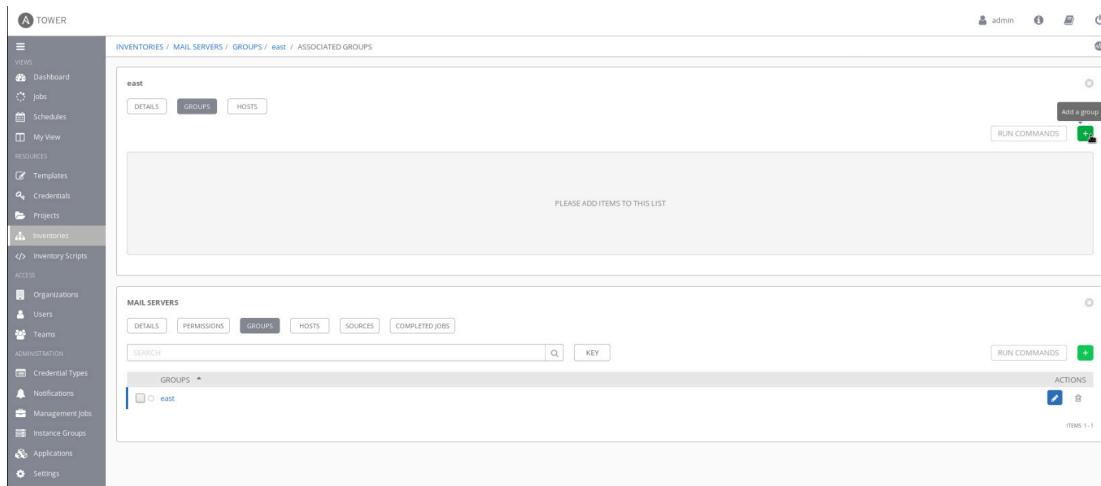


Figure 12.2: Adding a child host group

On the **east** group details screen, click the **+** button. Choose **New Group** from the drop-down menu. On the **CREATE GROUP** screen, enter **southeast** in the **NAME** field. Click the **SAVE** button on the **CREATE GROUP** screen to finalize the addition of the child group to the inventory.

On the **EAST** group details screen, click the **+** button again and repeat for the **northeast** group.

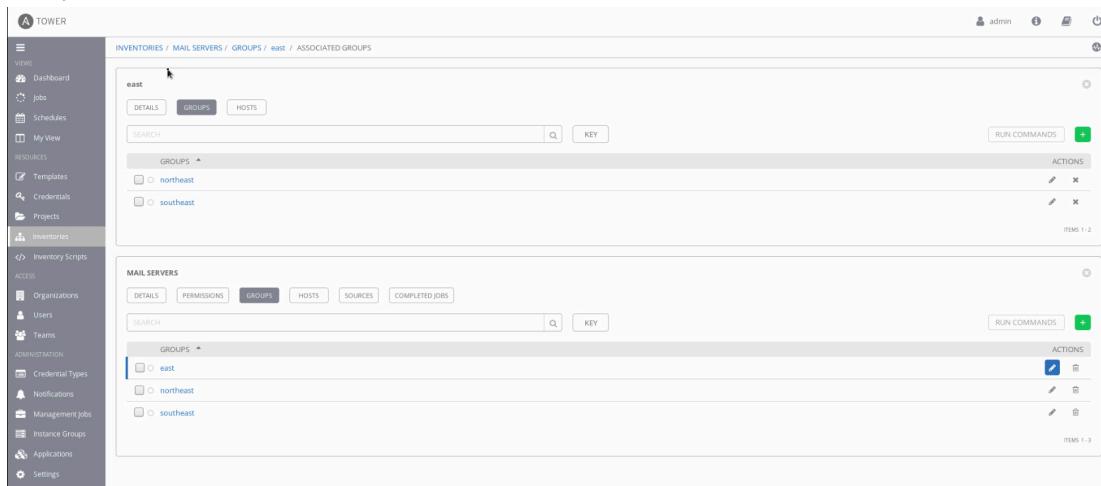


Figure 12.3: New child host groups

7. Add the `raleigh1.example.com` and `atlanta1.example.com` hosts to the `southeast` child group just added to the inventory.

On the **east** group details screen, click the **southeast** group link.

On the **southeast** group details screen, click the **HOSTS** button. Click the **+** button, from the drop-down menu choose **New Host**. On the **CREATE HOST** screen, enter `raleigh1.example.com` in the **NAME** field. Click the **SAVE** button on the **CREATE HOST** screen to finalize the addition of the host to the child group.

On the **SOUTHEAST** group details screen, click the **+** button again. Repeat the process to add the `atlanta1.example.com` host.

The screenshot displays two main sections of a management interface. The top section is titled 'southeast' and contains a 'HOSTS' tab. It lists two hosts: 'atlanta1.example.com' and 'raleigh1.example.com', both marked as 'ON'. Below this is a 'MAIL SERVERS' section showing three groups: 'east', 'northeast', and 'southeast', each with a green dot icon indicating they are active.

Figure 12.4: Partially configured child groups

8. Add the `boston1.example.com` host to the `northeast` child group.
On the EAST group details screen, click the `northeast` group link.
On the NORTHEAST group details screen, click the HOSTS button. Click the `+` button, and choose New Host from the drop-down menu. On the CREATE HOST screen, enter `boston1.example.com` in the NAME field. Click the SAVE button on the CREATE HOST screen to finalize the addition of the host to the child group.

This screenshot shows the completed configuration for the 'east' group. The 'HOSTS' tab in the top section now includes the host 'boston1.example.com'. The 'MAIL SERVERS' section at the bottom remains the same, displaying the three groups: 'east', 'northeast', and 'southeast'.

Figure 12.5: Completed host group detail screen

9. The completed Mail Servers inventory screen looks like the following:

The screenshot shows two main sections of the Red Hat Inventory Management interface. The top section, titled 'HOSTS', lists four hosts: atlanta1.example.com, boston1.example.com, london1.example.com, and raleigh1.example.com. These hosts are categorized under 'RELATED GROUPS' as southeast, northeast, and ungrouped. The bottom section, titled 'INVENTORIES', lists two inventories: 'Demo Inventory' (type: Inventory, organization: Default) and 'MAIL SERVERS' (type: Inventory, organization: Default).

Figure 12.6: Completed inventory detail screen

All four hosts in the inventory are shown: the one in northeast, the two in southeast, and the ungrouped one. This provides an immediate overview of all the hosts in the inventory.

Clicking on GROUPS displays a page showing all host groups in the Mail Servers inventory. Clicking on east displays a page showing the northeast and southeast child groups. Clicking on the GROUPS displays the ASSOCIATED GROUPS page and all the group members of east, providing an overview of the child groups in that group in the inventory. Clicking on the HOSTS displays all hosts in the east group.

The document stack icon can be used to move a host or host group to another host group or to the top level of the inventory.

If you click on a trash can icon next to a host, it is deleted from all host groups in the inventory.

If you click on a trash can icon next to a host group, you are given a choice before the host group is deleted:

- Delete all children belonging to the host group.
- *Promote* children belonging to the host group to its parent object. For example, when deleting east, its child groups can be promoted to top level groups in the inventory.

INVENTORY ROLES

A previous section covered how RBAC roles can be assigned to allow users and teams to view and manage teams and organizations. Roles can also be assigned to allow them to view and manage other objects such as inventories.

Users and teams can be assigned the ability to read, use, or manage an inventory by assigning appropriate roles for that inventory. In some cases, instead of directly assigning a role to the user or team, they may inherit a role granting them permissions to work with an inventory indirectly.

The following is a list of available roles for an inventory:

Admin

The inventory **Admin** role grants users full permissions over an inventory. These permissions include deletion and modification of the inventory. In addition, this role also grants permissions associated with the inventory roles **Use**, **Ad Hoc**, and **Update**, explained later in this chapter.

Use

The inventory **Use** role grants users the ability to use an inventory in a job template resource. This controls which inventory is used to launch jobs using the job template's playbook. Using an inventory in a job template is discussed in detail in a later chapter.

Ad Hoc

The inventory **Ad Hoc** role grants users the ability to use the inventory to execute ad hoc commands. Using Ansible Tower for ad hoc command execution is discussed in detail in Ansible Tower User Guide.

Update

The inventory **Update** role grants users the ability to update a dynamic inventory from its external data source. This is discussed in more detail later in this course.

Read

The inventory **Read** role grants users the ability to view the contents of an inventory.

When an inventory is first created, it is only accessible by users who have the **Admin**, **Inventory Admin**, or **Auditor** roles for the organization to which the inventory belongs. All other access must be specifically configured.

This is done by assigning users and teams appropriate roles as discussed above. The inventory must be created and saved before users and teams can be assigned roles on it.

Roles are assigned through the PERMISSIONS section of the inventory's editor screen (accessible through the pencil icon next to its name). The following procedure details the steps to assign users and teams roles on an inventory:

1. Log in as a user with **Admin** role on the organization in which the inventory was created.
2. Click **Inventories** in the quick navigation links to display the organization's inventory list.
3. Click the pencil icon for the inventory to enter the inventory's editor screen.
4. On the inventory's editor screen, click the **PERMISSIONS** button to enter the permissions editor.
A list appears, naming users and teams, with their assigned roles. If there is an X next to a role, it can be clicked to remove that role from the user.
5. Click the + button to add permissions.
6. In the **ADD USERS / TEAMS** selection screen, click either **USERS** or **TEAMS**. Select the check boxes next to the users or teams to be assigned new roles.
7. Under **Please assign roles to the selected users/teams**, click the **SELECT ROLES** drop-down menu. Select the desired inventory role for each user or team.
Click **KEY** for a list of inventory roles and their definitions.
8. Click **SAVE** to finalize the new roles.

CONFIGURING INVENTORY VARIABLES

Ansible supports *inventory variables* that apply values to variables on particular hosts (*host variables*) or to host groups (*group variables*).

If you are using Ansible by itself, the recommended way to set inventory variables is to use **host_vars** and **group_vars** directories in the same working directory as the inventory file. These directories contain YAML files named after the host or host group they represent, which

define variables and values that apply to that host or host group. Variables can be assigned to all hosts in an inventory by defining them for the special **all** group.

For example, the file **group_vars/southeast** might define the value of the **ntp** variable to **ntp-se.example.com** for the host group **southeast**:

```
---  
ntp: ntp-se.example.com
```

When you manage a static inventory in the Ansible Tower web UI, you may define inventory variables directly in the inventory object instead of using **host_vars** and **group_vars** directories.



IMPORTANT

If the project does have **host_vars** or **group_vars** files that set inventory variables applying to hosts in the play, they are honored. However, you are not able to edit those files in the Tower web UI.

Furthermore, if the same host or group variable is defined in both the project files and in a static inventory object managed through the web UI, and have different values, which value Tower uses might not be easy to predict.

When using static inventories managed in the Tower web UI, avoid creating conflicts like this by setting inventory variables in one place or the other, not both.

On the INVENTORIES screen, variables can be set by clicking the Edit inventory (pencil) icon next to the inventory's name. On the DETAILS screen for the inventory, you can set variables that affect all hosts in the inventory:

The screenshot shows the 'MAIL SERVERS' inventory details screen. At the top, there are tabs for DETAILS, PERMISSIONS, GROUPS, HOSTS, SOURCES, and COMPLETED JOBS. The DETAILS tab is selected. Below the tabs, there are fields for NAME (MAIL SERVERS), DESCRIPTION, and ORGANIZATION (Default). Under the 'INSIGHTS CREDENTIAL' section, there is a search bar and a dropdown for INSTANCE GROUPS. At the bottom, there is a 'VARIABLES' section with tabs for YAML and JSON. The JSON tab is selected, showing a single entry: '1'. There are 'CANCEL' and 'SAVE' buttons at the bottom right.

Figure 12.7: Variables for all hosts

When a host group is created within an inventory, group variables can be defined using either YAML or JSON in the VARIABLES field on the CREATE GROUP screen. They can also be set by clicking on the Edit group (pencil) icon next to the host group's name in the inventory. These variables apply to all hosts that are part of the group:

The screenshot shows two Ansible Tower interfaces. The top part is a modal window for a host group named 'southeast'. It has tabs for 'DETAILS', 'GROUPS', and 'HOSTS'. Under 'DETAILS', there's a 'NAME' field with 'southeast' and a 'DESCRIPTION' field. A 'VARIABLES' section contains YAML code: '1: ...' and '2: http: ntp-se.example.com'. The bottom part is a 'MAIL SERVERS' interface showing a tree view of inventory groups: 'east' (selected), 'northeast', and 'southeast'. Each group has edit and delete icons next to it.

Figure 12.8: Variables for a host group

Likewise, host variables can be defined using either YAML or JSON in the VARIABLES field on the CREATE HOST screen when an individual host is created within an inventory, or by clicking on the Edit host (pencil) icon next to the host's name in the inventory, once created. Variables defined in this manner only apply to the specific host.

The screenshot shows a modal window for a host named 'boston1.example.com'. It has tabs for 'DETAILS', 'FACTS', 'GROUPS', and 'COMPLETED JOBS'. Under 'DETAILS', there's a 'HOST NAME' field with 'boston1.example.com' and a 'DESCRIPTION' field. A 'VARIABLES' section contains YAML code: '1: ...'. The bottom part is a similar interface to Figure 12.8, showing the 'east' group selected in the inventory tree.

Figure 12.9: Variables for an individual host**IMPORTANT**

Inventory variables can be overridden by variables with a higher precedence.

Extra variables defined in a job template and playbook variables both have higher precedence than inventory variables.

**REFERENCES**

Further information is available in the Inventories section of the *Ansible Tower User Guide* at
<https://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

Further information is available in the Built-in Roles section of the *Ansible Tower User Guide* at
<https://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

► GUIDED EXERCISE

CREATING A STATIC INVENTORY

In this exercise, you will create a static inventory in Ansible Tower and grant users permission to manage and use that inventory.

OUTCOMES

You should be able to create and manage a static Inventory containing hosts and groups and assign appropriate permissions to a team.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run **lab host-inventory setup**.

```
[student@workstation ~]$ lab host-inventory setup
```

This setup script creates some additional inventories, hosts, and groups needed for this exercise.

- ▶ 1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
- ▶ 2. Create a static Inventory named `Prod`.
 - 2.1. Open the Inventories page by clicking `Inventories` in the left quick navigation bar.
 - 2.2. Click the `+` button to add a new Inventory.
 - 2.3. From the drop-down menu, select the normal `Inventory` type.
 - 2.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Prod
DESCRIPTION	Production Inventory
ORGANIZATION	Default

- 2.5. Click `SAVE` to create the new Inventory. You are redirected to the Inventory details page.

► 3. Create the group **prod-servers** in the Prod Inventory.

- 3.1. Click the GROUPS button.
- 3.2. Click the + button to add a new group.
- 3.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	prod-servers
DESCRIPTION	Production servers

- 3.4. Click SAVE to create the new group.

► 4. Add the host **servere.lab.example.com** to the group **prod-servers** in the Prod Inventory.

- 4.1. Click the HOSTS button to add hosts to the group you just created.
- 4.2. Click the + button to add a new host to the group. From the drop-down menu, select New Host.
- 4.3. On the next screen, fill in the details as follows:

FIELD	VALUE
HOST NAME	servere.lab.example.com
DESCRIPTION	Server E

- 4.4. Click SAVE to add the new host.

► 5. Assign the Operations team **Admin** role to the Prod Inventory.

- 5.1. Click Inventories in the left quick navigation bar.
- 5.2. On the same line as the Prod Inventory entry, click the pencil icon to edit the Inventory.
- 5.3. On the next page, click PERMISSIONS to manage the Prod Inventory's permissions.
- 5.4. Click the + button to add permissions.
- 5.5. Click TEAMS to display the list of available teams.
- 5.6. In the first section, check the box next to the Operations team. This displays that team in the second section.
- 5.7. In the second section below, select the **Admin** role from the drop-down menu.
- 5.8. Click SAVE to finalize the role assignment. This redirects you to the list of permissions for the Prod Inventory, which now shows that all members of the Operations team are assigned the **Admin** role on the Prod Inventory.

► 6. Verify access to the Prod Inventory with the user **oliver**, who belongs to the Operations team.

- 6.1. Click the Log Out icon in the top right corner to log out. Then log back in as **oliver** with a password of **redhat123**. This user is assigned the **Member** role for the Operations team.
 - 6.2. Click Inventories in the left quick navigation bar.
 - 6.3. Click the link for the Prod Inventory created earlier.
 - 6.4. Review the contents of the Prod Inventory to see the hosts and group it contains.
- 7. Add the host, `serverf.lab.example.com`, to the Prod Inventory while logged in as the user **oliver**.
- 7.1. Click the link for the prod-servers group to enter the group.
 - 7.2. Click the HOSTS button.
 - 7.3. Click the + button to add a new host to the group. From the drop-down menu, select New Host.
 - 7.4. On the next screen, fill in the details as follows:
- | FIELD | VALUE |
|-------------|-------------------------|
| HOST NAME | serverf.lab.example.com |
| DESCRIPTION | Server F |
- 7.5. Click SAVE to create the second host.
- 8. Assign the **Use** role on the Test Inventory to the Developers team.
- 8.1. Click the Log Out icon in the top right corner to log out and then log back in as **admin** with a password of **redhat**.
 - 8.2. Click Inventories in the left navigation bar.
 - 8.3. On the same line as the Test Inventory, click the pencil icon to edit the Inventory.
 - 8.4. On the next page, click PERMISSIONS to manage the Inventory's permissions.
 - 8.5. Click the + button to add permissions.
 - 8.6. Click TEAMS to display the list of available teams.
 - 8.7. In the first section, check the box next to the Developers team. This displays that team in the second section underneath the first one.
 - 8.8. In the second section, select the **Use** role from the drop-down menu.
 - 8.9. Click SAVE to make the role assignment. This redirects you to the list of permissions for the Test Inventory, which now shows that all members of the Developers team are assigned the **Use** role on the Test Inventory.

- ▶ 9. Verify access to the Test Inventory with the daniel user who belongs to the Developers team.
- 9.1. Click the Log Out icon in the top right corner to log out. Log back in as daniel with a password of redhat123. This user is assigned the **Admin** role for the Developers team.
 - 9.2. Click Inventories in the left navigation bar.
 - 9.3. Click the link for the Test Inventory.
 - 9.4. Review the contents of the Test Inventory to see the hosts and group it contains. Note that even though daniel is an **Admin** of the Developers team, he cannot manage the Test Inventory because you only granted the **Use** role for the Inventory to the **Developers** team.

**NOTE**

This is representative of a real world scenario where developers may have access to the list of systems in the testing environment but are not able to modify the list.

- 9.5. Click the Log Out icon in the top right corner to log out of the Tower web interface.

This concludes the guided exercise.

CREATING MACHINE CREDENTIALS FOR ACCESS TO INVENTORY HOSTS

OBJECTIVES

After completing this section, students should be able to create a machine credential for inventory hosts to allow Ansible Tower to run jobs on the inventory hosts using SSH.

Figure 12.9: Creating machine credentials for access to inventory hosts

CREDENTIALS

Credentials are Ansible Tower objects used to authenticate to remote systems. They may provide secrets such as passwords, SSH keys, or other supporting information needed to successfully access or use a remote resource.

Ansible Tower is responsible for maintaining secure storage for the secrets in these Credential objects. Credential passwords and keys are encrypted before they are saved to the Tower database, and can not be retrieved in clear text from the Tower user interface. Users and teams can be assigned privileges to use Credentials, but the secrets are not exposed to them. This means that when a user changes teams or leaves the organization, the credentials and systems do not need to be re-keyed. When a Credential is needed by Tower, it decrypts the data internally and passes it to SSH or other program directly.



IMPORTANT

Once sensitive authentication data is entered into a Credential and encrypted, it can no longer be retrieved in decrypted form through the Tower web interface.

CREDENTIAL TYPES

Ansible Tower has a number of different types of Credentials that it can manage. Some of these include:

- *Machine* credentials are used to authenticate playbook logins and privilege escalation for Inventory hosts
- *Network* credentials are used when Ansible network modules are used to manage networking equipment
- *Source Control* (or SCM) credentials are used by Projects to clone and update Ansible project materials from a remote version control system such as Git, Subversion, or Mercurial
- *Vault* credentials are used to decrypt sensitive information stored in project files protected by Ansible Vault
- Several types of inventory credentials are available to update dynamic inventory information from one of Ansible Tower's built-in dynamic inventory sources. There are separate credential types for each inventory source in question: Amazon Web Services, VMware vCenter, Red Hat Satellite 6, Red Hat CloudForms, Google Compute Engine, Microsoft Azure Resource Manager, OpenStack, and so on.

- It is also possible for Tower administrators to create *custom credential types* that may be used like the built-in credential types, specified using a YAML definition. For more information on custom credential types, see the *Ansible Tower User Guide*.

This section focuses on how to set up machine credentials that provide appropriate login and privilege escalation information for use with hosts in an inventory. Some of the other types of credentials are discussed in more detail elsewhere in this course.

CREATING MACHINE CREDENTIALS

Credentials are managed through the CREDENTIALS page under Tower's Credentials link on the left navigation bar.

Any user can create a Credential, and is considered the *owner* of that Credential. If the Credential is not assigned to an Organization, it is a *private* Credential. This means that only the user owning the Credential and users with the Tower System Administrator singleton role can use it, and only they and users with the Tower System Auditor singleton role can see it.

On the other hand, if the Credential is assigned to an Organization, then it is an *Organization* Credential. Only Tower System Administrators and Users with **Admin** on an Organization can create Credentials assigned to an Organization. Users and teams in the Organization can be granted roles on a Credential assigned to the Organization, to use or manage the Credential.

In summary, the main distinctions between private Credentials and those assigned to an Organization are:

- Any user can create a private Credential, but only Tower System Administrators and users with Organization **Admin** can create an Organization Credential
- If a Credential belongs to an Organization, users and teams can be granted roles on it, and it can be shared. Private Credentials not assigned to an Organization can only be used by the owner and the Tower singleton roles, and other users and teams cannot be granted roles on it.



IMPORTANT

The Tower **Admin** user can assign an Organization to an existing private Credential, converting it into an Organization Credential.

The following procedure details how Machine Credentials are created.

- Login as a user with the appropriate role assignment. If creating a private Credential, there are no specific role requirements. If creating an Organization Credential, login as a user with the **Admin** role for the Organization.
- Click Credentials to enter the Credentials management interface.
- On the CREDENTIAL screen, click + to create a new Credential.
- On the NEW CREDENTIAL screen, enter the required information for the new Credential. A NAME is required, and the TYPE drop-down menu should be used to select Machine. If the user has Organization **Admin** privileges, the ORGANIZATION can be set to assign this Credential to an Organization. If the user does not have **admin** privileges, then the ORGANIZATION field is not present and only private Credentials can be created.
- For Machine Credentials, additional fields appear in the TYPE DETAILS section, as shown in the following illustration:

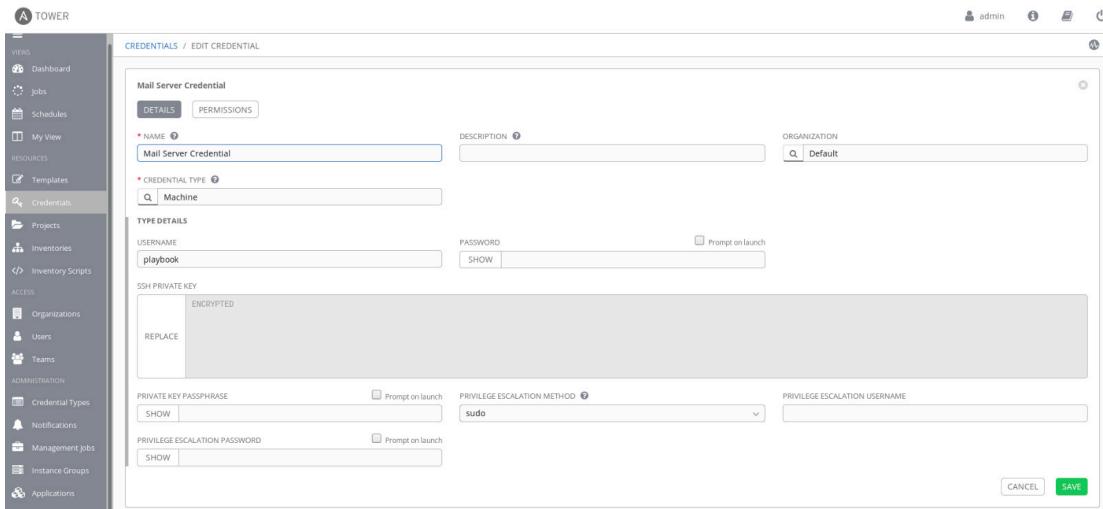


Figure 12.10: New machine Credential (after save)

These fields can contain the information needed to authenticate to and escalate privileges on the machines that use this Credential. Many of them are mapped to settings, which might be in an `ansible.cfg` file:

- **USERNAME** is the username used to log in to the managed hosts (`remote_user` in `ansible.cfg`)
- **PASSWORD** is the SSH password to use for that user. Leave this blank if private key authentication is used.
- The **SSH PRIVATE KEY** field contains an SSH private key that can be used to log in as **USERNAME** on the managed hosts. It is easier to cut and paste the text from the file rather than to manually type it in. If you are administering Ansible Tower from a Firefox browser running under GNOME 3, you can drag and drop the private key file from the Files application window into this field in your web browser window.

Once the Credential is saved, this is encrypted by Tower, so the field reads **ENCRYPTED**.

- **PRIVATE KEY PASSPHRASE** is used if the SSH key in **SSH PRIVATE KEY** is encrypted by SSH for protection. It accepts a passphrase to decrypt the key. Otherwise, this field can be blank.
- **PRIVILEGE ESCALATION METHOD** is a drop-down menu that specifies what type of privilege escalation, if any, is needed (`become_method`). This affects other fields that may appear.

For **sudo** privilege escalation, **PRIVILEGE ESCALATION USERNAME** is the privileged user that Ansible should use on the managed host (`become_user`). **PRIVILEGE ESCALATION PASSWORD** is the **sudo** password to use. This can be blank if no password is needed.

6. Click the **SAVE** button to finalize the creation of the new Machine Credential.

EDITING MACHINE CREDENTIALS

Once Machine Credentials are created, they can be edited, if needed, using the Credential editor interface. The following procedure details how Credentials are modified.

1. Log in as a user with the appropriate role assignment. If editing a private Credential, login as the user who created the Credential. If editing an Organization Credential, login as a user with **Admin** role on the Organization Credential.

2. Click the Credentials link to enter the Credentials management interface.
3. On the CREDENTIAL screen, click the name of the Credential to edit.
4. On the Credential editor screen, make the necessary changes to the desired Credential properties.
5. Click SAVE button to finalize the changes made to the Credential.

CREDENTIAL ROLES

As discussed earlier, private Credentials (Credentials that are not assigned to an Organization) are only accessible to their creators or to Users that have the System Administrator or System Auditor user type. Other users can not be assigned roles on private Credentials.

To assign roles to Credentials, the Credential must have an Organization. Then Users and Teams in that Organization can share that Credential through role assignments.

The following is the list of available Credential roles.

Admin

The Credential **admin** role grants Users full permissions on a Credential. These permissions include deletion and modification of the Credential, as well as the ability to use the Credential in a Job Template.

Use

The Credential **use** role grants Users the ability to use a Credential in a Job Template. Use of a Credential in a Job Template is discussed later in this course.

Read

The Credential **read** role grants Users the ability to view the details of a Credential. This still does not allow them to decrypt the secrets which belong to that Credential through the web interface.

MANAGING CREDENTIAL ACCESS

When an Organization Credential is first created, it is only accessible by the owner and users with either the **Admin** or **Auditor** role in the Organization in which the Credential was created. Additional access, if desired, must be specifically configured.

Additional roles cannot be assigned until it is first saved, after which they can be set by editing the Credential.

Roles are assigned through the PERMISSIONS section of the Credential editor screen.

The following procedure details the steps for granting permissions to an Organization Credential after its creation.

1. Log in as a user with **Admin** role on the Organization that the Credential was created in.
2. Click the Credentials to enter the Credentials management interface.
3. Click the name of the Credential to edit in order to enter the Credential editor screen.
4. On the Credential editor screen, click the PERMISSIONS button to enter the permissions editor.
5. Click the + button to add permissions.

6. In the user and team selection screen, click either USERS or TEAMS and then select the check boxes for the users or teams to be assigned roles.
7. Under Please assign roles for the selected users/teams, click KEY to display the list of Credential roles and their definitions.
8. Click the SELECT ROLES drop-down menu and select the desired Credential role for each user or team.
9. Click SAVE to finalize the changes to permissions.



IMPORTANT

Permissions for Credentials can also be added through either the user or team management screens.

COMMON CREDENTIAL SCENARIOS

To help you understand ways in which Credentials are used, here are some common Credential scenarios:

Credentials Protected by Tower, Not Known to Users

One common scenario for the use of Tower Credentials is the delegation of task execution from administrators to Tier 1 support staff.

For example, suppose that the support staff needs to be delegated the ability to run a playbook ensuring a web application has been restarted to restore service when outages occur outside of business hours. The support staff's Credential uses a shared account, **support**, and a passphrase-protected private key for SSH authentication to managed hosts. The **support** account needs to escalate privileges using **sudo**, with a **sudo** password, in order to run the playbook.

Since the Credential is shared by the support staff's team, an Organization Credential resource should be created to store the username, SSH private key, and SSH key passphrase needed to authenticate SSH sessions to the managed hosts. The Credential also stores the privilege escalation method, username, and **sudo** password information. Once created, the Credential can be used by the support staff to launch Jobs on the managed hosts without needing to know the SSH key passphrase or **sudo** password.

Figure 12.11: Organization Credential for shared account

Credential Prompts for Sensitive Password, Not Stored in Tower

Another scenario is to use Credentials to store username authentication information while still prompting interactively for a sensitive password when the Credential is used.

Suppose that a database administrator wants to run a playbook managed in Tower to execute tasks on a database server which houses sensitive data for the company financials. Due to the highly sensitive nature of the data, the company's financial compliance regulations forbid the storage of the account's password.

A machine Credential is still used to configure the database administrator's authentication to the database server. Since the Credential is not to be shared, a private Credential can be used to store the SSH USERNAME. It is also configured to prompt the user for the account's password when the Credential is used by a Job, by selecting the Prompt on launch check box for PASSWORD.

The screenshot shows the 'CREATE CREDENTIAL' page in Ansible Tower. A 'Machine' type credential is being created with the name 'Finance DBA'. The 'PASSWORD' field has the 'Prompt on launch' checkbox checked. Other fields like 'DESCRIPTION' and 'ORGANIZATION' are also visible.

Figure 12.12: Private Credential with password prompting



IMPORTANT

Ansible Tower has a feature that allows playbooks to be run automatically on a schedule, much like a **cron** job. Credentials configured to prompt interactively for password information at runtime can not be used with scheduled jobs, since Tower can not provide that information without user interaction.



REFERENCES

Further information is available in the Credentials section of the *Ansible Tower User Guide* at
<https://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

Further information is available in the Built-in Roles section of the *Ansible Tower User Guide* at
<https://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

► GUIDED EXERCISE

CREATING MACHINE CREDENTIALS FOR ACCESS TO INVENTORY HOSTS

In this exercise, you will create a machine credential and assign roles to users that permit them to use it.

OUTCOMES

You should be able to create and manage Machine Credentials to be used against hosts and groups in an Inventory.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

- ▶ 1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
- ▶ 2. Create a new Credential called `Operations`.
 - 2.1. Click `Credentials` in the left navigation bar.
 - 2.2. Click the `+` button to add a new Credential.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Operations
DESCRIPTION	Operations Credential
ORGANIZATION	Default
CREDENTIAL TYPE	Machine
USERNAME	devops
PASSWORD	redhat
PRIVILEGE ESCALATION METHOD	sudo
PRIVILEGE ESCALATION USERNAME	root
PRIVILEGE ESCALATION PASSWORD	redhat

- 2.4. Leave the other fields untouched and click `SAVE` to create the new Credential.

- ▶ 3. Assign the Operations Team the **Admin** role on the Operations Credential.
- 3.1. Click **Credentials** in the left navigation bar.
 - 3.2. On the same line as the **Operations** Credential, click the pencil icon to edit the **Operations** Credential.
 - 3.3. On the next page, click **PERMISSIONS** to manage the permissions for the Credential.
 - 3.4. Click the **+** button to add permissions.
 - 3.5. Click **TEAMS** to display the list of available Teams.
 - 3.6. In the first section, check the box next to the **Operations** Team. This causes the Team to display in the second section underneath the first one.
 - 3.7. In the second section below, select the **Admin** Role from the drop-down menu.
 - 3.8. Click **SAVE** to make the role assignment. This redirects you to the list of permissions for the **Operations** Credential which now shows that all members of the **Operations** Team, **oliver** and **ophelia**, are assigned the **Admin** role on the **Operations** Credential.
- ▶ 4. Verify the permissions of the **Admin** role to the **Operations** Credential with the User **oliver**, who belongs to the **Operations** Team.
- 4.1. Click the **Log Out** icon in the top right corner to log out and sign back in as **oliver** with password of **redhat123**. This User is a **Member** of the **Operations** Team.
 - 4.2. Click **Credentials** in the left navigation bar.
 - 4.3. Click the link for the **Operations** Credential created earlier. Note how **oliver** can modify the Credential.
- ▶ 5. Assign the Developers Team the **Use** role on the **Operations** Credential.
- 5.1. Click the **Log Out** icon in the top right corner to log out and log back in as **admin** with a password of **redhat**.
 - 5.2. Click **Credentials** in the left navigation bar.
 - 5.3. On the same line as the **Operations** Credential entry, click the pencil icon to edit the Credential.
 - 5.4. On the next page, click **PERMISSIONS** to manage the permissions.
 - 5.5. Click the **+** button to add permissions.
 - 5.6. Click **TEAMS** to display the list of available Teams.
 - 5.7. In the first section, select the check box next to the **Developers** Team. This causes the Team to display in the second section underneath the first one.
 - 5.8. In the second section below, select the **Use** role from the drop-down menu.
 - 5.9. Click **SAVE** to finalize the role assignment. This redirects you to the list of permissions for the **Operations** Credential which now shows that all members of the **Developers** Team, **daniel**, **david** and **sam**, are assigned the **Use** role on the **Operations** Credential.

- 6. Verify the **Use** role for the **Operations** Credential with the user **daniel**, who belongs to the **Developers** team.
- 6.1. Click the Log Out icon in the top right corner to log out and log back in as **daniel** with **redhat123** as the password. This user has an **Admin** role for the **Developers** team.
 - 6.2. Click **Credentials** in the left navigation bar.
 - 6.3. Click the link for the **Operations** Credential created earlier. Note that **daniel** cannot modify the Credential even though he has an **Admin** role for the team.
 - 6.4. Click the Log Out icon in the top right corner to log out of the Tower web interface.

This concludes the guided exercise.

► LAB

CREATING AND MANAGING INVENTORIES AND CREDENTIALS

PERFORMANCE CHECKLIST

In this lab, you will create inventories and credentials and assign roles to users that permit the users to manage them.

OUTCOMES

You should be able to manage Inventories and Credentials in order for a team to be able to run a playbook against an inventory.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
2. Create a new Inventory called `Dev` within the `Default` Organization.
3. Create a group called `dev-servers` in the `Dev` Inventory.
4. Add two hosts with the host names `servera.lab.example.com` and `serverb.lab.example.com` to the `dev-servers` group.
5. Grant the `Admin` role on the `Dev` Inventory to the `Developers` team.
6. Create a new Credential, `Developers`, with the following information:

FIELD	VALUE
NAME	Developers
DESCRIPTION	Developers Credential
ORGANIZATION	Default
TYPE	Machine
USERNAME	devops
PASSWORD	redhat
PRIVILEGE ESCALATION	sudo
PRIVILEGE ESCALATION USERNAME	root
PRIVILEGE ESCALATION PASSWORD	redhat

7. Grant the `Admin` role on the `Developers` Credential to the `Developers` team.
8. Run the command, `lab host-review grade`, on `workstation` to grade your exercise.

► SOLUTION

CREATING AND MANAGING INVENTORIES AND CREDENTIALS

PERFORMANCE CHECKLIST

In this lab, you will create inventories and credentials and assign roles to users that permit the users to manage them.

OUTCOMES

You should be able to manage Inventories and Credentials in order for a team to be able to run a playbook against an inventory.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
2. Create a new Inventory called `Dev` within the `Default` Organization.
 - 2.1. Click **Inventories** in the left quick navigation bar.
 - 2.2. Click the **+** button. From the drop-down list, select **Inventory** to add a new Inventory.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Dev
DESCRIPTION	Development Inventory
ORGANIZATION	Default

- 2.4. Click **SAVE** to create the new Inventory. You are redirected to the Inventory details page.
3. Create a group called `dev-servers` in the `Dev` Inventory.
 - 3.1. Click the **GROUPS** button.
 - 3.2. Click the **+** button to add the new group.
 - 3.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	dev-servers

FIELD	VALUE
DESCRIPTION	Development servers

- 3.4. Click **SAVE** to create the new group.
- 4.** Add two hosts with the host names `servera.lab.example.com` and `serverb.lab.example.com` to the `dev-servers` group.
- 4.1. Click the **HOSTS** button, within the group you just created.
 - 4.2. Click the **+** button. From the drop-down menu, select **New Host** to add a new host to the group.
 - 4.3. On the next screen, fill in the details as follows:

FIELD	VALUE
HOST NAME	<code>servera.lab.example.com</code>
DESCRIPTION	Server A

- 4.4. Click **SAVE** to create the new host.
- 4.5. Click the **+** button. From the drop-down menu, select **New Host** to add a new host to the group.
- 4.6. On the next screen, fill in the details as follows:

FIELD	VALUE
HOST NAME	<code>serverb.lab.example.com</code>
DESCRIPTION	Server B

- 4.7. Click **SAVE** to create the new host.
- 5.** Grant the **Admin** role on the **Dev Inventory** to the **Developers** team.

 - 5.1. Click **Inventories** in the left quick navigation bar.
 - 5.2. On the same line as the **Dev Inventory**, click the pencil icon to edit the Inventory.
 - 5.3. On the next screen, click **PERMISSIONS** button to manage the Inventory's permissions.
 - 5.4. Click the **+** button to add permissions.
 - 5.5. Click **TEAMS** to display the list of available Teams.
 - 5.6. In the first section, check the box next to the **Developers Team**. This displays that team in the second section underneath the first one.
 - 5.7. In the second section, select the **Admin** role from the drop-down menu.
 - 5.8. Click **SAVE** to finalize the role assignment. You are redirected to the list of permissions for the **Dev Inventory**, which now shows that all members of the **Developers** team are assigned the **Admin** role on the **Dev Inventory**.

6. Create a new Credential, Developers, with the following information:

FIELD	VALUE
NAME	Developers
DESCRIPTION	Developers Credential
ORGANIZATION	Default
TYPE	Machine
USERNAME	devops
PASSWORD	redhat
PRIVILEGE ESCALATION	sudo
PRIVILEGE ESCALATION USERNAME	root
PRIVILEGE ESCALATION PASSWORD	redhat

- 6.1. Click Credentials in the left quick navigation bar.
 6.2. Click the + button to add a new Credential.
 6.3. Create a new Credential, Developers, with the following information:

FIELD	VALUE
NAME	Developers
DESCRIPTION	Developers Credential
ORGANIZATION	Default
TYPE	Machine
USERNAME	devops
PASSWORD	redhat
PRIVILEGE ESCALATION	sudo
PRIVILEGE ESCALATION USERNAME	root
PRIVILEGE ESCALATION PASSWORD	redhat

- 6.4. Leave the other fields untouched and click SAVE to create the new Credential.

7. Grant the Admin role on the Developers Credential to the Developers team.
 - 7.1. Click **Credentials** in the left quick navigation bar.
 - 7.2. On the same line as the Developers Credential, click the pencil icon to edit the Credential.
 - 7.3. On the next page, click **PERMISSIONS** to manage the Credential's permissions.
 - 7.4. Click the **+** button to add permissions.
 - 7.5. Click **TEAMS** to display the list of available teams.
 - 7.6. In the first section, check the box next to the Developers team. This causes the team to display in the second section underneath the first one.
 - 7.7. In the second section below, select the Admin Role from the drop-down menu.
 - 7.8. Click **SAVE** to finalize the role assignment. This redirects you to the list of permissions for the Developers Credential which now shows that the users, daniel, david, and donnie, are assigned the Admin role on the Developers Credential.
 - 7.9. Click the Log Out icon to exit the Tower web interface.
8. Run the command, **lab host-review grade**, on workstation to grade your exercise.

SUMMARY

In this chapter, you learned:

- Inventory resources are used to manage Ansible inventories of hosts and host groups and their inventory variables
- Multiple inventories can be configured, and roles can be used to manage who can use and administrate particular inventories
- Static inventories can be manually configured through the web interface
- Credentials are used to store authentication information for machines, network devices, source control, and dynamic inventory updates
- Machine credentials are used to allow Ansible Tower to authenticate access and to enable privilege escalation on inventory hosts for playbook execution
- Credentials assigned to an organization can be shared by granting roles to users and teams
- Credentials not assigned to an organization are private to the user who created it and to the Ansible Tower singleton roles, and cannot be shared without assigning it to an organization

CHAPTER 13

MANAGING PROJECTS AND LAUNCHING ANSIBLE JOBS

GOAL

Create projects and job templates in the web UI, using them to launch Ansible Playbooks that are stored in Git repositories in order to automate tasks on managed hosts.

OBJECTIVES

- Create and manage Ansible Playbooks in a Git repository, following recommended practices.
- Create and manage a project in Ansible Tower that gets playbooks and other project materials from an existing Git repository.
- Create and manage a job template that specifies a project and playbook, an inventory, and credentials that you can use to launch Ansible jobs on managed hosts.

SECTIONS

- Managing Ansible Project Materials Using Git (and Guided Exercise)
- Creating a Project for Ansible Playbooks (and Guided Exercise)
- Creating Job Templates and Launching Jobs (and Guided Exercise)

LAB

- Managing Projects and Launching Ansible Jobs

MANAGING ANSIBLE PROJECT MATERIALS USING GIT

OBJECTIVES

After completing this section, students should be able to create and manage Ansible Playbooks in a Git repository, following recommended practices.

INFRASTRUCTURE AS CODE

One key DevOps concept is the idea of *infrastructure as code*. Rather than managing infrastructure through the manual execution of commands, essential components are built and maintained through automated, programmatic procedures. Ansible projects and their playbooks are key tools which help implement this.

If Ansible projects are the code which is used to manage the infrastructure, then, in accordance with development best practices, a *version control system* such as Git should be used to track and control changes.

Version control allows administrators to implement a life cycle for the different stages of their infrastructure code, such as development, QA, and production. By managing infrastructure code with a version control tool, administrators can test their infrastructure code changes in noncritical development and QA environments to minimize surprises and disruptions when deployments are implemented in production environments.

Ansible Tower provides a central location from which Ansible playbooks can be run. It can pull projects containing those playbooks from a Git repository, and can even be configured so that particular Tower projects pull materials from specific branches of a Git repository.

INTRODUCING GIT

Git is a *distributed version control system* (DVCS) that allows developers to manage changes to files in a project in a collaborative manner. Each revision of a file is committed to the system. Old versions of files can be restored, and a log of who made the changes is maintained.

Version control systems provide many benefits, including:

- The ability to review and restore old versions of files
- The ability to compare two versions of the same file to identify changes
- A record or log of who made what changes at what time
- Mechanisms for multiple users to collaboratively modify files, resolve conflicting changes, and merge the changes together

Git is a *distributed version control system*. Each developer can start by *cloning* an existing shared project from a *remote repository*. This gives them a complete copy of the original remote repository as their *local repository*. This is a local copy of the entire history of the files in the version control system, not just the latest snapshot of the project files.

The developer makes edits in their *working tree*, a checkout of a single snapshot of the project. A set of related changes are then staged and committed to the local repository. At this point, no changes have been made to the shared remote repository.

When the developer is ready to share their work, they can *push* their changes to the remote repository. Alternatively, if their local repository is accessible from the network, they can ask the owner of the remote repository to *pull* the changes from the developer's repository to the remote repository.

Likewise, when a developer is ready to update their local repository with the latest changes to the remote repository, they can pull the changes (fetching them from the remote repository and merging them into their local work).

To use Git effectively, a user must be aware of three states that a file in the working tree can be in: *modified*, *staged*, or *committed*.

- *Modified*: the copy of the file in the working tree has been edited and is different from the latest version in the repository
- *Staged*: the modified file has been added to a list of changed files to commit as a set, but has not yet been committed
- *Committed*: the modified file has been committed to the local repository

Once the file is committed to the local repository, the commit can be pushed to or pulled by a remote repository.



Figure 13.1: The four areas where Git manages files



NOTE

The presentation in this section assumes that you are using Git from the command line of a Bash shell. A number of programming editors and IDEs have integrated Git support, and while the configuration and UI details may differ, they simply provide a different front end to the same workflow.

INITIAL GIT CONFIGURATION

Since Git users frequently modify projects with multiple contributors, Git records the user's name and email address on each of their commits. These values can be defined at a project level, but global defaults can also be set for a user. The **git config** command controls these settings. Using it with the **--global** option manages the default settings for all Git projects to which the user contributes by saving the settings in their **~/.gitconfig** file.

```
[peter@host ~]$ git config --global user.name 'Peter Shadowman'
```

```
[peter@host ~]$ git config --global user.email peter@example.com
```

If **bash** is the user's shell, another useful but optional thing to do is to configure your prompt to automatically modify itself to report the status of your working tree. The easiest way is to use the **git-prompt.sh** script shipped with the **git** package.

To modify your shell prompt in this way, add the following lines to your `~/.bashrc` file. If your current directory is in a Git working tree, the name of the current Git branch for the working tree is displayed in parentheses. If you have untracked, modified, or staged files that aren't committed in your working tree, the prompt will indicate this:

- (branch *) means a tracked file is modified
- (branch +) means a tracked file is modified and staged with **git add**
- (branch %) means untracked files are in your tree
- Combinations of markers are possible, such as (branch *+)

```
source /usr/share/git-core/contrib/completion/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=true
export GIT_PS1_SHOWUNTRACKEDFILES=true
export PS1='[\u@\h \w$(declare -F __git_ps1 &>/dev/null && __git_ps1 " (%s)")]\$ '
```

THE GIT WORKFLOW

When working on shared projects, the Git user clones an existing upstream repository with the **git clone** command. The path name or URL provided determines which repository is cloned into the current directory. A working tree is also created so that the directory of files is ready for revisions. Since the working tree is unmodified, it is initially in the clean state.

For example, the following command clones the repository **project.git** at **git.lab.example.com** by connecting using the SSH protocol and authenticating as user **git**:

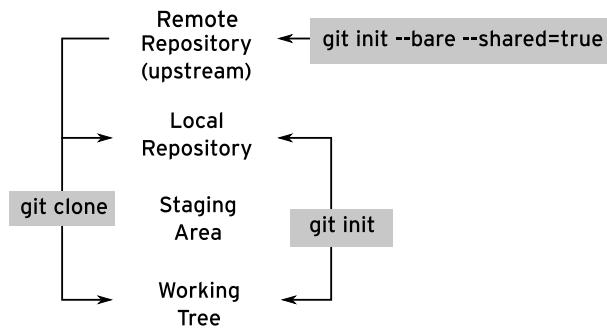
```
[peter@host ~]$ git clone git@git.lab.example.com:project.git
```



NOTE

Another way to start the Git workflow is to create a new, private project with the **git init** command. A project started in this way has no remote repository at first.

An advanced setup is to use **git init --bare** to create a *bare repository* on a server. This is used only as a remote repository by other developers, and does not have a local working tree and therefore is not usable as a local repository by anyone. The server also needs to be set up to allow users to clone, pull from, and push to the repository using the HTTPS or SSH protocol.

**Figure 13.2: Git subcommands used to create a repository**

As a developer works, new files are created and existing files are modified in the working tree. This changes the working tree to a dirty state. The **git status** command displays detailed information about which files in the working tree are modified but unstaged, untracked (new), or staged for the next commit.

The **git add** command stages files, preparing them to be committed. Only files that are staged to the staging area are saved in the repository on the next commit.

If a user is working on two changes at the same time, the files can be organized into two commits for better tracking of changes. One set of changes is staged and committed, and then the rest of the changes are staged and committed.

The **git rm** command removes a file from the working directory and also stages its removal from the repository on the next commit.

The **git reset** command removes a file from the staging area that has been added for the next commit. This command has no effect on the file's contents in the working tree.

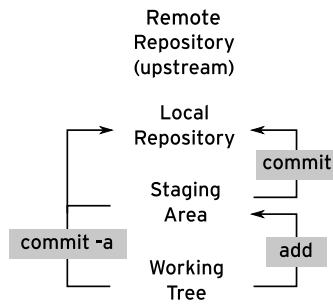
The **git commit** command commits the staged files to the local Git repository. A log message must be provided that explains why the current set of staged files is being saved. Failure to provide a message will abort the commit. Log messages do not have to be long, but they should be meaningful so that they are useful.



IMPORTANT

The **git commit** command by itself does not automatically commit changed files in the working tree.

The **git commit -a** command stages and commits *modified* files in one step. However, that command does not include any *untracked* (newly created) files in the directory. When you add a new file, you must explicitly **git add** that file to stage it the first time so that it is tracked for future **git commit -a** commands.

**Figure 13.3: Git subcommands that add/update local repository content**

The **git push** command uploads changes made to the local repository to the remote repository. One common way to coordinate work with Git is for all developers to push their work to the same, shared, remote repository.

Before Git pushes can work, the default push method must be defined. The following command sets the default push method to the **simple** method. This is the safest option for beginners.

```
[peter@host ~]$ git config --global push.default simple
```

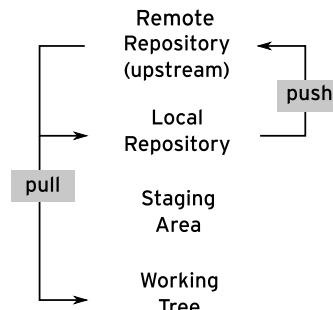
The **git pull** command fetches commits from the remote repository and to the local repository. It also merges the changes into your working tree.

This command should be run frequently to stay current with the changes that others are making to the project in the remote repository.

**NOTE**

An alternative approach to get commits from the remote repository is to use **git fetch** to download changes in the remote repository into a special *tracking branch* of your local repository, then to use **git merge tracking-branch** to merge the changes in the tracking branch to your current branch. There are a number of advantages and disadvantages to this approach.

In order to keep this discussion simple, this lesson defers discussing branches in depth, despite their importance in Git, and focus on the **git pull** approach.

**Figure 13.4: Git subcommands that interact with a remote repository**

Examining the Git Log

Part of the point of a version control system is to track a history of commits. Each commit is identified by a commit hash. The `git log` command displays the commit log messages with the associated ID hashes for each commit. The `git show commit-hash` command shows what was in the change set for a particular *commit-hash*. The entire hash doesn't need to be entered with the command, only enough of it to uniquely identify a particular commit in the repository. These hashes can also be used to revert to earlier commits or otherwise explore the version control system's history.

Git Quick Reference

COMMAND	DESCRIPTION
<code>git clone URL</code>	Clone an existing Git project from the remote repository at <i>URL</i> into the current directory.
<code>git status</code>	Display the status of modified and staged files in a working tree.
<code>git add file</code>	Stage a file for the next commit.
<code>git rm file</code>	Stage removal of a file for the next commit.
<code>git reset</code>	Unstage files for the next commit (the opposite of <code>git add</code>).
<code>git commit</code>	Commit the staged files to the local repository with a descriptive message.
<code>git push</code>	Push changes in the local repository to the remote repository.
<code>git pull</code>	Fetch updates from the remote repository to the local repository and merge them into the working tree.



IMPORTANT

This has been a highly simplified introduction to Git. It has made some assumptions and avoided discussion of the important topics of branches and merging. Some of these assumptions included:

- The local repository was cloned from a remote repository
- Only one local branch is in use
- The local branch is configured to fetch from and push to a branch on the original remote repository
- Write access is provided to the remote repository, such that `git push` works

Links to more detailed information and tutorials on how to use Git are available in the References at the end of this section.

STRUCTURING ANSIBLE PROJECTS IN GIT

It's highly recommended that Ansible playbooks, roles, and any other details used by an Ansible Tower project are stored in a version control system like Git. This provides an easy way for the administrative team to collaborate on playbook development and share their work with other

teams. In addition, the version control system itself provides a record of changes that can be used as an audit trail.

Each project should have its own Git repository. However, projects should not assume they can import roles or content from other projects. One way to import roles which are used by multiple projects would be to have each project use Git submodules (see **git-submodule(1)** for more information). This also helps limit the size of the download when a playbook is updated, and helps contributors avoid confusing the code for different projects.

The basic structure of the files in that repository should follow the recommended practices for Ansible documented at http://docs.ansible.com/ansible/playbooks_best_practices.html. For example, the directory structure might look something like this:

```
library/          # for custom modules (optional)
filter_plugins/  # for custom filter plugins (optional)

site.yml         # MASTER PLAYBOOK includes other playbooks
webservers.yml   # playbook for webserver tier
dbservers.yml    # playbook for dbserver tier

roles/           # directory for roles
  webserver/      # a particular role
    tasks/
      main.yml     # tasks for the role, can include other files
    defaults/
      main.yml     # default low-priority variables for the role
    templates/
      httpd.conf.j2 # a Jinja2 template used by the role
    files/
      motd         # a file used by the role
    handlers/
      main.yml     # handlers used by the role
    meta/
      main.yml     # role information and dependencies
...additional roles...
```

Instead of using **host_vars** and **group_vars** directories in the project, a better practice is to store variables with the Inventory object in Tower.



IMPORTANT

Playbooks must not use the **vars_prompt** feature to set variables interactively, because it does not work with Ansible Tower. Instead, use Tower's Survey functionality, as discussed later in this course.

Ansible Tower grants Users access to Ansible projects stored in version control systems through the use of Credentials. These credentials allow access to data stored in Git at a repository level. The next section covers how to configured Credentials.

Administrators need to keep in mind that Tower Users have all-or-nothing access to Git repositories. When granted Tower Credentials to access a Git repository, a Tower user is able to access all of the repository's contents. Therefore, only the playbooks and roles meant to be shared with an intended audience should be kept in the same Git repository.



REFERENCES

gittutorial(7) and git(1) man pages

Git

<https://git-scm.com>

Pro Git by Scott Chacon and Ben Straub (free book)

<https://git-scm.com/book/en/v2>

Further information is available in the *GitHub Getting Started Tutorial* at

<http://try.github.io>

A useful hands-on tool for learning about branching and merging in Git is at

<http://learngitbranching.js.org>

Ansible Tower User Guide: Best Practices

http://docs.ansible.com/ansible-tower/latest/html/userguide/best_practices.html

Ansible User Guide: Best Practices

http://docs.ansible.com/ansible/playbooks_best_practices.html

► GUIDED EXERCISE

MANAGING ANSIBLE PROJECT MATERIALS USING GIT

In this exercise, you will clone an existing Git repository that contains an Ansible Playbook, make edits to files in that repository, commit the changes to your local repository, and push them to the original repository.

OUTCOMES

You should be able to use basic **git** commands to manage a version controlled project with Ansible Tower.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on workstation and run **lab provision-git setup**. This setup script installs the package for Git and creates the Git repository needed for the exercise.

```
[student@workstation ~]$ lab provision-git setup
```

- 1. Perform basic configuration of Git by setting the following user name, email address, and default push method using **git config**:

```
[student@workstation ~]$ git config --global user.name 'Daniel George'  
[student@workstation ~]$ git config --global user.email daniel@lab.example.com  
[student@workstation ~]$ git config --global push.default simple
```

- 2. Check your configuration using **git config**. The output should resemble the following:

```
[student@workstation ~]$ git config --global -l  
user.name=Daniel George  
user.email=daniel@lab.example.com  
push.default=simple
```

- 3. Create and change directory to a new directory, called **git-repos**, for your Git repositories.

```
[student@workstation ~]$ mkdir git-repos && cd git-repos
```

- 4. Clone and examine the repository called `my_webservers_DEV`.

- 4.1. Clone the repository using **git clone**. This creates a directory called `my_webservers_DEV` in your current directory. This new directory contains a playbook intended to setup an Apache web server.

```
[student@workstation git-repos]$ git clone \
```

```
> git@git.lab.example.com:my_webservers_DEV.git
Cloning into 'my_webservers_DEV'...
Warning: Permanently added 'git.lab.example.com' (ECDSA) to the list of known
hosts.
remote: Counting objects: 27, done.
remote: Compressing objects: 100% (27/27), done.
remote: Total 27 (delta 5), reused 0 (delta 0)
Receiving objects: 100% (27/27), done.
Resolving deltas: 100% (5/5), done.
```

- 4.2. Change directory to the new directory. This is the root directory of the Git repository.

```
[student@workstation git-repos]$ cd my_webservers_DEV
```

- 4.3. Take a look at the files in that repository using the **tree** command. The **apache-setup.yml** file is the playbook. There are two Jinja2 templates that the playbook uses to build static files. The **httpd.conf.j2** template is used to generate the Apache server configuration. The **index.html.j2** template is used to generate a basic index page to be served by the server.

```
[student@workstation my_webservers_DEV]$ tree
.
└── apache-setup.yml
└── templates
    ├── httpd.conf.j2
    └── index.html.j2
```

- 5. Edit the **index.html.j2** template so it displays **HELLO WORLD** at the bottom of the page and then verify your modification.

- 5.1. Using a text editor, open the **templates/index.html.j2** template file for editing.

```
[student@workstation my_webservers_DEV]$ vi templates/index.html.j2
```

- 5.2. Append the string **HELLO WORLD** to the end of the file. The **index.html.j2** file should look like this after the modification:

```
{{ apache_test_message }} {{ ansible_distribution }}
{{ ansible_distribution_version }} <br>
Current Host: {{ ansible_hostname }} <br>
Server list: <br>
{% for host in groups['all'] %}
{{ host }} <br>
{% endfor %}
HELLO WORLD <br>
```

- 5.3. Save the changes made to the file and then exit from the text editor.

- 5.4. Use **git status** to check the status of your modifications. Git shows that you have unstaged changes.

```
[student@workstation my_webservers_DEV]$ git status
# On branch master
```

```
# Changes not staged for commit:  
#   (use "git add <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working directory)  
#  
#       modified:   templates/index.html.j2  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

► 6. Add the template to the staging area and check the status of your modifications.

- 6.1. Use **git add** to add the template to the staging area.

```
[student@workstation my_webservers_DEV]$ git add templates/index.html.j2
```

- 6.2. Use **git status** to check the status of your modifications. Git shows that you have modifications in the staging area which are ready to be committed.

```
[student@workstation my_webservers_DEV]$ git status  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       modified:   templates/index.html.j2  
#
```

► 7. Commit your changes and check the status of your modifications.

- 7.1. Use **git commit** with the commit message **My first commit**:

```
[student@workstation my_webservers_DEV]$ git commit -m "My first commit"  
[master 918ceb7] My first commit  
 1 file changed, 1 insertion(+)
```

- 7.2. Use **git status** to check the status of your modifications. Git shows that there are no more modifications to be either staged or committed on the local repository.

```
[student@workstation my_webservers_DEV]$ git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 1 commit.  
#   (use "git push" to publish your local commits)  
#  
nothing to commit, working directory clean
```

► 8. Push the changes to the remote repository using **git push**.

```
[student@workstation my_webservers_DEV]$ git push  
Counting objects: 7, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (4/4), 427 bytes | 0 bytes/s, done.  
Total 4 (delta 1), reused 0 (delta 0)  
To git@git.lab.example.com:my_webservers_DEV.git
```

```
c414fe9..918ceb7 master -> master
```

- ▶ 9. Change directory to the parent directory and clone the repository into a new directory called **my_webservers_DEV_2** to verify the changes have been pushed to the remote repository.

```
[student@workstation my_webservers_DEV]$ cd ..
[student@workstation git-repos]$ git clone \
> git@git.lab.example.com:my_webservers_DEV.git my_webservers_DEV_2
```

- ▶ 10. Verify that **HELLO WORLD** appears in the template.

```
[student@workstation git-repos]$ grep "HELLO WORLD" my_webservers_DEV_2/templates/
index.html.j2
HELLO WORLD <br>
```

- ▶ 11. Make a change to the remote repository from the second clone.

- 11.1. Change directory to the **my_webservers_DEV_2** clone directory.

```
[student@workstation git-repos]$ cd my_webservers_DEV_2
```

- 11.2. Using a text editor, remove the **HELLO WORLD** line from the **templates/index.html.j2** file.

- 11.3. Add the template to the staging area using **git add**.

```
[student@workstation my_webservers_DEV_2]$ git add templates/index.html.j2
```

- 11.4. Commit the changes using **git commit** with the comment **Removed HELLO WORLD**.

```
[student@workstation my_webservers_DEV_2]$ git commit -m "Removed HELLO WORLD"
[master 3a4cb00] Removed HELLO WORLD
 1 file changed, 1 deletion(-)
```

- 11.5. Push the changes to the remote repository.

```
[student@workstation my_webservers_DEV_2]$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 414 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 3 (delta 1)
To git@git.lab.example.com:my_webservers_DEV.git
 918ceb7..3a4cb00  master -> master
```

- 12. Use **git pull** to pull the latest changes from the remote repository.

12.1. Change directory to the first clone of the repository, **my_webservers_DEV**.

```
[student@workstation my_webservers_DEV_2]$ cd ../my_webservers_DEV
```

12.2. Verify that the **HELLO WORLD** line is present in the **index.html.j2** file.

```
[student@workstation my_webservers_DEV]$ grep "HELLO WORLD" templates/index.html.j2
HELLO WORLD <br>
```

12.3. Pull the latest changes using **git pull**.

```
[student@workstation my_webservers_DEV]$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git.lab.example.com:my_webservers_DEV
  918ceb7..3a4cb00  master      -> origin/master
Updating 918ceb7..3a4cb00
Fast-forward
 templates/index.html.j2 | 1 -
 1 file changed, 1 deletion(-)
```

12.4. Verify that the **HELLO WORLD** line is no longer present in the **index.html.j2** file.

```
[student@workstation my_webservers_DEV]$ grep "HELLO WORLD" templates/index.html.j2
[student@workstation my_webservers_DEV]$
```

This concludes the guided exercise.

CREATING A PROJECT FOR ANSIBLE PLAYBOOKS

OBJECTIVES

After completing this section, students should be able to create and manage a project in Ansible Tower that gets playbooks and other project materials from an existing Git repository.

Figure 13.4: Creating a Project for Ansible playbooks

PROJECTS

An Ansible project represents at least one playbook and its associated collection of related playbooks and roles. Whether or not Ansible Tower is being used, it is a good practice for these materials to be managed together in a version control system. The design of Ansible Tower assumes that most Ansible projects are managed in a version control system for this reason, and can automatically retrieve updated materials for a project from several commonly used version control systems.

In the Ansible Tower web interface, each Ansible project is represented by a Project resource. The Project is configured to retrieve these materials from a version control system (also referred to by Ansible Tower as a *source control management* or SCM system). Ansible Tower supports the ability to download and automatically get updates of project materials from SCMs using Git, Subversion, or Mercurial.



NOTE

It is possible to simply copy projects to the Ansible Tower server in a location known as the *Project Base Path*. Configured by `/etc/tower/settings.py`, this directory is located by default at `/var/lib/awx/projects`.

This is not a recommended practice, however. Updating such projects requires manual intervention outside Ansible Tower interfaces. It also requires that project administrators have direct access to make changes in the operating system environment on the Ansible Tower, which reduces the security of the Ansible Tower server. It is better to have Ansible Tower get project materials from an SCM system.

CREATING A PROJECT

The following is the procedure for creating a Project to share a collection of Ansible playbooks and roles managed in an existing Git repository. The hands-on exercise following this section covers this in more detail.

1. Log in to the Ansible Tower web interface as a user with **Admin** role on an Organization.
2. Click **Projects** in the left quick navigation bar to go to the Projects management screen.
3. Click the **+** button to create the new Project.
4. Enter a unique name for the Project in the **NAME** field.
5. Optionally, enter a description for the Project in the **DESCRIPTION** field.
6. Click the magnifying glass icon next to the **ORGANIZATION** field to display a list of Organizations within Ansible Tower. Select an Organization from the list and click **SELECT**.

7. In the SCM TYPE drop-down menu, select the Git option.
8. Under the SOURCE DETAILS section, enter the location of the Git repository in the SCM URL field.
9. Optionally, in the SCM BRANCH/TAG/COMMIT field, specify the branch, tag or commit of the repository to obtain the contents from.
10. If authentication is required to access the Git repository, click the magnifying glass icon next to the SCM CREDENTIAL field to display a list of available SCM Credentials. Select an SCM Credential from the list and click SELECT. The creation of SCM Credentials is discussed later in this section.
11. Lastly, select the desired action to be taken to update the Project against its SCM source. Available options are **Clean**, **Delete on Update**, and **Update Revision on Launch**. These three options are discussed in further detail at the end of this section.
12. Click SAVE to finalize the creation of the Project.

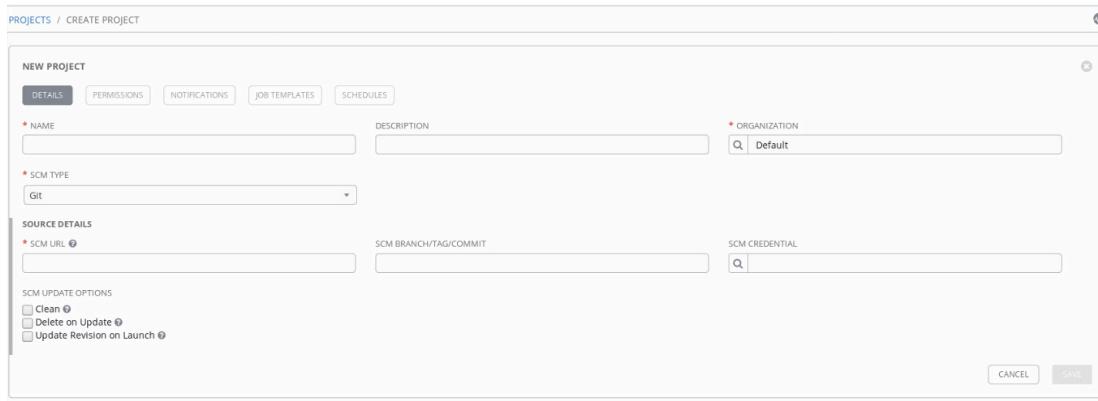


Figure 13.5: New project

PROJECT ROLES

Users are granted permissions to Project resources through assigned roles on the Project resource. Users can be directly assigned roles, or they can be assigned. As always, assignment of roles can be made directly to a user or indirectly through a team. For example, in order for a user to gain permissions on a specific Project, they must be assigned or inherit a role for that Project.

The following are the list of available project roles:

Admin

The **Admin** role grants users full access to a Project. When granted this role on a Project, a user can delete the Project and modify its properties, permissions included. In addition, this role also grants users the **Use**, **Update**, and **Read** roles, which are discussed later in this section.

Use

The **Use** role grants users the ability to use a Project in a Template resource. Use of a Project in a Template resource will be discussed in detail in a later section. This role also grants users the permissions associated with the Project **Read** role.

Update

The **Update** role grants users the ability to manually update or schedule an update of a Project's materials from its SCM source. This role also grants users the permissions associated with the Project **Read** role.

Read

The **Read** role grants users the ability to view the details, permissions, and notifications associated with a Project.

MANAGING PROJECT ACCESS

When a Project is first created, it is only accessible by users with the **Admin** or **Auditor** role in the Project's Organization.

Other access by users must be specifically configured. Roles cannot be assigned when the Project is created, but must be added by editing the Project.

Roles are assigned in the PERMISSIONS section of the Project editor screen. The following procedure details the steps to set roles for a Project:

1. Log in as a user with **Admin** role for the Organization in which the Project was created.
2. Click **Projects** in the left quick navigation bar to display the list of Projects.
3. Click the pencil icon for the Project to edit to enter the Project editor screen.
4. On the Project editor screen, click the **PERMISSIONS** button to enter the permissions editor.
5. Click the **+** button to add permissions.
6. In the user and team selection screen, click either **USERS** or **TEAMS** and then select the users or teams to be granted permissions.
7. Click **KEY** to display the list of Project roles and their definitions.
8. Click the **SELECT ROLES** drop-down menu and select the desired Project role for each user or team.
9. Click **SAVE** to finalize the changes to permissions.

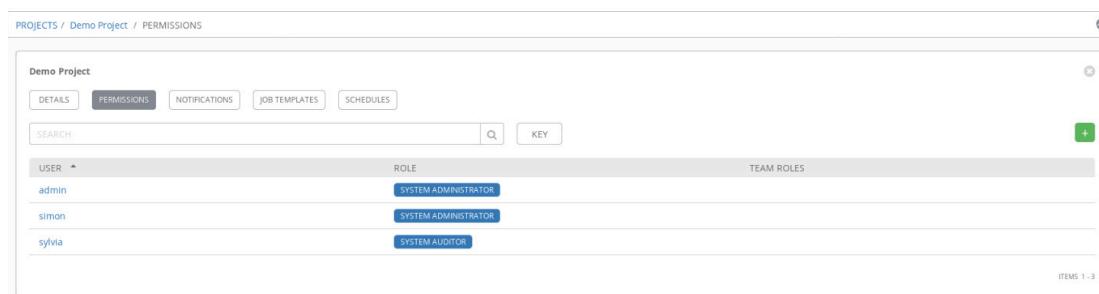


Figure 13.6: Assigning Project roles to a user


NOTE

Roles for Projects can also be added through either the user or team management screens.

CREATING SCM CREDENTIALS

In a previous section of this course, you learned how to use Machine Credentials, which store the authentication information playbooks need to connect to managed hosts and perform authentication tasks on them. Source Control Credentials, also called SCM Credentials, store authentication information that Ansible Tower can use to access project materials stored in a

version control system like Git. SCM Credentials store the user name, and the password or private key (and private key passphrase, if any) needed to authenticate access to the source control repository.

This is an outline of the procedure you would use to create an SCM Credential so that Ansible Tower can retrieve playbooks, roles, or other materials from a Git repository for a Project. Do not do this now. The hands-on exercise following this section covers this in more detail.

1. Log in as a user with the appropriate role assignment:
 - If creating a private SCM Credential, there are no specific role requirements.
 - If creating an SCM Credential belonging to an Organization, log in as a user with **Admin** role for the Organization.
2. Click **Credentials** in the left quick navigation bar to enter the Credentials management interface.
3. On the **CREDENTIAL** screen, click **+** to create a new Credential.
4. On the **CREATE CREDENTIAL** screen, enter the required information for the new Credential.
 1. Enter a unique name for the Credential in the **NAME** field.
 2. If creating an Organization Credential, click the magnifying glass next to the **ORGANIZATION** field, select the Organization to create the Credential in, and then click **SELECT**. Skip this step if creating a private Credential.
 3. Click the **CREDENTIAL TYPE** drop-down menu and select the **Source Control** Credential type, and then click **SELECT**.
5. Once a **Source Control** Credential type is selected in the previous step, the appropriate fields appear in the **TYPE DETAILS** section.

Enter authentication data into their respective fields. For example, you may need to specify a **USERNAME**. If a password is needed, you must enter that in the **PASSWORD** field. If you are using an SSH private key to authenticate, either copy and paste or drag and drop your private key into the **SCM PRIVATE KEY** field. That key can be a passphrase encrypted SSH private key, in which case you can provide the passphrase to Ansible Tower in the **PRIVATE KEY PASSPHRASE** field.
6. Click the **SAVE** button to finalize the creation of the new SCM Credential.

The screenshot shows the 'CREATE CREDENTIAL' dialog box. At the top, there are tabs for 'DETAILS' (which is selected) and 'PERMISSIONS'. Below the tabs are fields for 'NAME' (with placeholder 'My New Credential'), 'DESCRIPTION' (empty), and 'ORGANIZATION' (with a 'SELECT AN ORGANIZATION' dropdown). Under the 'CREDENTIAL TYPE' section, 'Source Control' is selected. The 'TYPE DETAILS' section contains fields for 'USERNAME' (empty) and 'PASSWORD' (with a 'SHOW' button). Below these is a note about dragging an SSH private key file into the 'SCM PRIVATE KEY' field. At the bottom of the dialog are 'PRIVATE KEY PASSPHRASE' (empty) and 'SHOW' buttons, along with 'CANCEL' and 'SAVE' buttons.

Figure 13.7: New SCM Credential

SCM CREDENTIAL ROLES

Just like Machine Credentials, private SCM Credentials are only usable by their creators and **System Administrator** and **System Auditor** users. SCM Credentials assigned to an Organization can be shared with other users by assigning either users or teams the appropriate roles for that credential.

The following is the list of roles available for providing Users access to SCM Credentials:

Admin

The **Admin** role grants users full permissions over an SCM Credential. These permissions include deletion and modification of the SCM Credential. This role also grants Users permissions associated with the Credential **Use** and **Read** roles.

Use

The **Use** role grants Users the ability to associate an SCM Credential with a Project resource. This role also grants users the permissions associated with the Credential **Read** role.

The **Use** role does not control whether a user can themselves use the SCM Credential to *update* a Project, only whether they can assign that SCM Credential so that it can then be used by someone who has the **Update** role on the Project.

For example, if an SCM Credential is associated with a Project, any user assigned the **Update** role on the Project can use the associated SCM Credential without being granted **Use** role on the Credential.

Read

The **Read** role grants users the ability to view the details of an SCM Credential.

MANAGING ACCESS TO SCM CREDENTIALS

When an Organization SCM Credential is first created, it is only accessible by users with either the **Admin** or **Auditor** role in the Organization to which the Credential is assigned. Additional access for other users must be specifically configured.

The assignment of SCM Credential roles to users or teams dictates who has permissions to an SCM Credential belonging to an Organization. These permissions cannot be assigned at the time of the SCM Credential's creation. They are adjusted after its creation by editing the Credential.

Roles are assigned through the PERMISSIONS section of the Credential editor screen. The following procedure details the steps for granting permissions to an SCM Credential that has been assigned to an Organization, after the credential's creation.

1. Log in as a user with **Admin** role on the Organization that the SCM Credential belongs to.
2. Click **Credentials** in the left quick navigation bar to enter the Credentials management interface.
3. Click the name of the SCM Credential to edit in order to enter the Credential editor screen.
4. On the Credential editor screen, click the **PERMISSIONS** button to enter the permissions editor.
5. Click the **+** button to add permissions.
6. In the user and team selection screen, click either **USERS** or **TEAMS** and then select the users or teams to be granted permissions.
7. Click **KEY** to display the list of Credential roles and their definitions.

8. Click the SELECT ROLES drop-down menu and select the desired Credential role for each user or team.
9. Click SAVE to finalize the changes to permissions.

USER	ROLE	TEAM ROLES
admin	ADMIN SYSTEM ADMINISTRATOR	
simon	SYSTEM ADMINISTRATOR	
sylvia	SYSTEM AUDITOR	

Figure 13.8: Assigning an SCM Credential role to a user



IMPORTANT

Permissions for SCM Credentials can also be added through either the user or team management screens under Ansible Tower's Settings interface.

UPDATING PROJECTS

An SCM Project resource in Ansible Tower represents a copy of playbooks and roles obtained from an SCM source. Since modifications to the contents of these playbooks and roles are managed in an external SCM system, their respective counterparts in a Ansible Tower Project must be updated routinely from the SCM source to reflect new changes.

There are several ways to update SCM Project resources in Ansible Tower. As previously mentioned, Projects can be configured to update from their SCM sources by choosing one of three SCM update options in the Project's detail screen.

Clean

This SCM update option removes local modifications before getting the latest revision from the source control repository.

Delete on Update

This SCM update option completely removes the local Project repository on Ansible Tower before getting the latest revision from the source control repository. This takes longer than Clean for large repositories.

Update on Launch

This SCM update option updates the Project from the source control repository each time the Project is used to launch a job. The update itself is tracked as a separate job by Ansible Tower.

You can manually update a Project to the latest version in the source control repository, if you do not want to use these automatic settings. The following procedure outlines the steps needed to manually update a Project from its SCM source:

1. Log in as a user with **Update** role on the Project.
2. Click the **Projects** in the left quick navigation bar to enter the Projects management interface.

3. In the table of Projects, a double arrow icon appears under the ACTIONS column if the user has the **Update** role for a given Project.
4. To trigger a manual, immediate update on a Project, click on its double-arrow icon to initiate an update of its contents against its SCM source.

NAME	TYPE	REVISION	LAST UPDATED	ACTIONS
Demo Project	Manual		10/18/2018 10:47:12 AM	Get latest SCM revision
My Webservers DEV	Git	cc5f83c	10/26/2018 3:32:41 PM	Get latest SCM revision

ITEMS 1 - 2

Figure 13.9: Updating an SCM project

**IMPORTANT**

The **Update** role only dictates whether a user can manually trigger an update of a Project against its SCM source. It does not impact the update behavior configured by the Project's SCM update option. For example, a Project configured with an **Update on Launch** SCM update option still performs updates even when the Project is used by a user who has not been granted the **Update** role on the Project.

SUPPORT FOR ANSIBLE ROLES

Projects may specify external Ansible roles that are stored in Ansible Galaxy or other source control repositories as dependencies. At the end of a Project update, if a project's repository includes a **roles** directory that contains a valid **requirements.yml** file, Red Hat Ansible Tower will automatically run **ansible-galaxy** to install the roles:

```
ansible-galaxy install -r roles/requirements.yml -p ./roles/ --force
```

For more information and syntax examples of the **requirements.yml** file, refer to the Installing multiple roles from a file section of the Ansible Galaxy guide at https://docs.ansible.com/ansible/latest/reference_appendices/galaxy.html#installing-multiple-roles-from-a-file

**REFERENCES****Ansible Tower User Guide**

<http://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

► GUIDED EXERCISE

CREATING A PROJECT FOR ANSIBLE PLAYBOOKS

In this exercise, you will create a new Source Control credential, a new project, and assign a role to one of your teams allowing team members to use that project.

OUTCOMES

You should be able to create a Project which will allow Ansible Tower to leverage an external Git repository containing a playbook.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run **`lab provision-project setup`**. This setup script ensures that the `workstation` and `tower` virtual machines are started.

```
[student@workstation ~]$ lab provision-project setup
```

- ▶ 1. Log into the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
- ▶ 2. Create the new Source Control Credential needed to create a new Project.
 - 2.1. In the left navigation bar, click **Credentials** to manage Credentials.
 - 2.2. Click the **+** button to add a new Credential.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	student-git
DESCRIPTION	Student Git Credential
ORGANIZATION	Default
TYPE	Source Control
USERNAME	git
SCM PRIVATE KEY	Copy the contents of the <code>/home/student/.ssh/lab_rsa</code> private key file on <code>workstation</code> into this field

- 2.4. Leave the other fields untouched and click **SAVE** to create the new Credential.

- 3. Create a new Project called "My Webservers DEV".
- 3.1. In the left navigation bar, click the Projects quick navigation link.
 - 3.2. Click the + button to add a new Project.
 - 3.3. On the next screen, fill in the details as follows:
- | FIELD | VALUE |
|----------------|--|
| NAME | My Webservers DEV |
| DESCRIPTION | Development Webservers Project |
| ORGANIZATION | Default |
| SCM TYPE | Git |
| SCM URL | ssh://git.lab.example.com/home/git/my_webservers_DEV.git |
| SCM CREDENTIAL | student-git |
- 3.4. Click SAVE to create the new Project. This automatically triggers the SCM update of the Project. Ansible Tower uses the values provided in the SCM URL and SCM CREDENTIAL fields to pull down a local copy of that repository.
- 4. Observe the automatic SCM update of the project My Webservers DEV.
- 4.1. Scroll down the page and wait a couple of seconds. In the list of Projects, there is a status icon left of the Project, My Webservers DEV. This icon is white at the start, red with an exclamation mark when it fails, and green when it succeeds.
 - 4.2. Click on the status icon to show the detailed status page of the SCM update job. As you can see in the DETAILS window, the SCM update job runs like any other Ansible playbook.
 - 4.3. Verify that the STATUS of the job in the DETAILS section shows **Successful**.

► 5. Give the Developers Team **Admin** role on the Project, My Webservers DEV.

- 5.1. In the left navigation bar, click the Projects quick navigation link.
- 5.2. On the same line as the Project, My Webservers DEV, click the pencil icon on the right to edit the Project.
- 5.3. On the next page, click PERMISSIONS to manage the Project's permissions.
- 5.4. Click the + button on the right to add permissions.
- 5.5. Click TEAMS to display the list of available teams.
- 5.6. In the first section, check the box next to the Developers team. This causes the team to display in the second section underneath the first one.
- 5.7. In the second section below, select the **Admin** role from the drop-down list.
- 5.8. Click SAVE to make the role assignment. This redirects you to the list of permissions for the Project, My Webservers DEV, which now shows that all members of the Developers team are assigned the **Admin** role on the Project.
- 5.9. Click the Log Out icon to exit the Tower web interface.

This concludes the guided exercise.

CREATING JOB TEMPLATES AND LAUNCHING JOBS

OBJECTIVES

After completing this section, students should be able to create and manage a job template that specifies a project and playbook, an inventory, and credentials that you can use to launch Ansible jobs on managed hosts.

Figure 13.9: Creating Job Templates and launching Jobs

JOB TEMPLATES, PROJECTS, AND INVENTORIES

In Red Hat Ansible Tower, a *Job Template* is a template that you can use to launch jobs which run playbooks. A Job Template associates a playbook from a Project with an Inventory of hosts, Credentials for authentication, and other parameters used when you launch an Ansible job to run that playbook. Whether a user can launch jobs or create job templates with particular Projects and Inventories depends on the roles that you have assigned them. When granted the **Use** role, users can use a Job Template to associate Projects with Inventories.

A Job Template defines the parameters for the execution of an Ansible job. An Ansible job executes a playbook against a set of managed hosts. Therefore, a Job Template must define what Project provides the playbook and what Inventory contains the list of managed hosts.

Additionally, the Job Template must also define the Machine Credential which will be used to authenticate to the managed hosts. Like Projects and Inventories, a user must have the **Use** role assigned to a Machine Credential to be able to associate it to a Job Template.

Once defined, a Job Template allows for the repeated execution of a job and is therefore ideal for routine execution of tasks. Since the Project, Inventory, and Machine Credential parameters are part of the Job Template definition, the job is ensured to run the same way each time.

CREATING JOB TEMPLATES

Unlike other Ansible Tower resources, Job Templates do not directly belong to an Organization but are used by a Project that belongs to an Organization. A Job Template's relationship to an Organization is determined by the Project that it uses. Therefore, you do not need to have the **Admin** role in an Organization to create a Job Template. Instead, you only need to have the **Use** role for the Project that you assign to the Job Template.

Since a Job Template has to be defined with an Inventory, Project, and Machine Credential, a User can only create a Job Template if they have **Use** roles assigned to one or more of each of these three Ansible Tower resources. The following procedure details how to create a Job Template for running a playbook on a set of managed hosts with a focus on just the mandatory parameters (optional parameters are discussed later):

1. Log in to the Ansible Tower web interface as a user who has been assigned the **Use** role for the Inventory, Project, and Machine Credential resources.
2. Click the **Templates** in the left quick navigation bar to go to the Templates management interface.
3. Click the **+** button and then select **Job Template**.
4. Enter a name for the Job Template in the **NAME** field.

5. Select Run as the JOB TYPE.
6. Specify the managed hosts that the job will be executed against.
 1. Click the magnifying glass icon for the INVENTORY field.
 2. Select the desired Inventory and then click SELECT.
7. Specify the Project containing the playbook that the job will execute.
 1. Click the magnifying glass icon for the PROJECT field.
 2. Select the desired Project and then click SELECT.
8. Specify the playbook that the job will execute.
 1. Click the drop-down menu for the PLAYBOOK field. All playbooks contained in the Project selected in the previous field are listed.
 2. Select the desired playbook.
9. Specify the Credential required for authenticate against the managed hosts.
 1. Click the magnifying glass icon for the CREDENTIAL field.
 2. Select the desired Credential and then click SELECT.
10. Select the desired setting from the drop-down menu for the VERBOSITY field. This determines the level of detail that is generated in the output of the job run.
11. Click SAVE to finalize creation of the new Job Template.

The screenshot shows the 'CREATE JOB TEMPLATE' dialog box. At the top, there are tabs for DETAILS, PERMISSIONS, NOTIFICATIONS, COMPLETED JOBS, SCHEDULES, and ADD SURVEY. The DETAILS tab is selected. The form contains the following fields:

- NAME:** A text input field.
- DESCRIPTION:** A text input field.
- JOB TYPE:** A dropdown menu set to "Run".
- PROMPT ON LAUNCH:** A checkbox.
- INVENTORY:** A dropdown menu with a magnifying glass icon.
- FORKS:** A dropdown menu set to "DEFAULT".
- PROJECT:** A dropdown menu with a magnifying glass icon.
- PLAYBOOK:** A dropdown menu with a magnifying glass icon.
- LIMIT:** A dropdown menu.
- CREDENTIAL:** A dropdown menu with a magnifying glass icon.
- FORKS:** A dropdown menu.
- JOB TAGS:** A dropdown menu.
- SKIP TAGS:** A dropdown menu.
- LABELS:** A text input field.
- INSTANCE GROUPS:** A dropdown menu with a magnifying glass icon.
- SHOW CHANGES:** A dropdown menu set to "OFF".
- PROMPT ON LAUNCH:** A checkbox.
- OPTIONS:** A group of checkboxes:
 - Enable Privilege Escalation
 - Allow Provisioning Callbacks
 - Enable Concurrent Jobs
 - Use Fact Cache
- EXTRA VARIABLES:** A dropdown menu with "YAML" and "JSON" options. Below it is a preview area showing "1" and an ellipsis (...).
- PROMPT ON LAUNCH:** A checkbox.

Figure 13.10: A new Job Template

MODIFYING JOB EXECUTION

A Job Template has other settings you can use to adjust how Ansible Tower runs the playbook when the template is launched:

DESCRIPTION

This field is used to store an optional description of the Job Template.

FORKS

Use this field to specify the **forks** setting that controls the number of parallel processes to allow during playbook execution. This is the equivalent of the **-f** or **--forks** option for the **ansible-playbook** command. A value of **0** causes the default setting from the Ansible configuration file to be used.

LIMIT

Use this field to restrict the list of managed hosts provided by the Job Template's Inventory. Filtering is accomplished by supplying a host pattern as a value for this field. This is the equivalent of the **-l** or **--limit** option for the **ansible-playbook** command.

JOB TAGS

This field accepts a comma separated list of tags which exist in a playbook. Tags are used to identify distinct portions of a playbook. By specifying a list of tags in this field, you can selectively execute only certain portions of a playbook. This is the equivalent of the **-t** or **--tags** option for the **ansible-playbook** command.

SKIP TAGS

This field accepts a comma-separated list of tags that exist in a playbook. By specifying a list of tags in this field, you can selectively skip certain portions of a playbook during its execution. This is the equivalent of the **--skip-tags** option for the **ansible-playbook** command.

LABELS

Labels are names that you can attach to Job Templates to help you group or filter Job Templates.

Enable Privilege Escalation

When enabled, this check box causes the playbook to be executed with escalated privileges. This is the equivalent of the **--become** option for the **ansible-playbook** command.

Allow Provisioning Callbacks

When enabled, this check box results in the creation of a provisioning callback URL on Ansible Tower which can be used by hosts to request a configuration update using the Job Template.

Enable Concurrent Jobs

When enabled, this check box allows for multiple, simultaneous executions of this Job Template.

Use Fact Cache

When enabled, this check box causes the usage of cached facts and stores newly discovered facts in the fact cache on Ansible Tower. Fact caching is discussed in more detail later in this course.

EXTRA VARIABLES

Equivalent to the **-e** or **--extra-vars** options for the **ansible-playbook** command, this field can be used to pass extra command line variables to the playbook executed by a job. These extra variables are defined as key/value pairs using either YAML or JSON.

PROMPTING FOR JOB PARAMETERS

When executing playbooks from the command line using the **ansible-playbook** command, administrators have the ability to modify playbook execution through the use of command-line options. Ansible Tower allows for some of this flexibility by allowing certain parameters in Job Templates to prompt for user input at the time of job execution. This *Prompt on launch* option is available for:

- JOB TYPE
- INVENTORY

- CREDENTIAL
- LIMIT
- VERBOSITY
- JOB TAGS
- SKIP TAGS
- EXTRA VARIABLES

The flexibility to change job parameters at the time of job execution encourages playbook reuse. For example, rather than creating multiple job templates to run the same playbook on different sets of managed hosts, one only needs a single job template which has the **Prompt on launch** option enabled for the Inventory field. When the job is launched, the user executing the job is given the option to specify an Inventory to execute the playbook on. When prompted, users are only able to select from Inventories that they have been assigned the **Use** role on.

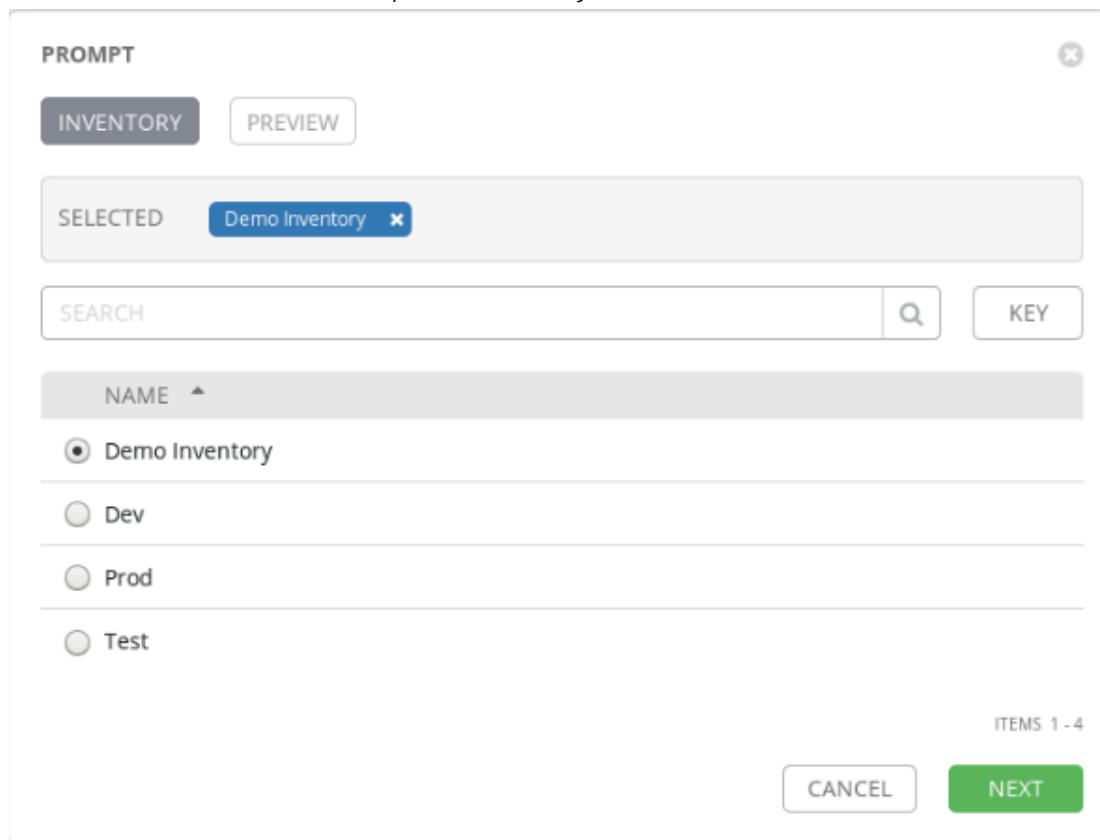


Figure 13.11: Prompting for Inventory on launch

JOB TEMPLATES ROLES

There are three roles used to control user access to Job Templates.

Admin

The **Admin** role provides a user the ability to delete a Job Template or edit its properties, including its associated permissions. This role also grants permissions associated with the Job Template **Execute** and **Read** roles.

Execute

The **Execute** role grants users permission to execute a job using the Job Template. It also grants the Users permission to schedule a job using the Job Template. This role also grants permissions associated with the Job Template **Read** role.



IMPORTANT

A Job Template makes use of other Ansible Tower resources such as Projects, Inventories, and Credentials. For a user to execute a job using a Job Template, they only need to be assigned the **Execute** role on the Job Template and do not need to be assigned **Use** roles to any of these associated Ansible Tower resources.

Read

The **Read** role grants users read-only access to view the properties of a Job Template. It also grants access to view other information related to the Job Template, such as the list of jobs executed using the Job Template, as well as its associated permissions and notifications.

MANAGING JOB TEMPLATE ACCESS

When a Job Template is first created, it is only accessible by the user that created it or users with either an **Admin** or **Auditor** role in the Organization that the Project was created in. Additional access must be specifically configured if desired.

The assignment of the previously discussed Job Template roles to users or teams dictates who has permissions to a Job Template. These permissions cannot be assigned at the time of a Job Template's creation. They are administered after a Job Template has been created by editing the Job Template.

Roles are assigned through the PERMISSIONS section of the Job Template editor screen. The following procedure details the steps for granting permissions to a Job Template after its creation.

1. Log in as a user with **Admin** role on the Organization that the Job Template is associated with or as the user who created the Job Template.
2. Click **TEMPLATES** in the left quick navigation bar to display the list of templates.
3. Click the pencil icon for the Job Template to edit to enter the Job Template editor screen.
4. On the Job Template editor screen, click the **PERMISSIONS** button to enter the permissions editor.
5. Click the **+** button to add permissions.
6. In the user and team selection screen, click either **USERS** or **TEAMS** and then select the users or teams to be granted permissions.
7. Click **KEY** to display the list of Job Template roles and their definitions.
8. Click the **SELECT ROLES** drop-down menu and select the desired Job Template role for each user or team.
9. Click **SAVE** to finalize the changes to permissions.

The screenshot shows the 'PERMISSIONS' tab for the 'Demo Job Template'. At the top, there are tabs for DETAILS, PERMISSIONS, NOTIFICATIONS, COMPLETED JOBS, and SCHEDULES. Below the tabs is a search bar and a key button. The main area displays a table with columns for USER, ROLE, and TEAM ROLES. The users listed are admin (SYSTEM ADMINISTRATOR), simon (SYSTEM ADMINISTRATOR), and sylvia (SYSTEM AUDITOR). A green '+' button is in the top right corner of the table area.

Figure 13.12: Assigning a Job Template role to a user

LAUNCHING JOBS

Once a Job Template is created, it can be used to launch a job with the following procedure.

1. Log in as a user that possesses the **Execute** role on the desired Job Template.
2. Click **Templates** in the left quick navigation bar to see the list of templates.
3. Locate the desired Job Template in the list of templates and click the rocket icon under the **ACTIONS** column to launch the job.
4. If any of the Job Template parameters have the **Prompt on launch** option enabled, then you are prompted for input prior to the job execution. Enter the desired input for each parameter prompted for and the click **LAUNCH** to launch the job.

The screenshot shows the 'TEMPLATES' page with the 'Demo Job Template' selected. The template details include ACTIVITY (Ansible Playbook), INVENTORY (Demo Inventory), PROJECT (Demo Project), and CREDENTIALS (Demo Credential). The last modified date is 10/31/2018 10:59:32 AM by admin, and the last run date is 10/31/2018 10:59:02 AM. On the right side, there is a 'Start a job using this template' button with a tooltip. Below the details, there are icons for launching, cloning, and deleting the template.

Figure 13.13: Launching a Job

EVALUATING THE RESULTS OF A JOB

Once a job run has been launched from a Job Template in the Ansible Tower web interface, the user is automatically redirected to the job's detail page. Users can also navigate to this same page by clicking **Jobs** in the left quick navigation bar to see the list of executed jobs and then clicking the link for the job of interest.

The job detail page is divided into two panes. The **DETAILS** pane displays the details of the job's parameters while the job output pane displays the output of the playbook executed by the job.

While the output in the job output pane resembles that which would have been generated by the execution of the playbook on the command line using the **ansible-playbook** command, it also offers several additional features. Across the top of the job output pane is a summary detailing the number of plays and tasks which were executed, the count of hosts which the job was executed against, and also the time it took for the job to execute. Additionally, controls are provided for maximizing this pane to full screen size, as well as for downloading the output of the job execution.

Along the left side of the output section, the + and - controls can be used to expand or collapse the output for each task in the playbook. Along the right side of the output section are controls for scrolling through the output as well as for jumping to the beginning and end of the output.

Job Details:

- STATUS: Successful
- STARTED: 10/31/2018 10:58:50 AM
- FINISHED: 10/31/2018 10:59:02 AM
- JOB TEMPLATE: Demo Job Template
- JOB TYPE: Run
- LAUNCHED BY: admin
- INVENTORY: Demo Inventory
- PROJECT: Demo Project
- PLAYBOOK: hello_world.yml
- CREDENTIAL: Demo Credential
- INSTANCE GROUP: tower
- EXTRA VARIABLES: YAML, JSON, EXPAND

Demo Job Template

```

PLAY [Hello World Sample] *****
ok: [localhost]
ok: [localhost] => {
    "msg": "Hello World!"
}
PLAY RECAP *****
localhost          : ok=2    changed=0    unreachable=0    failed=0

```

Figure 13.14: Job run results



REFERENCES

Ansible Tower User Guide

<http://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

► GUIDED EXERCISE

CREATING JOB TEMPLATES AND LAUNCHING JOBS

In this exercise, you will create a Job Template, assign a role to a team so that team members can use that Job Template, and use that Job Template to launch a job.

OUTCOMES

You should be able to create a Job Template and launch a Job from the Ansible Tower web interface.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run **`lab provision-job setup`**. This setup script ensures that the `workstation` and `tower` virtual machines are started.

```
[student@workstation ~]$ lab provision-job setup
```

- ▶ 1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
- ▶ 2. Create a new Job Template called `DEV webservers setup`.
 - 2.1. In the left navigation bar, click the `Templates` quick navigation link.
 - 2.2. Click the `+` button to add a new Job Template.
 - 2.3. From the drop-down menu, select `Job Template`.
 - 2.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	<code>DEV webservers setup</code>
DESCRIPTION	Setup apache on DEV webservers
JOB TYPE	Run
INVENTORY	Dev
PROJECT	My Webservers DEV
PLAYBOOK	<code>apache-setup.yml</code>
CREDENTIAL	Developers

- 2.5. Leave the other fields untouched and click `SAVE` to create the new Job Template.

- ▶ 3. Give the Developers team **Admin** role on the Job Template, `DEV webservers setup`.
 - 3.1. Click the PERMISSIONS button to manage the Job Template's permissions.
 - 3.2. Click the + button on the right to add permissions.
 - 3.3. Click TEAMS to display the list of available teams.
 - 3.4. In the first section, check the box next to Developers team. This causes the team to display in the second section underneath the first one.
 - 3.5. In the second section below, select the **Admin** role from the drop-down menu.
 - 3.6. Click SAVE to make the role assignment. This redirects you to the list of permissions for the Job Template, `DEV webservers setup`, which now shows that all members of the Developers team are assigned the **Admin** role on the Job Template.
- ▶ 4. Launch a Job using the Job Template, `DEV webservers setup`, as a member of the Developers team.
 - 4.1. Click the Log Out icon in the top right corner to log out and log back in as `daniel` with a password of `redhat123`.
 - 4.2. In the left navigation bar, click the Templates quick navigation link.
 - 4.3. On the same line as the Job Template, `DEV webservers setup`, click the rocket icon on the right to launch the Job. This redirects you to a detailed status page of the running job.
 - 4.4. Observe the live output of the running job for a minute.
 - 4.5. Verify that the STATUS of the Job in the DETAILS section displays **Successful**.
- ▶ 5. Verify that the web servers are up and running on `servera.lab.example.com` and `serverb.lab.example.com`.
 - 5.1. Open a web browser and go to `http://servera.lab.example.com`. You should see the following output:

```
This is a test message RedHat 7.6
Current Host: servera
Server list:
serverb.lab.example.com
servera.lab.example.com
```

- 5.2. Open a web browser and go to `http://serverb.lab.example.com`. You should see the following output:

```
This is a test message RedHat 7.6
Current Host: serverb
Server list:
serverb.lab.example.com
servera.lab.example.com
```

- ▶ 6. The setup script for this exercise created a directory that contains files which need to be pushed to the `my_webservers_DEV` git repository. On workstation, the **git-repos**

directory created in the first Guided Exercise in this chapter should exist. Pull the latest changes from the `my_webservers_DEV` git repository.

- 6.1. Use `git pull` to pull the latest changes from the remote repository. Change directory to the `/home/student/git-repos/my_webservers_DEV`.

```
[student@workstation ~]$ cd ~/git-repos/my_webservers_DEV
```

- 6.2. Pull the latest changes using `git pull`.

```
[student@workstation my_webservers_DEV]$ git pull  
Already up-to-date.
```

7. Copy the content of the `/home/student/D0409/labs/provision-job/` to the `/home/student/git-repos/my_webservers_DEV` directory. Push the new files to the master branch of your Git repository.

- 7.1. Use `cp -R` to copy the content of the `/home/student/D0409/labs/provision-job/` directory to the current working directory.

```
[student@workstation my_webservers_DEV]$ cp -R -v ~/D0409/labs/provision-job/* .  
'/home/student/D0409/labs/provision-job/ansible-vsftpd.yml' -> './ansible-  
vsftpd.yml'  
'/home/student/D0409/labs/provision-job/ftpclient.yml' -> './ftpclient.yml'  
'/home/student/D0409/labs/provision-job/site.yml' -> './site.yml'  
'/home/student/D0409/labs/provision-job/templates/vsftpd.conf.j2' -> './templates/  
vsftpd.conf.j2'  
'/home/student/D0409/labs/provision-job/vars' -> './vars'  
'/home/student/D0409/labs/provision-job/vars/defaults-template.yml' -> './vars/  
defaults-template.yml'  
'/home/student/D0409/labs/provision-job/vars/vars.yml' -> './vars/vars.yml'
```

- 7.2. Add all the new files to the staging area using `git add --all`.

```
[student@workstation my_webservers_DEV]$ git add --all
```

- 7.3. Commit the changes using `git commit` with the comment **Adding playbooks**.

```
[student@workstation my_webservers_DEV]$ git commit -m "Adding playbooks"  
[master a76733a] Adding playbooks  
 6 files changed, 118 insertions(+)  
 create mode 100644 ansible-vsftpd.yml  
 create mode 100644 ftpclient.yml  
 create mode 100644 site.yml  
 create mode 100644 templates/vsftpd.conf.j2  
 create mode 100644 vars/defaults-template.yml  
 create mode 100644 vars/vars.yml
```

- 7.4. Push the changes to the remote repository.

```
[student@workstation my_webservers_DEV]$ git push  
Counting objects: 12, done.
```

```
Delta compression using up to 2 threads.  
Compressing objects: 100% (10/10), done.  
Writing objects: 100% (10/10), 1.71 KiB | 0 bytes/s, done.  
Total 10 (delta 0), reused 0 (delta 0)  
To ssh://git@git.lab.example.com/home/git/my_webservers_DEV.git  
 042bc33..a76733a master -> master
```

► 8. Trigger an SCM update of the project My Webservers DEV.

- 8.1. Log in to the Ansible Tower web interface running on the tower system using the admin account and the redhat password.
- 8.2. In the left navigation bar, click the Projects quick navigation link.
- 8.3. In the line with the My Webservers DEV project, click the double arrow icon.
- 8.4. Observe the update and wait a couple of seconds. In the list of Projects, there is a status icon left of the My Webservers DEV Project. This icon is white at the start, red with an exclamation mark (!) when it fails, and green when it succeeds.

► 9. Create a new Job Template called DEV ftpservers setup.

- 9.1. In the left navigation bar, click the Templates quick navigation link.
- 9.2. Click the + button to add a new Job Template.
- 9.3. From the drop-down menu, select Job Template.
- 9.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	DEV ftpservers setup
DESCRIPTION	Setup FTP on DEV servers
JOB TYPE	Run
INVENTORY	Dev
PROJECT	My Webservers DEV
PLAYBOOK	site.yml
CREDENTIAL	Developers



NOTE

Notice that the My Webservers DEV project offers you additional playbooks to choose from, because you have pushed additional Ansible playbooks and templates to the same Git repository that serves as the remote source for the project.

- 9.5. Leave the other fields untouched and click SAVE to create the new Job Template.

- ▶ 10. Give the Developers team **Admin** role on the Job Template `DEV ftpservers` setup.
- 10.1. Click the PERMISSIONS button to manage the Job Template's permissions.
 - 10.2. Click the + button on the right to add permissions.
 - 10.3. Click TEAMS to display the list of available teams.
 - 10.4. In the first section, check the box next to Developers team. This causes the team to display in the second section underneath the first one.
 - 10.5. In the second section below, select the **Admin** role from the drop-down menu.
 - 10.6. Click SAVE to make the role assignment. This redirects you to the list of permissions for the Job Template `DEV ftpservers` setup, which now shows that all members of the Developers team are assigned the **Admin** role on the Job Template.
- ▶ 11. Launch a Job using the Job Template `DEV ftpservers` setup, as a member of the Developers team.
- 11.1. Click the Log Out icon in the top right corner to logout and log back in as `daniel` with a password of `redhat123`.
 - 11.2. In the left navigation bar, click the Templates quick navigation link.
 - 11.3. On the same line as the Job Template `DEV ftpservers` setup, click the rocket icon on the right to launch the Job. This redirects you to a detailed status page of the running job.
 - 11.4. Observe the live output of the running job for a minute.
 - 11.5. Verify that the STATUS of the Job in the DETAILS section displays **Successful**.
- ▶ 12. Verify that the FTP servers are up and running on `servera.lab.example.com` and `serverb.lab.example.com`.
- 12.1. Open a web browser and go to `ftp://servera.lab.example.com`. You should see the following output:

```
Index of ftp://servera.lab.example.com/
```

- 12.2. Open a web browser and go to `ftp://serverb.lab.example.com`. You should see the following output:

```
Index of ftp://serverb.lab.example.com/
```

- ▶ 13. Click the Log Out icon to log out of the Ansible Tower web interface.

This concludes the guided exercise.

► LAB

MANAGING PROJECTS AND LAUNCHING ANSIBLE JOBS

PERFORMANCE CHECKLIST

In this exercise, you will create a new project and job template, and assign an appropriate role for a team to be able to launch a job.

OUTCOMES

You should be able to manage Projects and Job Templates in order for a team to be able to launch a Job.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the `student` user on `workstation` and run `lab provision-review setup`. This setup script creates a new Git repository needed for the exercise.

```
[student@workstation ~]$ lab provision-review setup
```

1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
2. Create a new Project called `My Webservers TEST` with the following information:

FIELD	VALUE
NAME	My Webservers TEST
DESCRIPTION	Test Webservers Project
ORGANIZATION	Default
SCM TYPE	Git
SCM URL	<code>ssh://git.lab.example.com/home/git/my_webservers_TEST.git</code>
SCM CREDENTIAL	student-git

3. Give the `Developers` team both the `Use` and the `Update` roles on the Project, `My Webservers TEST`.
4. Create a new Job Template called `TEST webservers setup` with the following information:

FIELD	VALUE
NAME	TEST webservers setup
DESCRIPTION	Setup apache on TEST webservers

FIELD	VALUE
JOB TYPE	Run
INVENTORY	Test
PROJECT	My Webservers TEST
PLAYBOOK	apache-setup.yml
CREDENTIAL	Operations

5. Give the Developers team the **Execute** role on the Job Template, TEST webservers setup.
6. A new Git repository has been created under **git.lab.example.com:my_webservers_TEST.git**. Using the git remote user, clone the repository called my_webservers_TEST into the **git-repos** directory. Add the following two lines to the **index.html.j2** template and commit your changes to the remote repository.

```
...output omitted...
Current Memory: {{ ansible_memtotal_mb }} <br>
Current Free Memory: {{ ansible_memfree_mb }} <br>
Current Host: {{ ansible_hostname }} <br>
Server list: <br>
{% for host in groups['all'] %}
{{ host }} <br>
{% endfor %}
```

7. Update the My Webservers TEST project and launch a Job using the Job Template, TEST webservers setup, as the user, david, who is a member of the Developers team.
8. Log out of the Tower web interface and then verify that the web servers are up and running on serverc.lab.example.com and serverd.lab.example.com.
9. Run the command **lab provision-review grade** on workstation to grade your exercise.

This concludes the lab.

► SOLUTION

MANAGING PROJECTS AND LAUNCHING ANSIBLE JOBS

PERFORMANCE CHECKLIST

In this exercise, you will create a new project and job template, and assign an appropriate role for a team to be able to launch a job.

OUTCOMES

You should be able to manage Projects and Job Templates in order for a team to be able to launch a Job.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the `student` user on `workstation` and run `lab provision-review setup`. This setup script creates a new Git repository needed for the exercise.

```
[student@workstation ~]$ lab provision-review setup
```

1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
2. Create a new Project called `My Webservers TEST` with the following information:

FIELD	VALUE
NAME	My Webservers TEST
DESCRIPTION	Test Webservers Project
ORGANIZATION	Default
SCM TYPE	Git
SCM URL	<code>ssh://git.lab.example.com/home/git/my_webservers_TEST.git</code>
SCM CREDENTIAL	student-git

- 2.1. Click `Projects` in the left navigation bar.
- 2.2. Click the `+` button to add a new Project.
- 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	My Webservers TEST

FIELD	VALUE
DESCRIPTION	Test Webservers Project
ORGANIZATION	Default
SCM TYPE	Git
SCM URL	ssh://git.lab.example.com/home/git/my_webservers_TEST.git
SCM CREDENTIAL	student-git

- 2.4. Click SAVE to create the new Project.
- 2.5. Verify that the status icon of the SCM update job for the Project, My Webservers TEST, becomes green before proceeding.
3. Give the Developers team both the **Use** and the **Update** roles on the Project, My Webservers TEST.
- 3.1. Click Projects in the left navigation bar.
 - 3.2. On the same line as the Project, My Webservers TEST, click the pencil icon on the right to edit the Project.
 - 3.3. On the next page, click PERMISSIONS to manage the Project's permissions.
 - 3.4. Click the + button on the right to add permissions.
 - 3.5. Click TEAMS to display the list of available teams.
 - 3.6. In the first section, check the box next to the Developers team. This causes the team to display in the second section underneath the first one.
 - 3.7. In the second section below, select the **Use** role from the drop-down menu. Repeat that by clicking in the second section in the empty field after the **Use** role and selecting the **Update** role from the drop-down menu.
 - 3.8. Click SAVE to make the role assignment.
4. Create a new Job Template called TEST webservers setup with the following information:

FIELD	VALUE
NAME	TEST webservers setup
DESCRIPTION	Setup apache on TEST webservers
JOB TYPE	Run
INVENTORY	Test
PROJECT	My Webservers TEST
PLAYBOOK	apache-setup.yml

FIELD	VALUE
CREDENTIAL	Operations

- 4.1. Click the **Templates** in the left navigation bar.
- 4.2. Click the **+** button to add a new Job Template.
- 4.3. From the drop-down menu, select **Job Template**.
- 4.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	TEST webservers setup
DESCRIPTION	Setup apache on TEST webservers
JOB TYPE	Run
INVENTORY	Test
PROJECT	My Webservers TEST
PLAYBOOK	apache-setup.yml
CREDENTIAL	Operations

- 4.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.
5. Give the Developers team the **Execute** role on the Job Template, **TEST webservers setup**.
 - 5.1. Click **PERMISSIONS** to manage the Job Template's permissions.
 - 5.2. Click the **+** button on the right to add permissions.
 - 5.3. Click **TEAMS** to display the list of available teams.
 - 5.4. In the first section, click to select the box next to the Developers team. This causes the team to display in the second section underneath the first one.
 - 5.5. In the second section below, select the **Execute** role from the drop-down menu.
 - 5.6. Click **SAVE** to make the role assignment. This redirects you to the list of permissions for the Job Template **TEST webservers setup**, which now shows that all members of the Developers team are assigned the **Execute** role on the Job Template.
6. A new Git repository has been created under **git.lab.example.com:my_webservers_TEST.git**. Using the **git** remote user, clone the repository called **my_webservers_TEST** into the **git-repos** directory. Add the following two lines to the **index.html.j2** template and commit your changes to the remote repository.

```
...output omitted...
Current Memory: {{ ansible_memtotal_mb }} <br>
Current Free Memory: {{ ansible_memfree_mb }} <br>
```

```
Current Host: {{ ansible_hostname }} <br>
Server list: <br>
{% for host in groups['all'] %}
{{ host }} <br>
{% endfor %}
```

- 6.1. Change directory to the **git-repos** directory, where you store your Git repositories.

```
[student@workstation ~]$ cd git-repos
```

- 6.2. Clone the **my_webservers_TEST** repository using **git clone**. This creates a directory called **my_webservers_TEST** in your current directory. This new directory contains a playbook intended to setup an Apache web server.

```
[student@workstation git-repos]$ git clone \
> git@git.lab.example.com:my_webservers_TEST.git
Cloning into 'my_webservers_TEST'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
```

- 6.3. Change directory to the new directory. This is the root directory of the Git repository.

```
[student@workstation git-repos]$ cd my_webservers_TEST
```

- 6.4. Using your favorite editor, open the **templates/index.html.j2** template file for editing.

```
[student@workstation my_webservers_TEST]$ vim templates/index.html.j2
```

- 6.5. Add two additional lines that make use of Ansible facts. The **index.html.j2** file should look like this after the modification:

```
{{ apache_test_message }} {{ ansible_distribution }}
{{ ansible_distribution_version }} <br>
Current Memory: {{ ansible_memtotal_mb }} <br>
Current Free Memory: {{ ansible_memfree_mb }} <br>
Current Host: {{ ansible_hostname }} <br>
Server list: <br>
{% for host in groups['all'] %}
{{ host }} <br>
```

```
{% endfor %}
```

- 6.6. Save the changes made to the file and then exit from the editor.
- 6.7. Use **git add** to add the template to the staging area.

```
[student@workstation my_webservers_TEST]$ git add templates/index.html.j2
```

- 6.8. Commit your changes and check the status of your modifications. Use **git commit** with the comment message "**Added Ansible facts**".

```
[student@workstation my_webservers_TEST]$ git commit -m "Added Ansible facts"
[master 918ceb7] Added Ansible facts
 1 file changed, 22 insertion(+)
```

- 6.9. Push the changes to the remote repository using **git push**.

```
[student@workstation my_webservers_TEST]$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 472 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To git@git.lab.example.com:my_webservers_TEST.git
 2222420..6d777ac master -> master
```

7. Update the My Webservers TEST project and launch a Job using the Job Template, TEST webservers setup, as the user, **david**, who is a member of the Developers team.
 - 7.1. Click the Log Out icon in the top right corner to log out. Log back in as **david** with a password of **redhat123**.
 - 7.2. Click the **Projects** in the left navigation bar.
 - 7.3. On the same line as the My Webservers TEST project, click the double arrow icon. Wait for the update process to finish.
 - 7.4. Click the **Templates** in the left navigation bar.
 - 7.5. On the same line as the Job Template, TEST webservers setup, click the rocket icon on the right to launch the Job. This redirects you to a detailed status page of the running job.
 - 7.6. Observe the live output of the running job for a minute.
 - 7.7. Verify that the STATUS of the job in the DETAILS section displays **Successful**.
8. Log out of the Tower web interface and then verify that the web servers are up and running on **serverc.lab.example.com** and **serverd.lab.example.com**.
 - 8.1. Click the Log Out icon in the top right corner to logout of the Tower web interface.
 - 8.2. Open a web browser and go to **http://serverc.lab.example.com**. You should see the following output:

```
This is a test message RedHat 7.6
```

```
Current Memory: 991
Current Free Memory: 530
Current Host: serverc
Server list:
serverd.lab.example.com
serverc.lab.example.com
```

- 8.3. Open a web browser and go to **http://serverd.lab.example.com**. You should see the following output:

```
This is a test message RedHat 7.6
Current Memory: 991
Current Free Memory: 529
Current Host: serverd
Server list:
serverd.lab.example.com
serverc.lab.example.com
```

- 8.4. Click the Log Out icon to exit the Tower web interface.
9. Run the command **lab provision-review grade** on workstation to grade your exercise.

SUMMARY

In this chapter, you learned:

- Git is a distributed version control system. Users clone a Git project from a remote repository to a local repository, make changes to the local repository. When ready, changes are pushed to the remote repository to be shared with other users.
- An Ansible Tower Project contains one or more playbooks used to launch jobs.
- A project may get its materials from a source control repository, such as Git. The project may need to use a Source Control Credential (also called an SCM Credential) configured in Ansible Tower in order to authenticate to the source control repository.
- Job Templates are used to launch jobs that run Ansible Playbooks.
- A job template associates a playbook from a project, an inventory of hosts, and any credentials needed for authentication to the managed hosts in the inventory or to decrypt files protected with Ansible Vault.

CHAPTER 14

CONSTRUCTING ADVANCED JOB WORKFLOWS

GOAL

Use additional features of Job Templates to improve performance, simplify customization of Jobs, launch multiple Jobs, schedule automatically recurring Jobs, and provide notification of Job results.

OBJECTIVES

- Speed up Job execution by using and managing Fact Caching.
- Create a Job Template Survey to help users more easily launch a Job with custom variable settings.
- Create a Workflow Job Template and launch multiple Ansible jobs as a single workflow.
- Schedule automatic Job execution and configure notification of Job completion.
- Control Ansible Tower by accessing its API from Ansible Playbooks.

SECTIONS

- Improving Performance with Fact Caching (and Guided Exercise)
- Creating Job Template Surveys to Set Variables for Jobs (and Guided Exercise)
- Creating Workflow Job Templates and Launching Workflow Jobs (and Guided Exercise)
- Scheduling Jobs and Configuring Notifications (and Guided Exercise)
- Launching Jobs with the Ansible Tower API (and Guided Exercise)

LAB

- Constructing Advanced Job Workflows

IMPROVING PERFORMANCE WITH FACT CACHING

OBJECTIVES

After completing this section, students should be able to speed up job execution by using and managing fact caching.

FACT CACHING

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.

Normally, every play automatically runs the `setup` module before the first task, in order to gather facts from each managed host matching the play's host pattern. This ensures that the play has current facts, but also has some negative consequences.

Running the `setup` module to collect facts for every play has obvious performance consequences, especially on a large inventory of managed hosts. If you are not using any facts in the play, one way you could speed up execution would be to turn off automatic fact gathering. You can do this by setting `gather_facts: no` in the play. Note that you might not be able to do this if you are actually using facts in the play.

Playbooks can also reference another host's facts by using the "magic" variable `hostvars`. For example, a task running on the managed host `servera` can access the value of the fact `ansible_facts['default_ipv4']['address']` for `serverb` by referencing the variable `hostvars['serverb']['ansible_facts']['default_ipv4']['address']`. However, this only works if facts have already been gathered from `serverb` by this play or by an earlier play in the same playbook.

You can use *fact caching* to address both of these problems. One playbook can collect facts for all hosts in the inventory and cache those facts so that subsequent playbooks may use them without fact gathering or manually running the `setup` module.

Enabling Fact Caching in Ansible Tower

Red Hat Ansible Tower 3.2 and later include integrated support for fact caching and a database for the fact cache. You need to manage timeouts for the fact cache at a global level. Fact caching control is determined by the job template.

There is a global setting in Ansible Tower that controls when facts expire, per host. On the left-side navigation bar of the web UI, select `Settings` to display the `Configure Tower` pane, and then click `JOBS`. The setting `Per-Host Ansible Fact Cache Timeout` controls how long Ansible facts in the cache are considered to be valid after they have been collected. This is measured in seconds.

The default value is set to `0`, meaning that the information stored in the cache is always valid. However, if you do not periodically gather facts to update the cache, you risk facts becoming outdated due to changes on the managed hosts.

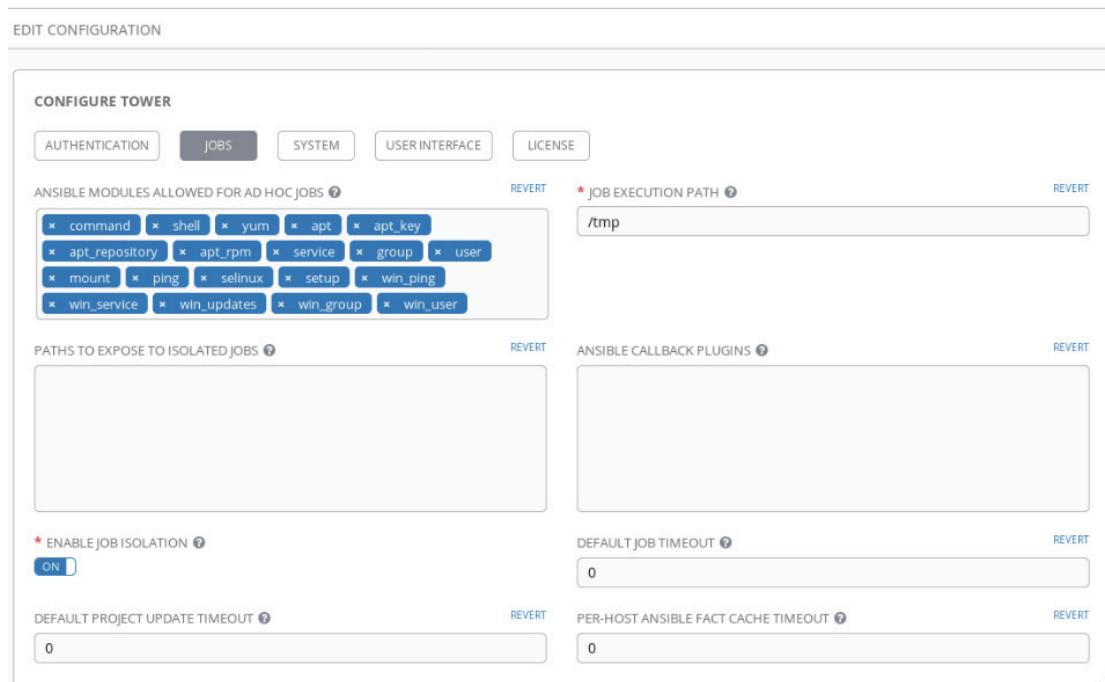


Figure 14.1: Setting Ansible Fact Cache Timeout

To optimize fact caching, set `gather_facts: no` to disable automatic fact gathering in the play. You will also need to enable Use Fact Cache for any Ansible Tower job templates that use playbooks containing those plays. The plays then depend on the information in the fact cache to use facts.

You will also need to periodically run a play that populates the fact cache to keep the cached facts current. The Ansible Tower job template for this playbook also needs to enable Use Fact Cache. One good way to do this in Ansible Tower is to set up a playbook that gathers facts as a scheduled job (discussed elsewhere in this course). That job could run a normal playbook that gathers facts, or you could set up a minimal playbook to gather facts, such as the following example:

```
- name: Refresh fact cache
  hosts: all
  gather_facts: yes
```

Note that because there is no `tasks` or `roles` section, the only thing this playbook does is gather facts.



NOTE

You do not need to gather facts for all your hosts at the same time. It might make sense to gather facts for smaller sets of your hosts to spread the load. The important thing is to make sure you gather facts for all of your hosts before they expire or become stale.

The following procedure shows how to enable fact caching in the Ansible Tower interface:

1. Click **Templates** in the left navigation bar.
2. Choose the appropriate job template and click its name to edit the settings.
3. In the **OPTIONS** section of the page, select the check box next to **Use Fact Cache**.

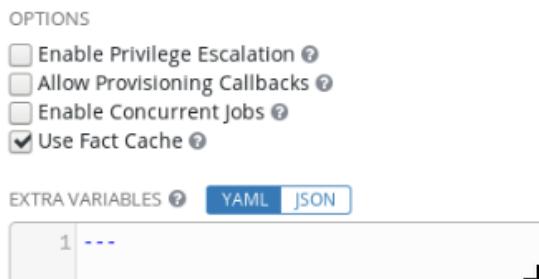


Figure 14.2: Enabling fact caching

4. Click SAVE to save the modified job template configuration.

Now whenever you run a new job based on a template that has the Use Fact Cache option enabled, the job will use the fact cache. If the Ansible Playbook also has the `gather_facts` variable set to `yes`, the job will gather facts, retrieve them, and store them in the fact cache.



NOTE

When a job is launched, Ansible Tower injects all `ansible_facts` for each of the managed hosts from the running job into memcache. After finishing the job, Ansible Tower retrieves all the records for a particular host from memcache, and then saves each fact that has an update time later than the cached copy in the fact cache database.



REFERENCES

Ansible User Guide: Caching Facts

https://docs.ansible.com/ansible/2.7/user_guide/playbooks_variables.html#caching-facts

Ansible Tower User Guide: Fact Caching

https://docs.ansible.com/ansible-tower/latest/html/userguide/job_templates.html#fact-caching

► GUIDED EXERCISE

IMPROVING PERFORMANCE WITH FACT CACHING

In this exercise, you will use fact caching to speed up job execution and explore how to manage the fact cache.

OUTCOMES

You should be able to use fact caching in job templates.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed, configured and running on the tower system.

Log in to workstation as student and run **lab project-facts setup**.

```
[student@workstation ~]$ lab project-facts setup
```

- ▶ 1. Log in to the Ansible Tower web UI running on the `tower` system as `daniel` using `redhat123` as the password.
- ▶ 2. Change the `My Webservers DEV` project so that it automatically triggers an SCM update.
 - 2.1. In the left navigation bar, click **Projects**.
 - 2.2. Click `My Webservers DEV` to edit the project settings.
 - 2.3. Under **SCM UPDATE OPTIONS**, select **Update Revision on Launch** and then click **SAVE**.
- ▶ 3. Launch a job using the `DEV webservers setup` Job Template. (This template was set up for you by the setup command.)

The job will fail. Watch the job's output screen and try to identify why it failed.

 - 3.1. In the left navigation bar, click **Templates**.
 - 3.2. Click the rocket icon for the `DEV webservers setup` Job Template to launch a job. This will redirect you to a detailed status page of the running job.
 - 3.3. Briefly observe the output of the running job.
 - 3.4. When the job completes, verify that the **STATUS** of the job in the **DETAILS** section displays **Failed**.
 - 3.5. Look at the output of the completed job in the right pane. Scroll down to the section that shows the output of the task that failed.

The job failed because the playbook it ran uses a variable that gets its value from the `ansible_distribution` Ansible fact. The playbook used for the job does not gather facts, and the Job Template does not use fact caching, so the task failed.

- 4. Fix the issue by setting the value of the `gather_facts` variable to `yes` in the `apache-setup.yml` Ansible Playbook.

- 4.1. On `workstation`, open a terminal and change to the `git-repos` directory, where you stored your Git repositories from the previous chapter.

```
[student@workstation ~]$ cd git-repos
```

- 4.2. Change to the `my_webservers_DEV` directory.

```
[student@workstation git-repos]$ cd my_webservers_DEV
```

- 4.3. Use the `git pull` command to pull the latest version from the Git server.

```
[student@workstation my_webservers_DEV]$ git pull  
...output omitted...  
Fast-forward  
 apache-setup.yml | 9 ++++++--  
 1 file changed, 5 insertions(+), 4 deletions(-)
```

- 4.4. Edit the `apache-setup.yml` playbook and change the `gather_facts:` variable value to `yes` to enable fact gathering.

1. Open the `apache-setup.yml` playbook for editing.

```
[student@workstation my_webservers_DEV]$ vi apache-setup.yml
```

2. Change the value of the `gather_facts` variable to `yes`.

```
---  
- hosts: all  
  name: Install the web server and start it  
  become: yes  
  gather_facts: yes  
  vars:  
  ...output omitted...
```

3. Save and close the file.

- 4.5. Use the `git add --all` command to add all files to the staging area.

```
[student@workstation my_webservers_DEV]$ git add --all
```

- 4.6. Use `git commit` to commit the changes; use the comment **Enabling facts gathering**.

```
[student@workstation my_webservers_DEV]$ git commit -m "Enabling facts gathering"  
[master 5e48392] Enabling facts gathering
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

- 4.7. Push the changes to the remote repository.

```
[student@workstation my_webservers_DEV]$ git push  
Counting objects: 5, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 316 bytes | 0 bytes/s, done.  
Total 3 (delta 2), reused 0 (delta 0)  
To ssh://git@git.lab.example.com/home/git/my_webservers_DEV.git  
 042bc33..5e48392 master -> master
```

- ▶ 5. Go back to the Ansible Tower web UI.

Edit the `DEV webservers setup` Job Template to enable the Use Fact Cache option. Launch a job using that edited Job Template.

Because the modified playbook has `gather_facts: yes` set in one of its plays, it will gather Ansible facts. Because the Job Template has Use Fact Cache set, the gathered facts will be stored in the fact cache for future use.

- 5.1. In the left navigation bar, click Templates.
- 5.2. Click the `DEV webservers setup` Job Template to edit the Template.
- 5.3. Select Use Fact Cache to enable the Fact Caching option.
- 5.4. Click SAVE.
- 5.5. Click the rocket icon for the `DEV webservers setup` Job Template. This redirects you to a detailed status page of the running job.
- 5.6. Briefly observe the output of the running job.
- 5.7. Verify that the STATUS of the job in the DETAILS section displays **Successful**.
This time the job succeeds and stores the Ansible facts for all hosts specified in the Dev inventory in the cache.

- 6. Set the `gather_facts` variable to `no` in the `apache_setup.yml` playbook. Launch another job using the `DEV webservers` setup Job Template.

Even though fact gathering is turned off, the job still succeeds because the fact used by the variable in the playbook can use the fact cache. This job should also run more quickly because it does not need to gather facts.

- 6.1. On `workstation`, open a terminal and make sure you are in the `~/git-repos/my_webservers_DEV` directory, where you stored your Git repositories from the previous chapter.

```
[student@workstation ~]$ cd git-repos
```

- 6.2. Change to the `my_webservers_DEV` directory.

```
[student@workstation git-repos]$ cd my_webservers_DEV
```

- 6.3. Edit the `apache-setup.yml` playbook and change the default behavior by adding the `gather_facts: no` variable to disable fact gathering.
 - 6.4. Open the `apache-setup.yml` playbook for editing.

```
[student@workstation my_webservers_DEV]$ vi apache-setup.yml
```

- 6.5. Add the `gather_facts` variable to the playbook and set its value to `no`.

```
---
- hosts: all
  name: Install the web server and start it
  become: yes
  gather_facts: no
  ...output omitted...
```

- 6.6. Save and close the file.

- 6.7. Use `git add --all` to add all files to the staging area.

```
[student@workstation my_webservers_DEV]$ git add --all
```

- 6.8. Use `git commit` to commit the changes; use the comment **Disabling facts gathering**.

```
[student@workstation my_webservers_DEV]$ git commit -m "Disabling facts gathering"
[master 5e48392] Disabling facts gathering
 1 file changed, 1 insertion(+)
```

- 6.9. Push the changes to the remote repository.

```
[student@workstation my_webservers_DEV]$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 316 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 2), reused 0 (delta 0)
To ssh://git@git.lab.example.com/home/git/my_webservers_DEV.git
  042bc33..5e48392 master -> master
```

▶ **7.** Go back to the Ansible Tower web UI.

As a member of the Developers team, launch a job using the `DEV_webservers_setup` Job Template. This new job, based on the modified playbook, will succeed because the Ansible facts have been cached and can be used to execute the playbook correctly.

- 7.1. In the left navigation bar, click **Templates**.
 - 7.2. On the same line as the `DEV_webservers_setup` Job Template, click the rocket icon on the right to launch the job. This will redirect you to a detailed status page of the running job.
 - 7.3. Briefly observe the live output of the running job.
 - 7.4. Verify that the **STATUS** of the job in the **DETAILS** section displays **Successful**.
 - 7.5. In the right pane, scroll down to the section that shows the output of the playbook. Notice that no facts were gathered and the task that previously failed succeeded this time. The cached facts were used to set the correct value for the variable that sets its own value from the Ansible facts.
- ▶ **8.** Verify that the Ansible facts have been stored in the Ansible Tower fact cache.
- 8.1. Click **Inventories** in the left navigation bar.
 - 8.2. Click **HOSTS** and then click the `servera.lab.example.com` link.
 - 8.3. Click **FACTS**. Scroll down and verify that all the Ansible facts from `servera.lab.example.com` server have been stored in the Ansible Tower database.
 - 8.4. When ready, log out from the Ansible Tower web UI.

This concludes the guided exercise.

CREATING JOB TEMPLATE SURVEYS TO SET VARIABLES FOR JOBS

OBJECTIVES

After completing this section, students should be able to create a job template survey to help users more easily launch a job with custom variable settings.

Figure 14.2: Creating Job Template Surveys to set variables for jobs

MANAGING VARIABLES

Ansible users are encouraged to write playbooks that can be reused in different situations or when deploying to systems that should have slightly different behaviors, configurations, or work in different environments. One easy way to deal with this is to use variables.

Variables can have values set in a number of ways by Ansible, and values can be overridden depending on how they were set. For example, a role can provide a default value for a variable that may in turn be overridden by values set for that variable by the inventory or by the playbook. However, in general it is best to set a value for a variable in exactly one place to help avoid issues with variable precedence.

When running playbooks using **ansible-playbook**, users have two ways to set values for variables interactively. Firstly, they can pass *extra variables* by using the **-e** or **--extra-vars** option to the command. Extra variables always take precedence. Alternatively, the playbook may have a **vars_prompt** section that can interactively prompt users for input when they run a playbook. The values set by **vars_prompt** variables have a lower precedence than extra variables and can be overridden by various things.

In Ansible Tower, this works a little differently. Extra variables can be set by the Job Template, users can be prompted for them when launching a Job Template, or they may be set automatically by rerunning a Job that was launched with extra variables defined. Playbooks with **vars_prompt** questions are not supported by Ansible Tower. The closest replacement for **vars_prompt** is the Surveys feature of Ansible Tower, which is discussed later in this section.



IMPORTANT

Ansible Tower does not support playbooks that use **vars_prompt** to interactively set variables.

DEFINING EXTRA VARIABLES

In Ansible Tower, Job Templates can be used to directly set extra variables in two ways:

- Extra variables can be set by entering them in YAML or JSON format in the **EXTRA VARIABLES** field of the Job Template.
- If **PROMPT ON LAUNCH** is selected for the **EXTRA VARIABLES** field, Ansible Tower users are prompted to interactively modify the list of extra variables used when they use the Job Template to launch a job.

These extra variables are exactly like the variables specified by the `-e` or `--extra-vars` options for `ansible-playbook`, and their values override any values set for those variables. The values set through extra variables always take precedence.

The following example demonstrates setting extra variables in the Job Template:

DEV ftpservers setup

DETAILS PERMISSIONS NOTIFICATIONS COMPLETED JOBS SCHEDULES ADD SURVEY

NAME DEV ftpservers setup DESCRIPTION Setup FTP on DEV servers JOB TYPE Run PROMPT ON LAUNCH

INVENTORY Dev PROJECT My Webservers DEV PLAYBOOK site.yml PROMPT ON LAUNCH

CREDENTIAL Developers CREDITS FOLDS PROMPT ON LAUNCH FORKS DEFAULT LIMIT PROMPT ON LAUNCH

VERBOSITY 0 (Normal) TAGS PROMPT ON LAUNCH SKIP TAGS PROMPT ON LAUNCH

LABELS INSTANCE GROUPS SHOW CHANGES OFF PROMPT ON LAUNCH

OPTIONS

- Enable Privilege Escalation
- Allow Provisioning Callbacks
- Enable Concurrent Jobs
- Use Fact Cache

EXTRA VARIABLES YAML JSON

```

1---
2msg_color: green
3welcome_msg: '{{ ansible_fqdn }} is up'

```

Figure 14.3: Adding extra variables to a Job Template

If PROMPT ON LAUNCH is selected for EXTRA VARIABLES, when a Job is launched using the Job Template a dialog box displays which allows Ansible Tower users to edit the extra variables for the job:

PROMPT OTHER PROMPTS PREVIEW

EXTRA VARIABLES YAML JSON

```

1---
2msg_color: green
3welcome_msg: '{{ ansible_fqdn }} is up'

```

CANCEL NEXT

Figure 14.4: Adjusting extra variables on job launch

If the resulting Job is later relaunched, the same extra variables are used again. The extra variables for a Job cannot be changed when relaunching it. Instead, launch the job from the original Job Template with different extra variables set.

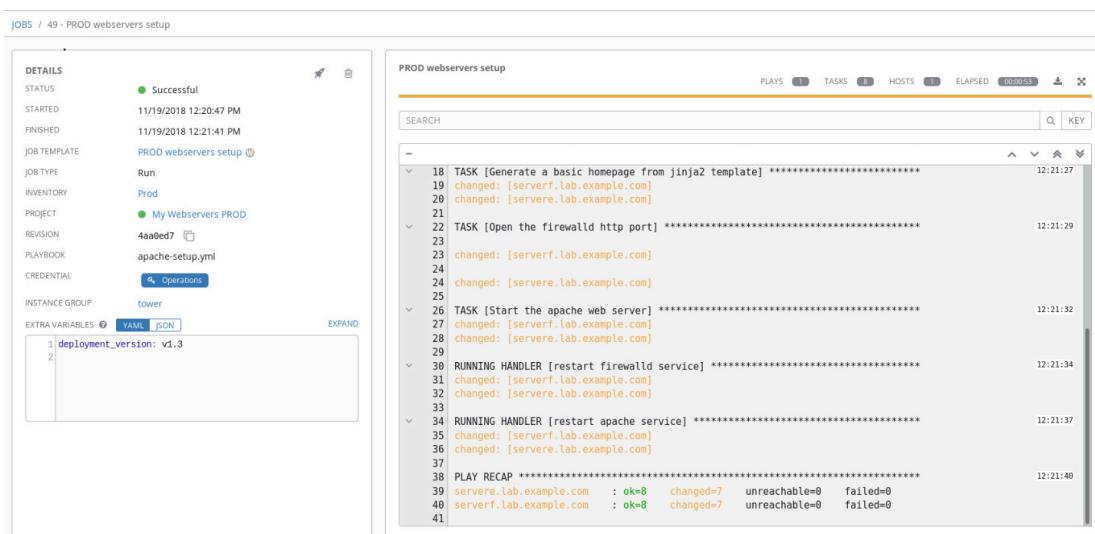


Figure 14.5: Relaunching a job that used extra variables

JOB TEMPLATE SURVEYS

Extra variables can be hard to use because the user launching the job needs to understand what variables are available and how they should be used with the Job Template's playbook. *Job Template Surveys* allow the Job Template to display a short form when used to launch a job to prompt users for information that is used to set values for extra variables.

Prompting for user input offers several advantages over other ways to set extra variables. Users do not need to have detailed knowledge about how extra variables work or that they are even being used. They also do not need to have knowledge of the names of the extra variables that are used by the playbook.

Because prompts can contain arbitrary text, they can be phrased in a manner that is user-friendly and easily understood by users who may not have detailed knowledge about Ansible.



IMPORTANT

Surveys set extra variables. In fact, the values set by Surveys for variables override the values set on variables of the same name in any other way. This includes the Job Template's EXTRA VARIABLES field or its PROMPT ON LAUNCH setting.

This is one way in which Surveys and **vars_prompt** are not direct replacements for each other. Variables set through **vars_prompt** have a lower precedence than extra variables and can be overridden in a number of ways. Values set by Surveys are extra variables and always win.

User-friendly Questions

Surveys allow users to be prompted with customized questions. This allows for more user-friendly prompts for user input of extra variable values than the PROMPT ON LAUNCH method.

Answer Type

Aside from offering a user-friendly prompt, Surveys can also define rules for, and perform validation of, user inputs. User responses to survey questions can be restricted to one of the following seven answer types:

TYPE	DESCRIPTION
Text	Single-line text
Textarea	Multiline text
Password	Data is treated as sensitive information.
Multiple Choice (single select)	A list of options from which only one option can be chosen as a response.
Multiple Choice (multiple select)	A list of options from which one or more options can be chosen as a response.
Integer	Integer number
Float	Floating-point decimal number

Answer Length

You can also define rules for the size of user responses to survey questions. For the following non-list answer types, a survey can define the minimum and maximum allowable character length for user responses: Text, Textarea, Password, Integer, and Float.

Default Answer

A default answer can be provided for a question. The question can also be marked as REQUIRED, which indicates that an answer must be provided for the question.

CREATING A JOB TEMPLATE SURVEY

During the creation of a Job Template, the addition of a Survey is not possible. A Survey can only be added to a Job Template after the Job Template has been created.

The following procedure illustrates how to add a Survey to an existing Job Template.

1. Click **Templates** in the left navigation bar to see the list of existing Job Templates.
2. Click the desired Job Template to edit the Job Template.
3. Click **ADD SURVEY** to enter the Survey creation interface.
4. Add a question to the Survey.
 1. In the **PROMPT** field, enter the question to display to the user.
 2. In the **ANSWER VARIABLE NAME** field, enter the name of the extra variable to assign the user's response to.
 3. From the **ANSWER TYPE** list, select the desired answer type for the user's response.
 4. If using a list answer type, define the list by entering one list item per line in the **MULTIPLE CHOICE OPTIONS** field.
 5. If using a non-list answer type, optionally specify the minimum and maximum character length for the user's response in the **MINIMUM LENGTH** and **MAXIMUM LENGTH** fields, respectively.
 6. If desired, optionally define a default value for the extra variable being surveyed in the **DEFAULT ANSWER** field. This value will be used if no user response are entered.

7. Select REQUIRED to specify that a response is required. Clear this field if a response is optional.
 8. Click +ADD to add the question to the Survey. A preview of the added question appears under the PREVIEW section of the interface.
5. Repeat the previous steps to add additional questions to the Survey. After you have added all your questions, scroll to the top of the screen. In the upper-left corner, next to SURVEY is a toggle button, which determines whether the Survey is enabled or disabled. By default, Surveys are enabled upon creation, so the button is set to **ON**. If the Survey should be disabled, set it to **OFF**.
6. Lastly, click SAVE to save the Survey.

The screenshot shows a configuration dialog for adding a survey prompt. At the top left, it says 'DEV ftsservers setup | SURVEY ON'. On the left, there's a form with fields: 'PROMPT' (containing 'Which version do you want to deploy?'), 'DESCRIPTION' (empty), 'ANSWER VARIABLE NAME' (containing 'version'), 'ANSWER TYPE' (set to 'Text'), 'MINIMUM LENGTH' (0), 'MAXIMUM LENGTH' (1024), 'DEFAULT ANSWER' (empty), and a checked 'REQUIRED' checkbox. At the bottom are 'CLEAR', '+ ADD', 'CANCEL', and 'SAVE' buttons. On the right, a 'PREVIEW' section shows the text 'PLEASE ADD A SURVEY PROMPT.'

Figure 14.6: Adding Survey to a job template

After you have added a Survey to a Job Template, users will be prompted for answers to the Survey's questions when they launch jobs with that Job Template. If the Job Template has extra variables or is configured to prompt the user to set extra variables, they will be set before the Survey is displayed to the user. Answers to the Survey override any extra variables that have already been set.

The screenshot shows a survey prompt dialog titled 'PROMPT'. It has tabs 'SURVEY' (selected) and 'PREVIEW'. Below is the question: '* WHAT VERSION ARE YOU DEPLOYING?'. A note says 'This version will be displayed at the bottom of the index page'. The answer field contains 'v1.0'. At the bottom are 'CANCEL' and 'NEXT' buttons.

Figure 14.7: Survey Prompt



NOTE

The Job Template Survey feature is only available with enterprise-level Red Hat Ansible Tower licenses.



REFERENCES

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide/index.html>

► GUIDED EXERCISE

CREATING JOB TEMPLATE SURVEYS TO SET VARIABLES FOR JOBS

In this exercise, you will add a Survey to an existing Job Template and launch a Job using that Survey.

OUTCOMES

You should be able to add a Survey to an existing Job Template and launch a Job using a Survey from the Ansible Tower web UI.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run `lab project-survey setup` command. This script ensures that the `workstation` and `tower` virtual machines are started.

```
[student@workstation ~]$ lab project-survey setup
```

- ▶ 1. Log in to the Ansible Tower web UI running on `tower` as `admin` using `redhat` as the password.
- ▶ 2. Add a Survey to the `DEV webservers setup` Job Template.
 - 2.1. Click `Templates` on the navigation bar.
 - 2.2. In the list of available templates, click `DEV webservers setup` to edit the Job Template.
 - 2.3. Click `ADD SURVEY` to add a Survey.
 - 2.4. On the next screen, fill in the details as follows:

FIELD	VALUE
PROMPT	What version are you deploying?
DESCRIPTION	This version number will be displayed at the bottom of the index page.
ANSWER VARIABLE NAME	<code>deployment_version</code>
ANSWER TYPE	Text
MINIMUM LENGTH	1
MAXIMUM LENGTH	40
DEFAULT ANSWER	v1.0

FIELD	VALUE
REQUIRED	Checked

- 2.5. Click +ADD to add that Survey Prompt to the Survey. This displays a preview of your Survey.

**IMPORTANT**

Before saving, make sure that the ON/OFF switch is set to **ON** at the top of the Survey editor window.

- 2.6. Click SAVE to add the Survey to the Job Template.

▶ **3.** Modify the **apache-setup** playbook to use the variable from the Survey.

- 3.1. Open a terminal on **workstation** and change to the **my_webservers_DEV** Git repository.

```
[student@workstation ~]$ cd git-repos/my_webservers_DEV
```

- 3.2. Edit the **index.html.j2** template.

```
[student@workstation my_webservers_DEV]$ vi templates/index.html.j2
```

- 3.3. Append the following line to the bottom of the file.

```
Deployment Version: {{ deployment_version }} <br>
```

- 3.4. Save the file.

- 3.5. Add, commit, and push the file to the remote repository.

```
[student@workstation my_webservers_DEV]$ git add --all
[student@workstation my_webservers_DEV]$ git commit -m "Display Deployment Version
on index page"
[student@workstation my_webservers_DEV]$ git push
[master 814b7c3] Display Deployment Version on index page
 1 file changed, 1 insertion(+)
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 429 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To git@git.lab.example.com:my_webservers_DEV.git
  d6e363f..814b7c3  master -> master
```

- ▶ 4. Update the local copy of the repository for the My Webservers DEV Project.
- 4.1. Click Projects on the navigation bar.
 - 4.2. On the same line as the My Webservers DEV Project, click the double arrow icon on the right to launch an SCM update of the Project and wait for the status icon to be steady green.
- ▶ 5. As a member of the Developers team, launch a Job using the updated DEV webservers setup Job Template.
- 5.1. Click the Log Out icon in the upper-right corner to log out, and then log back in as daniel using redhat123 as the password.
 - 5.2. Click Templates on the navigation bar.
 - 5.3. On the same line as the DEV webservers setup Job Template, click the rocket icon on the right to launch the Job. This opens the Survey you just created and asks for your input.
 - 5.4. You can leave **v1.0** in the text field and click NEXT followed by LAUNCH to launch the job. This redirects you to a detailed status page of the running job.
 - 5.5. Briefly observe the live output of the running Job.
 - 5.6. Verify that the STATUS of the job in the DETAILS section is **Successful**.
- ▶ 6. Verify that the web servers have been updated on servera.lab.example.com and serverb.lab.example.com.
- 6.1. Open a web browser and navigate to <http://servera.lab.example.com> and <http://serverb.lab.example.com> in separate tabs. You should see this line at the bottom of both pages:

```
...output omitted...
Deployment Version: v1.0
```

- ▶ 7. Click the Log Out icon to log out of the Ansible Tower web UI.

This concludes the guided exercise.

CREATING WORKFLOW JOB TEMPLATES AND LAUNCHING WORKFLOW JOBS

OBJECTIVES

After completing this section, students should be able to create a Workflow Job Template and launch multiple Ansible jobs as a single workflow.

Figure 14.7:

WORKFLOW JOB TEMPLATES

In a previous section, you learned how to use Job Templates to run single Ansible Playbooks as jobs. As an organization's use of Ansible grows, so does the number of Ansible Playbooks it has. Each playbook typically performs a set of tasks associated with a certain function.

Instead of writing one large playbook to automate a complex operation, you might want to run several playbooks in sequence. For example, to provision a server you might need to use the Networking team's playbook to allocate an IP address to the server and set up a DNS record, then use a separate playbook from the Operations team to install and configure the server's operating system. Finally, you would use a playbook from the Development team to deploy an application. In other words, there is a particular *workflow* that you need to follow for the process to succeed.

This could be managed in Ansible Tower by having users manually launch multiple jobs in sequence. But the jobs have to be executed in the correct order, as defined by your workflow, for everything to work correctly.

1. The Networking jobs have to be executed first.
2. The Operations jobs would follow only if the Networking jobs were successfully completed.
3. Likewise, the Application Development jobs would then follow only if the Networking and Operations jobs successfully completed.

Finally, if one of these playbooks fails, you might want to run other playbooks to recover.

To make this easier to manage, Red Hat Ansible Tower supports *Workflow Job Templates*. A Workflow Job Template connects multiple Job Templates into a workflow. When launched, the Workflow Job Template launches a job using the first Job Template, and depending on whether it succeeds or fails, determines which Job Template to launch next. This allows a sequence of jobs to be launched, and for recovery steps to be taken automatically if a job fails.

Workflow Job Templates can be started in many ways: manually, from the Ansible Tower web UI; as a scheduled job; by an external program using the Ansible Tower API.

Workflow Job Templates do not just run Job Templates in a serial fashion. Using the graphical workflow editor, Workflow Job Templates chain together multiple Job Templates and run different Job Templates depending on whether the previous one succeeded or failed.



NOTE

The Workflow Job Template feature is only available with Enterprise-level licenses.

CREATING WORKFLOW JOB TEMPLATES

You need to create a Workflow Job Template before a workflow can be defined and associated with it. They can be created with or without an Organization. Creating a Workflow Job Template within the context of an Organization requires that the user has the **admin** role for the Organization. The singleton **System Administrator** user type is required in order to create a Workflow Job Template that is not part of an Organization.

Figure 14.8: Creating a Workflow Job Template

Workflow Job Templates are created in a similar fashion to Job Templates.

1. Click **Templates** in the left navigation bar to access the Template management interface.
2. Click the **+** button and select **Workflow Template**.
3. Enter a unique name for the Workflow Job Template in the **NAME** field. Optionally enter any desired key-value pairs in the **EXTRA VARIABLES** field.
4. Click **SAVE** to create the Workflow Job Template. After a Workflow Job Template has been created, you can use the Workflow Visualizer to define an associated workflow.

Using the Workflow Visualizer

After you have created a Workflow Job Template, the **WORKFLOW VISUALIZER** becomes active in the Workflow Job Template editing screen. The Workflow Editor opens in a new window. The Workflow Visualizer is a graphical interface for defining the job templates to incorporate in a workflow as well as the decision tree structure, which should be used to chain the job templates together.

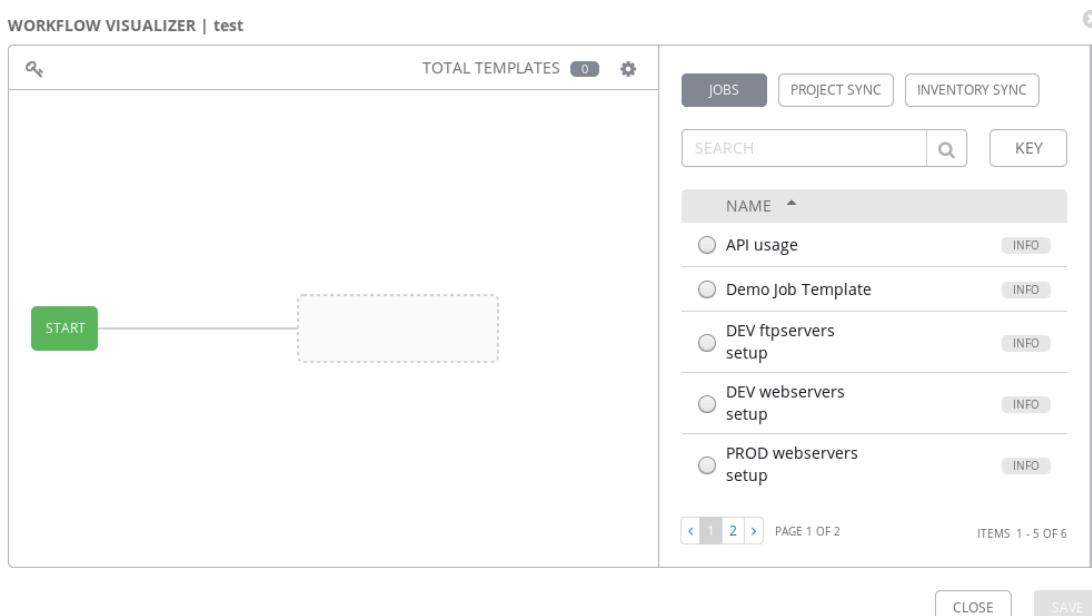


Figure 14.9: Starting a workflow in the Workflow Visualizer

When the Workflow Visualizer is launched, it contains a single START node representing the starting point for the execution of the workflow. Click START to initiate the workflow editing process; the Workflow Visualizer displays a list of Ansible Tower resources, which can be added as the first step of the workflow. You can select the desired resource type, the specific resource, and then click SELECT to add an Ansible Tower resource as the first node in the workflow.

In addition to Job Templates, jobs that synchronize Projects or Inventories can also be incorporated into a workflow. This is useful to ensure that Project and Inventory resources are updated prior to the use of Job Templates that depend on them. To make them easier to identify, Project Sync and Inventory Sync nodes are indicated by a P or an I in the lower-left of the node, respectively. This notation is explained by the key at the top of the Workflow Editor screen. Job Template nodes are not marked with any special notation because they are the main node type in a workflow.

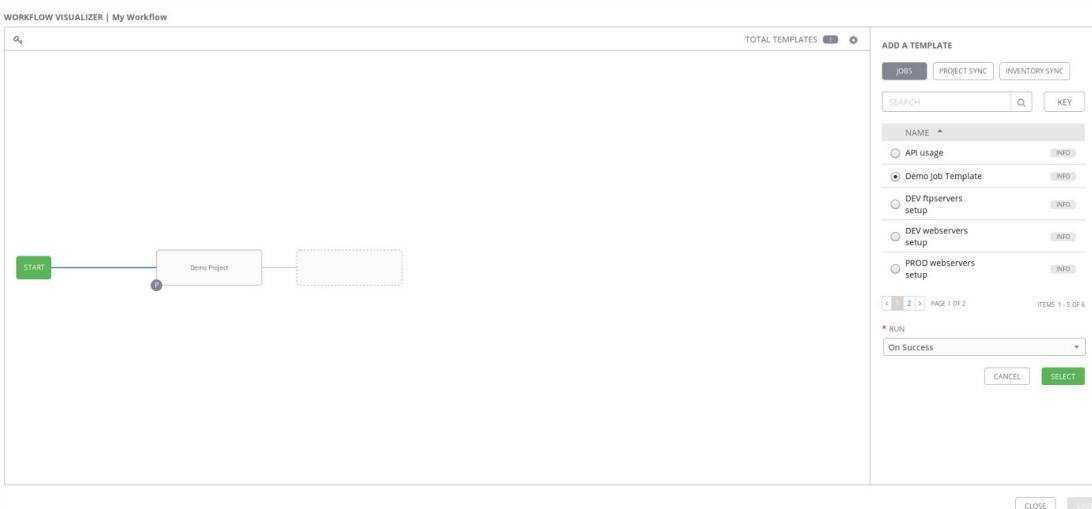


Figure 14.10: Adding an 'On Success' node to the workflow

After a resource has been added as the first workflow node, hovering over it causes two buttons to appear. The red - button deletes the node and the green + button adds a subsequent node. When adding subsequent nodes, the RUN prompt appears in the resource selection panel prompting for

additional input when you select a resource. The following three choices are offered by this prompt to designate the relationship between the new node and the preceding node.

RUN	NODE RELATIONSHIP
On Success	The node resource is executed upon successful completion of the actions associated with the previous node.
On Failure	The node resource is executed upon failure of the actions associated with the previous node.
Always	The node resource is executed regardless of the outcome of the actions associated with the previous node.

A node can have more than one child node. For example, a child node can be added with a parent node association type of On Success and another child node can be added with an association type of On Failure. For example, you can add a child node to a parent node using an On Success association type. You can add a second child node to the same parent using an On Failure association type. This creates a branch in the workflow structure such that one course of action is taken upon the success of an action, and a different course is taken upon failure.

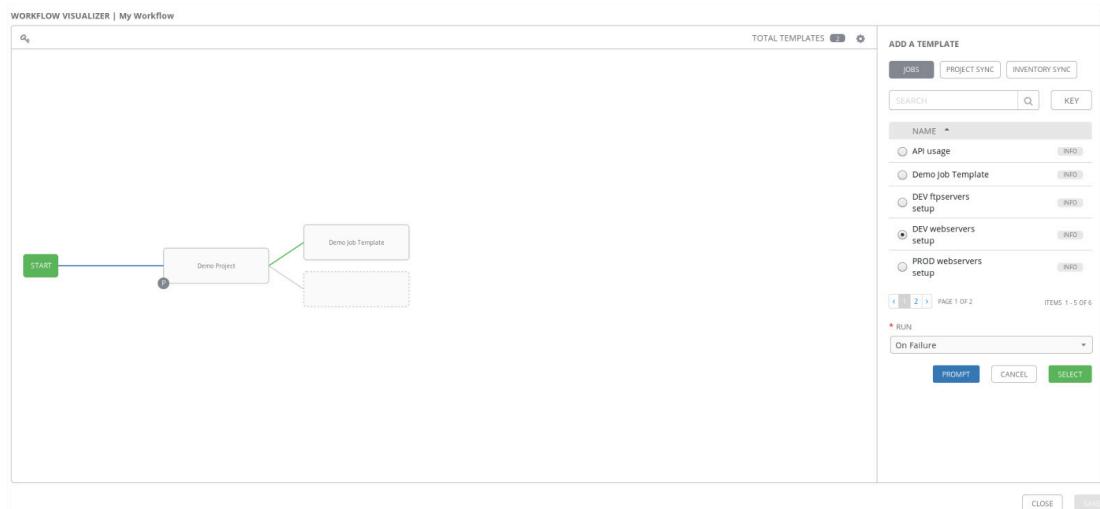
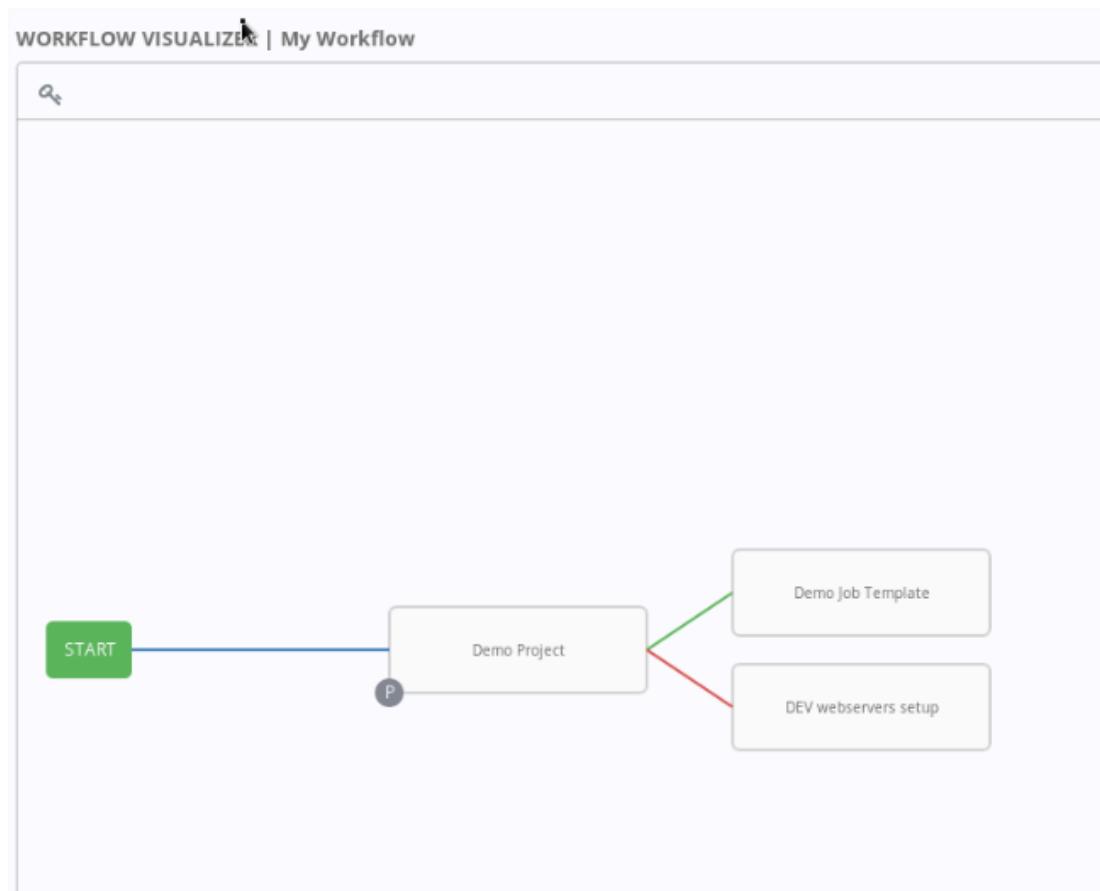


Figure 14.11: Adding multiple child nodes to the workflow

As nodes are added to a workflow, differently colored lines connecting the nodes in the Workflow Editor indicate the relationships between parent and child nodes. A green line indicates an On Success type relationship between a parent and child node, and a red line indicates an On Failure type relationship. A blue line indicates an Always type relationship.

**Figure 14.12: Workflow node relationships**

After the entire decision tree structure of the workflow has been created in the Workflow Editor, click **SAVE** to save the workflow.

Surveys

Workflow Job Templates have access to many of the features that have been discussed for Job Templates. Like Job Templates, Workflow Job Templates can have Surveys added to them to allow users to interactively set extra variables.



NOTE

When Surveys are added to a Workflow Job Template, the resulting extra variables are accessible by every job executed by the workflow.

LAUNCHING WORKFLOW JOBS

Like Job Templates, Users need the **execute** role on the Workflow Job Template to execute it. When assigned the **execute** role, a User can launch a job through a Workflow Job Template even if they do not have permissions to independently launch the Job Templates it uses.

The procedure to launch a Workflow Job Template is similar to how to launch a Job Template

1. Click **Templates** in the navigation bar to access the Template management interface.
2. Click the Workflow Job Template's rocket icon to launch the job.

Evaluating Workflow Job Execution

After a workflow job is launched, the Ansible Tower web UI displays the Job Details page for the job being executed. This page consists of two panes. The DETAILS pane displays details of the workflow job execution. The workflow progress pane displays the progress of the job through the steps in the workflow.

As each step is completed, its node is outlined in either green or red, indicating the success or failure of the actions associated with that step in the workflow. Progressions from one step to another are represented by colored lines indicating the decision responsible for the progression. Green indicates an **On Success** progression, and red indicates an **On Failure** progression. Blue indicates an **Always** progression.

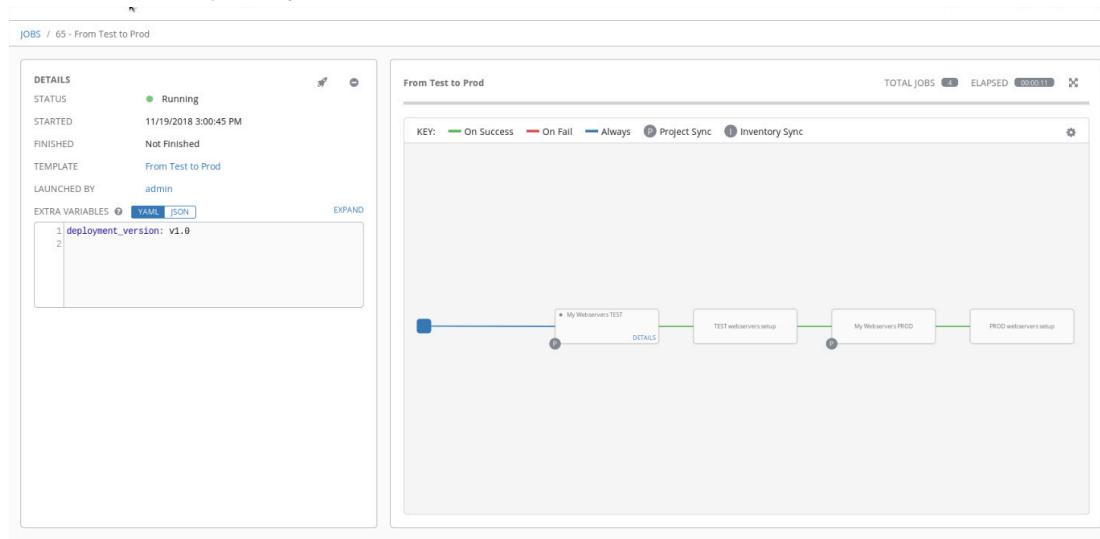


Figure 14.13: Workflow job progress

Details of the workflow job's run can be displayed either during or after the execution. Each node in the workflow diagram representing a currently running job or completed job provides a DETAILS link. You can click this link to display the results and standard output for the job run.



REFERENCES

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide/>

► GUIDED EXERCISE

CREATING WORKFLOW JOB TEMPLATES AND LAUNCHING WORKFLOW JOBS

In this exercise, you will create a Workflow Job Template to launch multiple jobs and launch it using the web UI.

OUTCOMES

You should be able to create a Workflow Job Template and launch a Workflow from the Ansible Tower web UI.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the `student` user on `workstation` and run `lab project-workflow setup`. This script ensures that the `workstation` and `tower` virtual machines are started.

```
[student@workstation ~]$ lab project-workflow setup
```

- ▶ 1. Log in to the Ansible Tower web UI running on the `tower` as `admin` using `redhat` as the password.
- ▶ 2. Create a Workflow Job Template called `From Dev to Test`.
 - 2.1. Click `Templates` in the left navigation bar.
 - 2.2. Click the `+` button to add a new Workflow Job Template.
 - 2.3. Select `Workflow Template` from the list.
 - 2.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	From Dev to Test
DESCRIPTION	Deploy to Dev and on success deploy to Test
ORGANIZATION	Default
EXTRA VARIABLES	<code>deployment_version: "v1.1"</code>

- 2.5. Click `SAVE` to create the new Workflow Job Template.

► 3. Configure the Workflow of the From Dev to Test Workflow Job Template.

- 3.1. Click WORKFLOW VISUALIZER to open the Workflow Visualizer.
- 3.2. Click START to add the first action to be performed. This displays a list of actions to be performed in the right panel.
- 3.3. In the right panel, click PROJECT SYNC to display the list of Projects available.
- 3.4. Select My Webservers DEV and click SELECT. This links the START node with a blue line (always perform) to the node for the Project, My Webservers DEV, in the Workflow Visualizer window.
- 3.5. Move your mouse over the new node and click on the green + button to add an action after the Project Sync of My Webservers DEV. This displays a list of actions to be performed in the right panel.
- 3.6. In the right panel, make sure you are in the JOBS section and select the DEV webservers setup Job Template.
- 3.7. In the RUN section below, select On Success and click SELECT. This links the node for the My Webservers DEV Project to a new node for the DEV webservers setup Job Template in the Workflow Visualizer window. The green link indicates that the progression only takes place upon success of the first step.
- 3.8. Move your mouse over the new node and click the green + button to add an action after the DEV webservers setup Job Template.
- 3.9. In the right panel, click PROJECT SYNC to display the list of available Projects.
- 3.10. Select My Webservers TEST. In the RUN section below, select On Success and click SELECT.
- 3.11. Move your mouse over the new node and click the green + button to add an action after the Project Sync of My Webservers TEST.
- 3.12. In the right panel, make sure you are in the JOBS section and select the TEST webservers setup Job Template. In the RUN section below, select On Success and click SELECT.
- 3.13. Click SAVE to save the Workflow Job Template.

► 4. Launch a job using the From Dev to Test Workflow Job Template.

- 4.1. Click Templates in the navigation bar.
- 4.2. On the same line as the From Dev to Test Workflow Job Template, click the rocket icon on the right to launch the Workflow. This redirects you to a detailed status page of the running Workflow.
- 4.3. Observe the running Jobs of the Workflow. You can click the DETAILS link of each running Job to see a more detailed live output of each Job.
- 4.4. Verify that the STATUS of the Workflow in the DETAILS section of the page is **Successful**.

- 5. Verify that the web servers have been updated on `servera.lab.example.com`, `serverb.lab.example.com`, `serverc.lab.example.com`, and `serverd.lab.example.com`.
- 5.1. Open a web browser and navigate to `http://servera.lab.example.com`, `http://serverb.lab.example.com`, `http://serverc.lab.example.com`, and `http://serverd.lab.example.com` in separate tabs. You should see this line at the bottom of each page:

```
Deployment Version: v1.1
```

- 6. Shut down `servera.lab.example.com` and launch a job using the `From Dev to Test` Workflow Job Template. Observe how the `DEV webservers setup` job node fails, which makes the whole workflow fail.
- 6.1. On `workstation`, open a terminal and open an SSH connection to `servera.lab.example.com`. Use `shutdown -h now` to shut down the server.

```
[student@workstation ~]$ ssh root@servera  
[root@servera ~]# shutdown -h now
```

- 6.2. Go back to the Ansible Tower web UI and click `Templates` in the navigation bar.
 - 6.3. On the same line as the `From Dev to Test` Workflow Job Template, click the rocket icon on the right to launch the Workflow. This redirects you to a detailed status page of the running Workflow.
 - 6.4. Observe the running Jobs of the Workflow. You can click the `DETAILS` link of each running Job to see a more detailed live output of each Job. When you click the `DETAILS` link for the failed `DEV webservers setup` node, you see that the job failed because `servera.lab.example.com` could not be reached. In this workflow example, when one of the nodes fails, the whole Workflow reports its status as being **Failed**.
 - 6.5. Verify that the `STATUS` of the Workflow in the `DETAILS` section of the page is **Failed**.
- 7. Start the `servera.lab.example.com` server.

This concludes the guided exercise.

SCHEDULING JOBS AND CONFIGURING NOTIFICATIONS

OBJECTIVES

After completing this section, students should be able to schedule automatic job execution and configure notification of job completion.

Figure 14.13: Configuring automatic notification of Job Success and Failure

SCHEDULING JOB EXECUTION

Sometimes you might need to launch a job template automatically at a particular time, or on a particular schedule. Red Hat Ansible Tower lets you configure *scheduled jobs*, which launch Job Templates on a customizable schedule.

If you have the Execute role on a Job Template, you can launch jobs from that template by setting up a schedule. To configure scheduled jobs, first select **Templates** from the left navigation bar. Click the Job Template that you want to schedule, and in the pane on the right, click **SCHEDULES**. Click the **+** button to create a new schedule for that Job Template.

Figure 14.14: Scheduling Job execution

Enter the required details:

- **NAME:** The name of the schedule.
- **START DATE:** The date the job schedule should start.
- **START TIME:** The time the associated Job will be launched.
- **LOCAL TIME ZONE:** The **tzselect** command-line utility can be used to determine your local time zone in this format.
- **REPEAT FREQUENCY:** How often to repeat the associated job. You can choose None (run once), Minute, Hour, Day, Week, Month, or Year.

Depending on the chosen frequency, you might need to provide additional information (for example, to launch a job every two days, or on the first Sunday of every month).

When finished, click **SAVE** to save the schedule. You can deactivate or activate a schedule using the ON/OFF button next to the schedule name.

Temporarily Disabling a Schedule

Click **Schedules** in the left navigation bar to display the **Scheduled Jobs** page. This page lists all the defined schedules. To the left of each schedule's name is an ON/OFF button. Set this to ON or OFF to activate or deactivate the schedule, respectively.

You can also edit or delete any schedule, assuming you have sufficient privileges to do so, from this page.

Scheduled Management Jobs

Red Hat Ansible Tower ships with two special scheduled jobs by default. These two schedules are for built-in *Management Jobs* that perform periodic maintenance on the Ansible Tower server itself, by cleaning up old log information for the Activity Stream and historic job execution.

Cleanup Job Schedule deletes the details of historic jobs to save space. It runs once a week on Sundays to remove information about jobs more than 120 days old by default, but you can change when this runs and how much data it keeps by editing the schedule.

Cleanup Activity Schedule runs once a week on Tuesdays to remove information from the activity stream that is more than 355 days old. Again, you can change when it runs and how much data it keeps by editing the schedule.

REPORTING JOB EXECUTION RESULTS

One of the benefits of using Ansible Tower to manage an enterprise's Ansible infrastructure is centralized logging and auditing. When a job is executed, details regarding the job execution are logged in the Ansible Tower database. This database can be referenced by users at a later time to determine the historical results of past job executions.

Historical job execution details are helpful to administrators for confirming the success and failure of scheduled and delegated job executions. The ability to retrieve historical job execution details is helpful, but for jobs related to critical functions, administrators likely desire immediate notification of a job's success or failure.

Red Hat Ansible Tower can send immediate alerts of job execution results. To use this feature, administrators create **Notification Templates** which define how notifications are to be sent. Ansible Tower supports many mechanisms to send notifications. Some are based on open protocols such as email and IRC, and others are based on proprietary solutions, such as HipChat and Slack.

Notification Templates

A **Notification Template** is defined in the context of an Organization. After it has been created, the Notification Template can be used to send notifications of the results of jobs run by Ansible Tower for that Organization.

A Notification Template defines the mechanism for how a notification is to be sent. Supported mechanisms include:

- Email
- Slack
- Twilio
- PagerDuty

- HipChat
- Webhook
- IRC

Creating Notification Templates

Notification Templates are created through the Notifications interface, accessible from the left navigation bar. Depending on the selected notification mechanism, the TYPE DETAILS section of the Notification Template interface prompts for different user input.

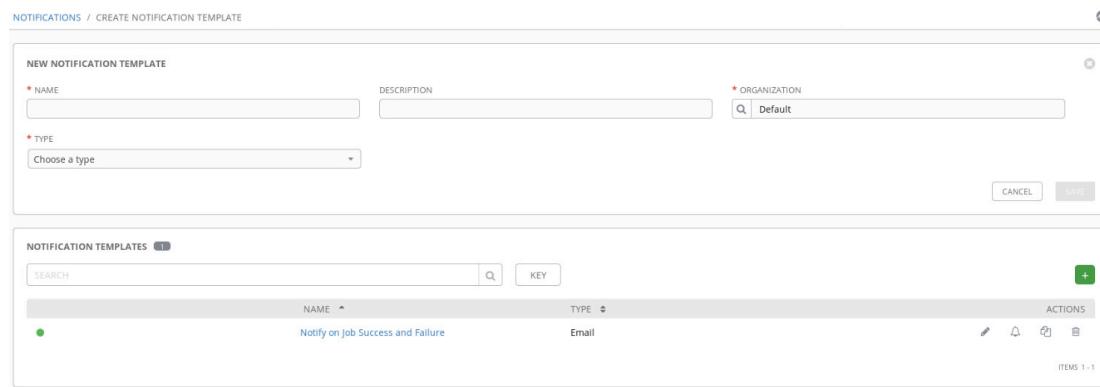


Figure 14.15: Creating notifications

The following steps are used to create an Email type Notification Template.

1. In the NOTIFICATIONS interface, click the + button to create a new Notification Template.
2. Enter a unique name for the Notification Template in the NAME field.
3. In the ORGANIZATION field, specify the Organization within which to create the Notification Template.
4. In the TYPE list, select Email as the mechanism to be used by the Notification Template for generating notifications. After you have selected the notification type, type-specific user input fields display under the TYPE DETAILS section.
5. In the SENDER EMAIL field, specify the sender's email address to be used when composing the notification email.
6. In the RECIPIENT LIST field, specify the email addresses of the recipients for the notification email, one on each line.
7. In the PORT field, specify the port to connect to on the SMTP relay host.
8. You can complete the fields for configuring SMTP authentication and for enabling secure transport, but these are optional.
9. Click SAVE to save the Notification Template.

Enabling Job Result Notification

After a Notification Template has been created, it becomes available for use by certain Ansible Tower resources that are part of the Notification Template's Organization, such as Job Templates, Projects, or Workflows.

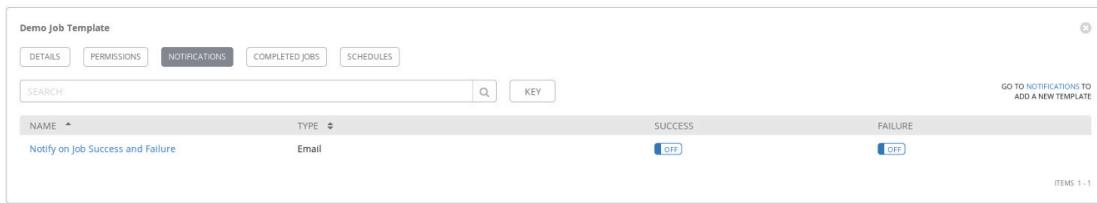


Figure 14.16: Enabling notifications on a Job Template

The following steps enable notifications for a Job Template.

1. Click **Templates** on the left navigation bar to display the list of templates.
2. Click the name of the required Job Template and then click **NOTIFICATIONS**.
3. A list of Notification Templates for the Organization is displayed in the **NOTIFICATIONS** interface.
4. Each listed Notification Template has controls for toggling success and failure notifications. Set the **SUCCESS** and **FAILURE** controls to achieve the desired notification configuration for the Job Template.



NOTE

You can also use Notification Templates to enable notifications for system jobs triggered by Project and Inventory resources to synchronize their data.



REFERENCES

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide>

► GUIDED EXERCISE

SCHEDULING JOBS AND CONFIGURING NOTIFICATIONS

In this exercise, you will create an email Notification Template, configure a Job Template to use the Notification Template, and launch a job to confirm that the notification works.

OUTCOMES

You should be able to create a Notification Template and use it with a Job Template to generate email notifications of job results.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in to `workstation` as `student` using `student` as the password and run **lab project-notification setup**. This script will set up the SMTP server.

```
[student@workstation ~]$ lab project-notification setup
```

- ▶ 1. Log in to the Ansible Tower web UI running on the `tower` system as `admin` using `redhat` as the password.
- ▶ 2. Add a Notification Template to the `DEV webservers` setup Job Template.
 - 2.1. Click Notifications in the navigation bar.
 - 2.2. Click `+` to add a Notification Template.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Notify on Job Success and Failure
DESCRIPTION	Sends an email to notify the status of the Job
ORGANIZATION	Default
TYPE	Email
HOST	localhost
RECIPIENT LIST	student@tower.lab.example.com
SENDER EMAIL	system@tower.lab.example.com

FIELD	VALUE
PORT	25

- 2.4. Leave all the other fields untouched and click **SAVE** to save the Notification Template. You will be redirected to the list of Notification Templates.
- 3. Validate the Notification Template.
- 3.1. On the same line as the Notification Template named **Notify on Job Success and Failure**, click the bell icon on the right to test the Notification process.
 - 3.2. Wait several seconds. You should see a green notification that reads "Notify on Job Success and Failure: Notification sent."
- 4. Configure the **DEV webservers setup** Job Template to notify you when its jobs succeeds or fails. Use the **Notify on Job Success and Failure** Notification Template.
- 4.1. Click **Templates** in the navigation bar.
 - 4.2. From the list of available Job Templates, click **DEV webservers setup** to edit it.
 - 4.3. Click **NOTIFICATIONS** to manage Notifications for that Job Template.
 - 4.4. On the same line as the **Notify on Job Success and Failure** Notification Template, set the ON/OFF switches for both **SUCCESS** and **FAILURE** to **ON**.
- 5. Verify that the **DEV webservers setup** Job Template triggers a notification email after job completion.
- 5.1. Open a terminal and connect to the **tower** VM.

```
[student@workstation ~]$ ssh tower
Last login: Thu Apr 20 11:33:22 2017 from workstation.lab.example.com
[student@tower ~]$
```

- 5.2. Use the **tail** command to read incoming messages to the local mailbox file of the student user. You should see the email that was sent by the Test Notification launched from a previous step.

```
[student@tower ~]$ tail -f /var/mail/student
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Tower Notification Test 1 https://tower.lab.example.com
From: system@tower.lab.example.com
To: student@tower.lab.example.com
Date: Thu, 08 Nov 2018 13:35:55 -0000
Message-ID: <20181108133555.1787.75547@tower.lab.example.com>
```

Ansible Tower Test Notification 1 <https://tower.lab.example.com>

- 5.3. Go back to the web UI and click **Templates** in the navigation bar.
- 5.4. On the same line as the **DEV webservers setup** Job Template, click the rocket icon on the right to launch the Job.
- 5.5. In the Survey window, enter **v1.2** for the deployment version, click **NEXT** and then click **LAUNCH**.
- 5.6. Go to your terminal running the **tail** command and wait. After about one minute, you should see a notification email arrive that looks similar to the following example:

```
From system@tower.lab.example.com Thu Nov  8 08:39:50 2018
Return-Path: <system@tower.lab.example.com>
X-Original-To: student@tower.lab.example.com
Delivered-To: student@tower.lab.example.com
Content-Type: text/plain; charset="utf-8"
Subject: Job #41 'DEV webservers setup' succeeded:
  https://tower.lab.example.com/#/jobs/playbook/41
From: system@tower.lab.example.com
To: student@tower.lab.example.com
Date: Thu, 08 Nov 2018 13:39:50 -0000
Status: R

Job #41 had status successful, view details at https://tower.lab.example.com/#/
jobs/playbook/41

{
  "status": "successful",
  "credential": "Developers",
  "name": "DEV webservers setup",
  "started": "2018-11-08T13:39:16.477269+00:00",
  "extra_vars": "{}",
  "traceback": "",
  "friendly_name": "Job",
  "created_by": "admin",
  "project": "My Webservers DEV",
  "url": "https://tower.lab.example.com/#/jobs/playbook/41",
  "finished": "2018-11-08T13:39:48.813859+00:00",
  "hosts": {
    "serverb.lab.example.com": {
      "skipped": 0,
      "ok": 6,
      "changed": 0,
      "dark": 0,
      "failed": false,
      "processed": 1,
      "failures": 0
    }
  },
  "credential": "Developers",
  "name": "DEV webservers setup",
  "started": "2018-11-08T13:39:16.477269+00:00",
  "extra_vars": "{}",
  "traceback": "",
  "friendly_name": "Job",
  "created_by": "admin",
  "url": "https://tower.lab.example.com/#/jobs/playbook/41"
}
```

```

"project": "My Webservers DEV",
"url": "https://tower.lab.example.com/#/jobs/playbook/41",
"finished": "2018-11-08T13:39:48.813859+00:00",
"hosts": {
    "serverb.lab.example.com": {
        "skipped": 0,
        "ok": 6,
        "changed": 0,
        "dark": 0,
        "failed": false,
        "processed": 1,
        "failures": 0
    },
    "servera.lab.example.com": {
        "skipped": 0,
        "ok": 6,
        "changed": 0,
        "dark": 0,
        "failed": false,
        "processed": 1,
        "failures": 0
    }
},
"playbook": "apache-setup.yml",
"limit": "",
"id": 41,
"inventory": "Dev"
}

```

- 5.7. Exit the terminal session on the tower system.

```
[student@tower ~]$ exit
```

- ▶ 6. Verify that the web servers have been updated successfully on servera.lab.example.com and serverb.lab.example.com.
- 6.1. Open a web browser and navigate to <http://servera.lab.example.com> and <http://serverb.lab.example.com> in separate tabs. You should see this line at the bottom of each page:

```
Deployment Version: v1.2
```

- 7. Based on the `DEV webservers setup` Job Template, schedule a new job at three minutes from the current time.

- 7.1. Click **Templates** in the navigation bar.
- 7.2. From the list of available Job Templates, click `DEV webservers setup` to edit that template.
- 7.3. Click **SCHEDULES** to manage automatic execution of a job based on this Job Template.
- 7.4. Click the **+** button to add a new schedule.
- 7.5. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Automatic job run
START TIME	Look at the current time on workstation (possibly by using the date command), and set the execution of the job to +3 minutes from the current time.

- 7.6. Click **SAVE** to create the new schedule. Click **Jobs** in the navigation bar and wait for the scheduled job to be executed.

This concludes the guided exercise.

LAUNCHING JOBS WITH THE ANSIBLE TOWER API

OBJECTIVE

After completing this section, students should be able to control Ansible Tower by accessing its API from Ansible Playbooks.

Figure 14.16: Launching Jobs with Ansible Tower API

THE RED HAT ANSIBLE TOWER REST API

Red Hat Ansible Tower provides a Representational State Transfer (REST) API. An API provides a mechanism that allows administrators and developers to control their Ansible Tower environment outside of the web UI. Custom scripts or external applications access the API using standard HTTP messages. This is useful when integrating Ansible Tower with other programs.

One of the benefits of the REST API is that any programming language, framework, or system with support for the HTTP protocol can use the API. This provides an easy way to automate repetitive tasks and integrate other enterprise IT systems with Ansible Tower.



NOTE

The API is in active development, and not all features of the web UI may be accessible through the API. There are two versions of the API currently available. We expect that version 1 will be deprecated in the near future.

Using the REST API

In case you are not familiar with REST APIs, the way they work is relatively straightforward. A client sends a request to a server element located at a Uniform Resource Identifier (URI) and performs operations with standard HTTP methods, such as GET, POST, PUT, and DELETE. The REST architecture provides a stateless communication channel between the client and server. Each client request acts independently of any other request, and contains all the necessary information to complete the request.

The following example request uses an HTTP GET method to retrieve a representation of the main entry point of the API. A graphical web browser or Linux command-line tools could be used to issue the request manually. This example uses the **curl** command to make the request from the command line:

```
[user@demo ~]$ curl -X GET https://tower.lab.example.com/api/ -k
{"description":"AWX REST API","current_version":"/api/v2/","available_versions":
{"v1":"/api/v1/","v2":"/api/v2/"}, "oauth2":"/api/
0/","custom_logo":"","custom_login_info":""}
```

The output of the API request is in JSON format, which is readily parsable by computer programs, but may be a little challenging for a human to read.

The Ansible Tower API is browsable. For example, if your Ansible Tower server is the host `tower.lab.example.com`, you can access the browsable API at `https://tower.lab.example.com/api/`.

`tower.lab.example.com/api/`. You can click the `/api/v2/` link on that page to browse information specific to version 2 of the API.

The following example shows how to do this using `curl`. The `json_pp` command is provided by the `perl-JSON-PP` RPM package and "pretty-prints" the JSON output of the API for easier reading by a human.

```
[user@demo ~]$ curl -X GET https://tower.lab.example.com/api/v2/ -k -s | json_pp
{
    "dashboard" : "/api/v2/dashboard/",
    "unified_jobs" : "/api/v2/unified_jobs/",
    "ping" : "/api/v2/ping/",
    "users" : "/api/v2/users/",
    "me" : "/api/v2/me/",
    "system_jobs" : "/api/v2/system_jobs/",
    "ad_hoc_commands" : "/api/v2/ad_hoc_commands/",
    "instance_groups" : "/api/v2/instance_groups/",
    "workflow_job_template_nodes" : "/api/v2/workflow_job_template_nodes/",
    "inventory_sources" : "/api/v2/inventory_sources/",
    "applications" : "/api/v2/applications/",
    "inventory" : "/api/v2/inventories/",
    "project_updates" : "/api/v2/project_updates/",
    "tokens" : "/api/v2/tokens/",
    "notifications" : "/api/v2/notifications/",
    "notification_templates" : "/api/v2/notification_templates/",
    "inventory_updates" : "/api/v2/inventory_updates/",
    "teams" : "/api/v2/teams/",
    "hosts" : "/api/v2/hosts/",
    "credentials" : "/api/v2/credentials/",
    "schedules" : "/api/v2/schedules/",
    "unified_job_templates" : "/api/v2/unified_job_templates/",
    "jobs" : "/api/v2/jobs/",
    "instances" : "/api/v2/instances/",
    "config" : "/api/v2/config/",
    "roles" : "/api/v2/roles/",
    "organizations" : "/api/v2/organizations/",
    "system_job_templates" : "/api/v2/system_job_templates/",
    "credential_types" : "/api/v2/credential_types/",
    "inventory_scripts" : "/api/v2/inventory_scripts/",
    "workflow_job_nodes" : "/api/v2/workflow_job_nodes/",
    "job_events" : "/api/v2/job_events/",
    "groups" : "/api/v2/groups/",
    "workflow_job_templates" : "/api/v2/workflow_job_templates/",
    "job_templates" : "/api/v2/job_templates/",
    "activity_stream" : "/api/v2/activity_stream/",
    "settings" : "/api/v2/settings/",
    "labels" : "/api/v2/labels/",
    "workflow_jobs" : "/api/v2/workflow_jobs/",
    "projects" : "/api/v2/projects/"
}
```

This entry point provides a collection of links in the API environment. As you can see in the example, there are many links to choose from.

The following example illustrates information accessible through the API. To examine what actions have been performed on the Ansible Tower server, you can use the `/api/v2/`

activity_stream/ URI. Make a GET request to that resource to retrieve the list of activity streams:

```
[user@demo ~]$ curl -X GET \
> https://tower.lab.example.com/api/v2/activity_stream/ -k
>{"detail":"Authentication credentials were not provided. To establish a login
session, visit /api/login/.\"}
```

As you can see in the output above, not all information generated by the API is publicly available. You need to log in to access this resource.

The next example shows the output of the **activity_stream** resource when correct authentication information is provided:

```
[user@demo ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/activity_stream/ -k -s | json_pp
{
  "next" : "/api/v2/activity_stream/?page=2",
  "previous" : null,
  "count" : 212,
  "results" : [
    {
      ...
      .output omitted...
      "summary_fields" : {
        "user" : [
          {
            "id" : 4,
            "last_name" : "Stephens",
            "first_name" : "Simon",
            "username" : "simon"
          }
        ],
        "actor" : {
          "id" : 1,
          "last_name" : "",
          "first_name" : "",
          "username" : "admin"
        }
      },
      "timestamp" : "2018-11-07T15:43:28.936831Z",
      "changes" : {
        "password" : [
          "hidden",
          "hidden"
        ]
      },
      "url" : "/api/v2/activity_stream/25/",
      "id" : 25,
      "type" : "activity_stream",
      "operation" : "update",
      "object2" : ""
      .output omitted...
    }
  ]
}
```

**IMPORTANT**

The output of the API may be *paginated*, as in the preceding example. Ansible Tower only returns a limited number of records for a particular request for performance reasons. The **next** value gives the URI for the next page of results. If it is **null**, you are on the last page. Likewise, the value of **previous** is the URI for the previous page of results, and if it is **null** you are on the first page.

The output of the API is in JSON format, which can be difficult to read without running it through a parser such as **json_pp**. You can also access the browsable REST API with a graphical web browser to get the same information in a more readable format. This example accesses the same API using the Firefox web browser:

```

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
X-API-Node: localhost
X-API-Time: 2.260s

{
  "count": 179,
  "next": "/api/v2/activity_stream/?page=2",
  "previous": null,
  "results": [
    {
      "id": 1,
      "type": "activity_stream",
      "url": "/api/v2/activity_stream/1",
      "related": {
        "setting": "/api/v2/settings/ui/"
      },
      "summary_fields": {
        "setting": [
          {
            "category": "ui",
            "name": "PENDO_TRACKING_STATE"
          }
        ]
      }
    }
  ]
}

```

Figure 14.17: Activity stream API output

You can click various links in the API to explore related resources.

```

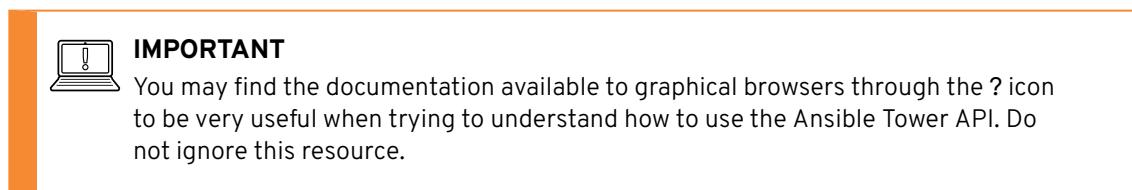
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
X-API-Node: localhost
X-API-Time: 0.044s

{
  "ping": "/api/v2/ping/",
  "instances": "/api/v2/instances/",
  "instance_groups": "/api/v2/instance_groups/",
  "config": "/api/v2/config/",
  "settings": "/api/v2/settings/",
  "me": "/api/v2/me/",
  "dashboard": "/api/v2/dashboard/",
  "organizations": "/api/v2/organizations/",
  "users": "/api/v2/users/",
  "projects": "/api/v2/projects/"
}

```

Figure 14.18: Browsable API

Click the ? icon next to an API endpoint name to get documentation on the access methods for that endpoint. The documentation also provides information on what data is returned when using those methods.



This screenshot shows the 'List Jobs:' section of the Ansible Tower API documentation. It includes a description of the GET request, the resulting data structure, and a JSON schema example. Below this, there is a 'Results' section with a list of fields for each job data structure.

- id**: Database ID for this job. (integer)
- type**: Data type for this job. (choice)
- url**: URL for this job. (string)

Figure 14.19: Documentation for API endpoints

You can also use PUT or POST methods on the specific API pages by providing JSON formatted text or files in the graphical interface.

This screenshot shows the 'CONTENT:' field for a POST method. It displays a JSON schema for creating a new job template. A green 'POST' button is visible at the bottom right.

```
{
  "name": "",
  "description": "",
  "job_type": "run",
  "inventory": null,
  "project": null,
  "playbook": "",
  "forks": 0,
  "limit": "",
  "verbosity": 0,
  "extra_vars": ""
}
```

Figure 14.20: Example of POST method text field

LAUNCHING A JOB TEMPLATE USING THE API

One common use of the API is to launch an existing Job Template. This example uses the **curl** command to quickly outline how to use the API to find and launch a Job Template that has already been configured in Ansible Tower.

Starting with Red Hat Ansible Tower 3.2, you can refer to a Job Template by name in the API.

For example, you can use the GET method to get information about a Job Template. The following example illustrates how to do it for the Demo Job Template Job Template. Because the name of

the Job Template contains spaces, you need to escape them using double quotes or URL percent encoding (%20 for each space character).

```
[user@demo ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/"Demo Job Template/" -k -s \
> | json_pp
```

Launching a Job Template from the API is done in two steps. First, you must access it with the GET method to get information about any parameters or data that you might need to launch the job. Then, you must access it with the POST method to actually launch the job.

The following example uses the GET method to launch the Demo Job Template Job Template through the API. The output is piped into **json_pp** for better readability.

```
[user@demo ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/"Demo Job Template"/launch/ - 
k -s | json_pp
{
  "job_template_data" : {
    "name" : "Demo Job Template",
    "id" : 5,
    "description" : ""
  },
  "ask_inventory_on_launch" : false,
  "defaults" : {
    "inventory" : {
      "name" : "Demo Inventory",
      "id" : 1
    },
    "verbosity" : 0,
    "extra_vars" : "",
    "credentials" : [
      {
        "passwords_needed" : [],
        "credential_type" : 1,
        "name" : "Demo Credential",
        "id" : 1
      }
    ],
    "diff_mode" : false,
    "skip_tags" : "",
    "job_type" : "run",
    "job_tags" : "",
    "limit" : ""
  },
  "ask_skip_tags_on_launch" : false,
  "passwords_needed_to_start" : [],
  "ask_variables_on_launch" : false,
  "ask_credential_on_launch" : false,
  "inventory_needed_to_start" : false,
  "ask_limit_on_launch" : false,
  "survey_enabled" : false,
  "ask_diff_mode_on_launch" : false,
  "ask_verbosity_on_launch" : false,
  "credential_needed_to_start" : false,
```

```

    "ask_job_type_on_launch" : false,
    "can_start_without_user_input" : true,
    "variables_needed_to_start" : [],
    "ask_tags_on_launch" : false
}

```

Most of this information is discussed in more detail in the *Ansible Tower API Guide* in the chapter on launching job templates. Notice that the **id** and name of the Inventory and machine Credential for the job is listed, and no extra information is needed to launch the job.

In this case, you can launch the job by accessing the URI with a POST method instead of a GET method. The entire output from the API call is shown in the following example. In particular, note the job **id** (72), that its **status** is pending, because it has been launched but has not completed yet, and the other information about the job that is initially returned.

```

[user@demo ~]$ curl -X POST --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/"Demo Job Template"/launch/ - 
k -s | json_pp
{
    "result_traceback" : "",
    "passwords_needed_to_start" : [],
    "elapsed" : 0,
    "ignored_fields" : {},
    "skip_tags" : "",
    "summary_fields" : {
        "unified_job_template" : {
            "unified_job_type" : "job",
            "name" : "Demo Job Template",
            "id" : 5,
            "description" : ""
        },
        "inventory" : {
            "organization_id" : 1,
            "total_hosts" : 1,
            "total_inventory_sources" : 0,
            "name" : "Demo Inventory",
            "inventory_sources_with_failures" : 0,
            "has_inventory_sources" : false,
            "description" : "",
            "kind" : "",
            "total_groups" : 0,
            "id" : 1,
            "hosts_with_active_failures" : 0,
            "groups_with_active_failures" : 0,
            "has_active_failures" : false
        },
        "extra_credentials" : [],
        "project" : {
            "scm_type" : "",
            "status" : "ok",
            "name" : "Demo Project",
            "id" : 4,
            "description" : ""
        },
        "credentials" : [
            {

```

```

        "cloud" : false,
        "kind" : "ssh",
        "credential_type_id" : 1,
        "name" : "Demo Credential",
        "id" : 1,
        "description" : ""
    }
],
"user_capabilities" : {
    "delete" : true,
    "start" : true
},
"job_template" : {
    "name" : "Demo Job Template",
    "id" : 5,
    "description" : ""
},
"modified_by" : {
    "id" : 1,
    "last_name" : "",
    "first_name" : "",
    "username" : "admin"
},
"labels" : {
    "count" : 0,
    "results" : []
},
"credential" : {
    "cloud" : false,
    "kind" : "ssh",
    "credential_type_id" : 1,
    "name" : "Demo Credential",
    "id" : 1,
    "description" : ""
},
"created_by" : {
    "id" : 1,
    "last_name" : "",
    "first_name" : "",
    "username" : "admin"
},
},
"timeout" : 0,
"url" : "/api/v2/jobs/72/",
"instance_group" : null,
"id" : 72,
"scm_revision" : "",
"ask_credential_on_launch" : false,
"job_cwd" : "",
"unified_job_template" : 5,
"vault_credential" : null,
"playbook" : "hello_world.yml",
"name" : "Demo Job Template",
"ask_limit_on_launch" : false,
"description" : "",
"ask_diff_mode_on_launch" : false,

```

```

"modified" : "2018-11-20T11:09:50.004538Z",
"related" : {
    "relaunch" : "/api/v2/jobs/72/relaunch/",
    "project" : "/api/v2/projects/4/",
    "create_schedule" : "/api/v2/jobs/72/create_schedule/",
    "job_host_summaries" : "/api/v2/jobs/72/job_host_summaries/",
    "credentials" : "/api/v2/jobs/72/credentials/",
    "notifications" : "/api/v2/jobs/72/notifications/",
    "modified_by" : "/api/v2/users/1/",
    "stdout" : "/api/v2/jobs/72/stdout/",
    "labels" : "/api/v2/jobs/72/labels/",
    "inventory" : "/api/v2/inventories/1/",
    "unified_job_template" : "/api/v2/job_templates/5/",
    "extra_credentials" : "/api/v2/jobs/72/extra_credentials/",
    "job_events" : "/api/v2/jobs/72/job_events/",
    "activity_stream" : "/api/v2/jobs/72/activity_stream/",
    "job_template" : "/api/v2/job_templates/5/",
    "cancel" : "/api/v2/jobs/72/cancel/",
    "credential" : "/api/v2/credentials/1/",
    "created_by" : "/api/v2/users/1/"
},
"forks" : 0,
"job_type" : "run",
"ask_job_type_on_launch" : false,
"job_tags" : "",
"type" : "job",
"credential" : 1,
"finished" : null,
"ask_inventory_on_launch" : false,
"job_env" : {},
"use_fact_cache" : false,
"artifacts" : {},
"project" : 4,
"status" : "pending",
"event_processing_finished" : false,
"diff_mode" : false,
"ask_skip_tags_on_launch" : false,
"failed" : false,
"execution_node" : "",
"job_explanation" : "",
"ask_variables_on_launch" : false,
"launch_type" : "manual",
"limit" : "",
"allow_simultaneous" : false,
"job_args" : "",
"start_at_task" : "",
"inventory" : 1,
"verbosity" : 0,
"extra_vars" : "{}",
"job" : 72,
"created" : "2018-11-20T11:09:49.813893Z",
"force_handlers" : false,
"ask_verbosity_on_launch" : false,
"job_template" : 5,
"controller_node" : "",
"started" : null,

```

```
"ask_tags_on_launch" : false
}
```

The JSON formatted output of this example shows the **id** of this job happens to be 72. You can use the job id to retrieve updated status information, such as whether or not the job has completed. For job 72, you would use the URL [/api/v2/jobs/72/](https://tower.lab.example.com/api/v2/jobs/72/), as indicated by the **url** field in the preceding output.

```
[user@demo ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/jobs/72/ -k -s | json_pp
```

This reports job **status** (success or failure), at what time the job finished, the **result_stdout** of the Ansible Playbook, which playbook was used, as well as other information about the inventory, credentials, and project from the job template, and more.



NOTE

You can also launch Job Templates using the internal ID number instead of their name. In earlier versions of Red Hat Ansible Tower, using the version 1 API, you had to launch Job Templates only using the ID number.

First, you need to find the **id** number of your Job Template. If you know the name of the Job Template, you can use the API to search for it. For example, if the desired Job Template is named Demo Job Template, you can search for it with the following **curl** command:

```
[user@demo ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/?name="A Job Template" \
> -k -s | json_pp
...output omitted...
{
    "timeout" : 0,
    "url" : "/api/v2/job_templates/6/",
    "id" : 6,
    "ask_credential_on_launch" : false,
    "last_job_failed" : false,
    "vault_credential" : null,
    "playbook" : "hello_world.yml",
    "name" : "A Job Template",
    ...output omitted...
```

Then you can refer to the Job Template using its ID number and not its name in the URL, as follows:

```
[user@demo ~]$ curl -X POST --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/6/launch/ -k -s
```

LAUNCHING A JOB USING THE API FROM AN ANSIBLE PLAYBOOK

You can use an Ansible Playbook to launch a Job Template by using the **uri** module to access the Ansible Tower API. It is also possible to run that playbook from a Job Template in Ansible Tower to launch another Job Template as one of its tasks.

In the playbook, you need to specify the correct URL to your Job Template, using the ID or the Named URL. You also need to provide sufficient credentials to Ansible Tower to authenticate as a user who has permission to launch the job.

The following Ansible Playbook can start a new job from one of the Ansible Tower server's existing Job Templates, using the Ansible Tower API:

```
---
- name: Tower API
  hosts: localhost
  become: false

  vars:
    tower_user: admin①
    tower_pass: redhat
    tower_host: demo.example.com
    tower_job: Demo%20Job%20Template②

  tasks:
    - name: Launch a new Job
      uri:③
        url: https://{{ tower_host }}/api/v2/job_templates/{{ tower_job }}/launch/
        method: POST
        validate_certs: no
        return_content: yes
        user: "{{ tower_user }}"
        password: "{{ tower_pass }}"
        force_basic_auth: yes
        status_code: 201
```

- ① To access the API, Ansible needs the login credentials for a user that is allowed to launch a Job from a Job Template. In the example, two variables are used to store the relevant information: `tower_user` and `tower_pass`.
- ② The playbook also needs the URL of the actual API that Ansible has to connect to. This example uses the named URL to the Demo Job Template. Notice how the spaces in the Job Template name have been specified using the `%20` code.
- ③ Ansible can access the Ansible Tower API from the Ansible Tower server itself by using the `uri` module.

The problem with this example is that it embeds the username and password for authentication to the Ansible Tower server in the playbook. To protect that data, you should either encrypt the playbook with Ansible Vault, or move the secrets into a variable file and encrypt that file with Ansible Vault. You should do this before you commit files containing those secrets to your source control repository.

```
[user@demo ~]$ ansible-vault encrypt api_demo.yml
New Vault password: your_password
Confirm New Vault password: your_password
```

Vault Credentials

For Ansible Tower to use encrypted files (such as a playbook or included variable file containing secrets), you need to set up a Vault Credential in Ansible Tower that can decrypt those files.

You also need to configure any Job Templates that use that Project with that Vault Credential in addition to any machine credentials or other credentials that Project needs.

First, you need to create a Vault Credential that stores the Vault password for those files. This Credential is encrypted and stored in the Ansible Tower server's database, just like Machine Credentials. The following procedure describes how to create this type of Credential:

1. Log in as a User with the appropriate role assignment. If creating a private Credential, there are no specific role requirements. If creating an Organization Credential, log in as a User with the Admin role for the Organization.
2. Click **Credentials** in the left navigation bar to open the Credentials management interface.
3. On the **CREDENTIAL** screen, click **+** to create a new Credential.
4. On the **NEW CREDENTIAL** screen, enter the required information for the new Credential.

Enter a name for the new Credential, and then select **Vault** from the **TYPE** list.

If the user has Organization Admin privileges, the **ORGANIZATION** can be set to assign this Credential to an Organization. If the User does not have admin privileges, the **ORGANIZATION** field is not present and only private Credentials can be created.

5. For Vault Credentials, additional fields appear in the **TYPE DETAILS** section, as shown in the following illustration:

The screenshot shows the 'CREDENTIALS / EDIT CREDENTIAL' dialog for a 'vault' credential. The 'DETAILS' tab is selected. The 'NAME' field contains 'vault-credentials-demo'. The 'DESCRIPTION' field contains 'Vault Credentials'. The 'ORGANIZATION' dropdown is set to 'Default'. In the 'CREDENTIAL TYPE' section, 'Vault' is selected. The 'TYPE DETAILS' section includes fields for 'VAULT PASSWORD' (checkboxes for 'REPLACE' and 'ENCRYPTED') and 'VAULT IDENTIFIER' (checkbox for 'Prompt on launch'). At the bottom right are 'CANCEL' and 'SAVE' buttons.

Figure 14.21: New Vault credential (after Save)

6. Two fields contain the information needed to decrypt the Vault protected Playbooks.
 - **VAULT PASSWORD** is the password with which the playbook has been encrypted.
 - **VAULT IDENTIFIER** is the optional Vault ID, only needed if the playbook has been encrypted using multiple passwords.
7. Click **SAVE** to save the new Vault Credential.

After you have the Vault Credential in place, you can create a new Job Template that will use it to decrypt your encrypted project file or files. The Job Template creation process is almost the same as before, the only difference being that the Job Template must include the Vault Credential needed to decrypt the project files as one of its Credentials.

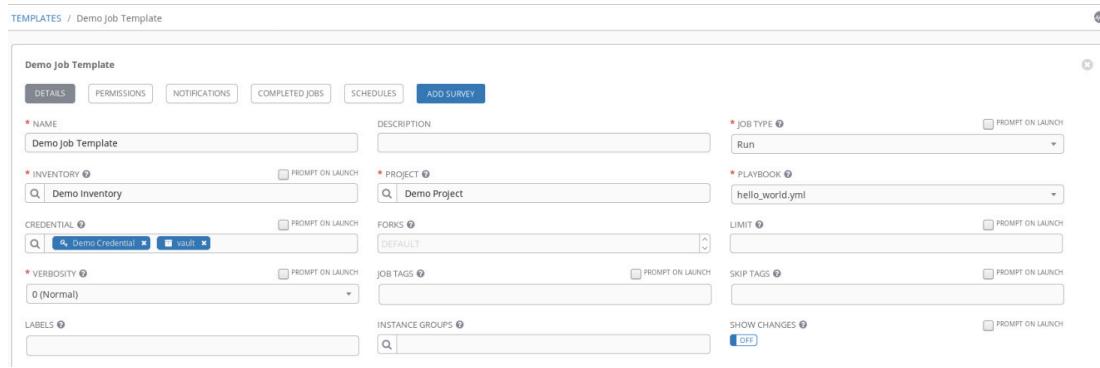


Figure 14.22: Job Template with Vault Credential

When you have finished, a job can be launched using the new Job Template. When you launch the Job Template, Ansible Tower decrypts the encrypted playbook using the Vault Credential. When it runs the playbook, the task that accesses the API to launch another Job Template is executed. This launches a new job on the Ansible Tower server, using the Job Template referenced by the playbook's task.

NOTE

In recent versions of Ansible, you can encrypt different files with different Ansible Vault passwords. Ansible Tower can use multiple Vault Credentials in the same Job Template to ensure that it can decrypt all files in the project that were encrypted with Ansible Vault.

REFERENCES

Further information is available in the *Ansible Tower API Guide*
<https://docs.ansible.com/ansible-tower/latest/html/towerapi/>

► GUIDED EXERCISE

LAUNCHING JOBS WITH THE ANSIBLE TOWER API

In this exercise, you will launch a job from an existing Job Template by running an Ansible Playbook.

OUTCOMES

You should be able to:

- Use the Ansible Tower API to launch a job from an existing template.

Ensure that the `workstation` and `tower` virtual machines are started.

Log in to `workstation` as `student` using `student` as the password.

On `workstation`, run `lab project-api setup`, which verifies that Ansible Tower services are running and all the required resources are available. It also creates additional Vault credentials and uploads a new playbook into the Git repository.

```
[student@workstation ~]$ lab project-api setup
```

- 1. In the first part of this exercise, you will directly use the REST API provided by Ansible Tower.

On `workstation`, use Firefox to view the resources available from your Ansible Tower server's API.

1. In Firefox, navigate to `https://tower.lab.example.com/api/`. You should see your Ansible Tower server's browsable API. If you are not logged in, log in as `admin` using `redhat` as the password.
2. Click the `/api/v2/` link to access the browsable list of all the available resources accessible through the API.

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
X-API-Node: localhost
X-API-Time: 0.008s

{
    "ping": "/api/v2/ping/",
    "instances": "/api/v2/instances/",
    "instance_groups": "/api/v2/instance_groups/",
    ...output omitted...
```

- 1.3. From the list, click the /api/v2/ping/ link to access that URI. You should see a recent heartbeat time stamp. This URL could be used by an external program to verify that the Ansible Tower server is operating.

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
X-API-Node: localhost
X-API-Time: 0.008s

{
    "instance_groups": [
        {
    ...output omitted...
            "node": "localhost",
            "heartbeat": "2018-11-16T09:24:17Z",
    ...output omitted...
```

- ▶ 2. In Firefox, use the API to launch a job from the existing Demo Job Template Job Template.
- 2.1. Click the Version 2 link at the top of the page to return to the /api/v2/ URI.
 - 2.2. Click the /api/v2/job_templates/ link to access the list of available job templates.
 - 2.3. From the list of results, find the Job Template containing "**name**": "**Demo Job Template**". Find its `url` resource and click the link to access the template. The link should be /api/v2/job_templates/5/ or similar.
 - 2.4. Notice that the Job Template has a related `named_url` resource, which gives you a link with the Job Template name that you can use when calling the API instead of using the Job Template's ID.
On the resulting page, look for the Job Template's `related` resource named **launch**. Click the link associated with that resource. The link should be /api/v2/job_templates/5/launch/ or similar.
This sends a GET request to the `launch` resource for that Job Template, providing information about what information is needed or may be provided to launch a job

from the template. Note that the resource `can_start_without_user_input` is **true**, so you can immediately launch a job without adding information.

- 2.5. To monitor the execution of the job you are about to launch, open another Firefox tab and log in to the Ansible Tower web UI as **admin** using **redhat** as the password. Click **Jobs** in the navigation bar.
- 2.6. Go back to the `https://tower.lab.example.com/api/v2/job_templates/5/launch/` tab. Scroll down to the bottom of this page and click the green POST button to launch the job.
The page immediately refreshes with JSON information about the launched job, including the job's **id** number.
- 2.7. Quickly switch to the tab displaying the **JOBs** page in the Ansible Tower web UI.
The launched job should be at the top of the list of jobs that have been run. You can confirm it is the job you launched by matching its ID with the `id` in the JSON output for the job on the other tab.

- 3. The **curl** command can also use the API to launch jobs from existing Job Templates. Use **curl** to launch a job from the existing **Demo Job Template** Job Template.
- 3.1. To make it easier to read the JSON output provided by **curl**, ensure the **perl-JSON-PP** RPM package is installed on **workstation**.

```
[student@workstation ~]$ sudo yum install perl-JSON-PP
[sudo] password for student:
...output omitted...
Total download size: 55 k
Installed size: 116 k
Is this ok [y/d/N]: y
...output omitted...
```

- 3.2. Use the API with a **name** filter to search for the **Demo Job Template** Job Template. Determine what the Job Template's ID is. This should be the same as the one that you saw using Firefox earlier in this exercise.

```
[student@workstation ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/?name="Demo Job Template" \
> -k -s | json_pp
```

- 3.3. Now that you have confirmed the ID for **Demo Job Template**, use that number and the Job Template's **launch** resource to get information about how to launch the job.

```
[student@workstation ~]$ curl -X GET --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/5/launch/ -k -s \
> | json_pp
{
  "passwords_needed_to_start" : [],
  "ask_limit_on_launch" : false,
  "ask_inventory_on_launch" : false,
  "can_start_without_user_input" : true,
  "defaults" : {
```

```
...output omitted...
```

- 3.4. Again, because you can launch this job without user input, you can issue a POST request to the same URL without any other data to launch the job.

```
[student@workstation ~]$ curl -X POST --user admin:redhat \
> https://tower.lab.example.com/api/v2/job_templates/5/launch/ -k -s \
> | json_pp
{
  ...output omitted...
  "unified_job_template" : 5,
  "network_credential" : null,
  "extra_vars" : "{}",
  "scm_revision" : "",
  "job" : 28,
  "modified" : "2017-05-24T16:34:49.086Z",
  "description" : "",
  "job_args" : "",
  "name" : "Demo Job Template",
  ...output omitted...
  "id" : 28,
  "verbosity" : 0,
  "timeout" : 0,
  "force_handlers" : false,
  "type" : "job",
  "playbook" : "hello_world.yml"
  ...output omitted...
}
```

Note the ID number for the launched job in the JSON output.

- 3.5. Return to the Ansible Tower web UI tab that displays the JOBS page.

The job launched by **curl** should now be at the top of the list of jobs that have been run. Again, you can confirm that it is the job you just launched by comparing the ID in the web UI with the ID in the JSON output from **curl**.

- 4. In the second part of this exercise, you will configure and use the provided **tower_api.yml** playbook. This playbook is encrypted using the **ansible-vault** command with a password of **redhat**. It can be used to remotely launch a new job based on an existing Job Template using the Ansible Tower API. The associated task in the playbook appears as follows:

```
...output omitted...
- name: Launch Job
  uri:
    url: https://{{ tower_host }}/api/v2/job_templates/DEV%20ftpservers
%20setup/launch/
    method: POST
    validate_certs: no
    return_content: yes
    user: "{{ tower_user }}"
    password: "{{ tower_pass }}"
    force_basic_auth: yes
    status_code: 201
  register: this
```

...output omitted...

In the Ansible Tower instance, new credentials called `vault` have been created to provide the correct Vault password for decrypting the new playbook. Associate the new credentials with the new API usage Job Template.

- 4.1. Click Templates in the left navigation bar.
- 4.2. Click the API usage Job Template to edit the settings.
- 4.3. Click the magnifying glass button under CREDENTIAL.
- 4.4. In the CREDENTIALS window, click the CREDENTIAL TYPE list to access the list of available credential types.
- 4.5. Choose the Vault credential type from the list. In the lower pane, select the `vault` credential name, and click SELECT to add this new credential to the API usage Job Template.

Using this new credential in the Job Template, Ansible Tower will be able to decrypt the provided and encrypted `tower_api.yml` playbook to access its own API to start a new job.

- 4.6. Click SAVE at the bottom of the API usage window.
- ▶ 5. Launch a new job based on the modified API usage Job Template.
 - 5.1. Scroll down the page and click the rocket icon in the API usage Job Template line. Observe the output of the playbook. Notice the Print Output task, which shows how the Ansible Tower API was used from the playbook to launch a new job.
 - 5.2. Click the Jobs link in the navigation bar. Notice how a new job called `DEV_ftpservers_setup` has been launched. This new job was launched by the `tower_api.yml` encrypted playbook used by the API usage Job Template.

Cleanup

On workstation, run the `lab project-api cleanup` script to clean up this exercise.

```
[student@workstation ~]$ lab project-api cleanup
```

► LAB

CONSTRUCTING ADVANCED JOB WORKFLOWS

PERFORMANCE CHECKLIST

In this exercise, you will create a new Workflow Job Template that uses a Survey to set variables, and Fact Caching to speed up the workflow.

OUTCOMES

You should be able to create a Workflow Job Template with an associated Survey and Notification Template and then launch a job from the Ansible Tower web UI using the Workflow Job Template.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user with `student` as password on `workstation` and run **`lab project-review setup`**. This script creates all the necessary objects (Git repository, Project, and Job Template) for use with the Prod Inventory.

```
[student@workstation ~]$ lab project-review setup
```

1. Log in to the Ansible Tower web UI running on `tower` as `admin` using `redhat` as the password.
2. Go back to the Ansible Tower web UI.
Enable the Fact Caching option for the `TEST webservers setup` Job Template and `PROD webservers setup` Job Template.
3. Create a Workflow Job Template called `From Test to Prod` with the following information.

FIELD	VALUE
NAME	<code>From Test to Prod</code>
DESCRIPTION	<code>Deploy to Test and on success deploy to Prod</code>
ORGANIZATION	<code>Default</code>
EXTRA VARIABLES	<code>deployment_version: "v1.3"</code>

4. Configure the `From Test to Prod` Workflow Job Template so that it contains the following steps.
 - Synchronize the `My Webservers TEST` Project.
 - Upon success of the previous step, launch the `TEST webservers setup` Job Template.
 - Upon success of the previous step, synchronize the `My Webservers PROD` Project.

- Upon success of the previous step, launch the PROD web servers setup Job Template.
5. Add a Survey containing the following information to the From Test to Prod Workflow Job Template.

FIELD	VALUE
PROMPT	What version are you deploying?
DESCRIPTION	This version number will be displayed at the bottom of the index page.
ANSWER VARIABLE NAME	deployment_version
ANSWER TYPE	Text
MINIMUM LENGTH	1
MAXIMUM LENGTH	40
DEFAULT ANSWER	v1.0
REQUIRED	Selected

6. Activate both the SUCCESS and FAILURE Notifications for the From Test to Prod Workflow Job Template, using the existing Notify on Job Success and Failure Notification Template.
7. Launch a Workflow using the From Test to Prod Workflow Job Template. When prompted by the Survey, enter **v1.3** for the deployment version.
8. Verify that the Workflow triggers an email notification after completion.
9. Verify that the web servers have been updated on `serverc.lab.example.com`, `serverd.lab.example.com`, `servere.lab.example.com`, and `serverf.lab.example.com`.
10. On workstation, run the **lab project-review grade** command to grade your exercise.

► SOLUTION

CONSTRUCTING ADVANCED JOB WORKFLOWS

PERFORMANCE CHECKLIST

In this exercise, you will create a new Workflow Job Template that uses a Survey to set variables, and Fact Caching to speed up the workflow.

OUTCOMES

You should be able to create a Workflow Job Template with an associated Survey and Notification Template and then launch a job from the Ansible Tower web UI using the Workflow Job Template.

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user with `student` as password on `workstation` and run **`lab project-review setup`**. This script creates all the necessary objects (Git repository, Project, and Job Template) for use with the Prod Inventory.

```
[student@workstation ~]$ lab project-review setup
```

1. Log in to the Ansible Tower web UI running on `tower` as `admin` using `redhat` as the password.
2. Go back to the Ansible Tower web UI.
Enable the Fact Caching option for the `TEST webservers setup` Job Template and `PROD webservers setup` Job Template.
 - 2.1. In the navigation bar, click **Templates**.
 - 2.2. Click the `TEST webservers setup` Job Template to edit the Template.
 - 2.3. Select **Use Fact Cache** to enable the Fact Caching option.
 - 2.4. Click **SAVE**.
 - 2.5. Scroll down and click `PROD webservers setup` Job Template to edit the Template.
 - 2.6. Select **Use Fact Cache** to enable the Fact Caching option.
 - 2.7. Click **SAVE**.
3. Create a Workflow Job Template called `From Test to Prod` with the following information.

FIELD	VALUE
NAME	<code>From Test to Prod</code>
DESCRIPTION	<code>Deploy to Test and on success deploy to Prod</code>

FIELD	VALUE
ORGANIZATION	Default
EXTRA VARIABLES	deployment_version: "v1.3"

- 3.1. Click Templates in the navigation bar.
- 3.2. Click the + button to add a new Workflow Job Template.
- 3.3. From the drop-down list, select Workflow Template.
- 3.4. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	From Test to Prod
DESCRIPTION	Deploy to Test and on success deploy to Prod
ORGANIZATION	Default
EXTRA VARIABLES	deployment_version: "v1.3"

- 3.5. Click SAVE to create the new Workflow Job Template.
4. Configure the From Test to Prod Workflow Job Template so that it contains the following steps.
 - Synchronize the My Webservers TEST Project.
 - Upon success of the previous step, launch the TEST webservers setup Job Template.
 - Upon success of the previous step, synchronize the My Webservers PROD Project.
 - Upon success of the previous step, launch the PROD webservers setup Job Template.
- 4.1. Click WORKFLOW VISUALIZER to open the Workflow Visualizer.
- 4.2. Click START to add the first action to be performed. This displays a list of actions to be performed in the right panel.
- 4.3. In the right panel, click PROJECT SYNC to display the list of available Projects. Select My Webservers TEST click SELECT. In the Workflow Visualizer window, this links the START node to the node for the My Webservers TEST Project with a blue line, indicating that this step will always be performed.
- 4.4. Move your mouse over the new node and click the green + button to add an action after the Project Sync of My Webservers TEST. This displays a list of actions to be performed in the right panel.
- 4.5. In the right panel, make sure that you are in the JOBS section. Select the TEST webservers setup Job Template. Depending on the size of your web browser window, you may need to look at the next page to find the Job Template. In the RUN section, select On Success and click SELECT.

The Workflow Visualizer window should show the node for the My Webservers TEST Project linked by a green line to the TEST webservers setup Job Template. This

indicates that if the Project Sync for My Webservers TEST is successful, the TEST webservers setup Job Template will be launched.

- 4.6. Move your mouse over the new node and click the green + button to add an action after the TEST webservers setup Job Template.
 - 4.7. In the right panel, click PROJECT SYNC to display the list of available Projects. Select My Webservers PROD and click SELECT.
 - 4.8. Move your mouse over the new node and click the green + button to add an action after the Project Sync of My Webservers PROD.
 - 4.9. In the right panel, make sure you are in the JOBS section and select the PROD webservers setup Job Template. In the RUN section below, select On Success and click SELECT.
 - 4.10. Click SAVE to save the Workflow Job Template.
- 5.** Add a Survey containing the following information to the From Test to Prod Workflow Job Template.

FIELD	VALUE
PROMPT	What version are you deploying?
DESCRIPTION	This version number will be displayed at the bottom of the index page.
ANSWER VARIABLE NAME	deployment_version
ANSWER TYPE	Text
MINIMUM LENGTH	1
MAXIMUM LENGTH	40
DEFAULT ANSWER	v1.0
REQUIRED	Selected

- 5.1. Click ADD SURVEY to add a Survey.
- 5.2. On the next screen, fill in the details as follows:

FIELD	VALUE
PROMPT	What version are you deploying?
DESCRIPTION	This version number will be displayed at the bottom of the index page.
ANSWER VARIABLE NAME	deployment_version
ANSWER TYPE	Text
MINIMUM LENGTH	1
MAXIMUM LENGTH	40

FIELD	VALUE
DEFAULT ANSWER	v1.0
REQUIRED	Selected

- 5.3. Click ADD to add the Survey Prompt to the Survey. This displays a preview of your Survey on the right.



IMPORTANT

Before saving, make sure that the ON/OFF switch is set to **ON** at the top of the Survey editor window.

- 5.4. Click SAVE to add the Survey to the Workflow Job Template.
6. Activate both the SUCCESS and FAILURE Notifications for the From Test to Prod Workflow Job Template, using the existing Notify on Job Success and Failure Notification Template.
- 6.1. Click NOTIFICATIONS to manage notifications for the From Test to Prod Workflow Job Template.
 - 6.2. On the same line as the Notify on Job Success and Failure Notification Template, set both ON/OFF switches for **SUCCESS** and **FAILURE** to **ON**.
7. Launch a Workflow using the From Test to Prod Workflow Job Template. When prompted by the Survey, enter **v1.3** for the deployment version.
- 7.1. Scroll down to the TEMPLATES section.
 - 7.2. On the same line as the From Test to Prod Workflow Job Template, click the rocket icon on the right to launch the Workflow. This opens the Survey you just created and prompts for input.
 - 7.3. Enter **v1.3** in the text field, click NEXT and then click LAUNCH to launch the Workflow. This redirects you to a detailed status page of the running Workflow.
 - 7.4. Observe the running Jobs of the Workflow. You can click the DETAILS link of a running or completed Job to see a more detailed live output of the Job.
8. Verify that the Workflow triggers an email notification after completion.
- 8.1. Open a terminal and connect to the tower VM.

```
[student@workstation ~]$ ssh tower
Last login: Thu Apr 20 11:33:22 2017 from workstation.lab.example.com
[student@tower ~]$
```

- 8.2. Use the **tail** command to view incoming messages to the local mailbox file of the student user. You should see this type of successful Workflow Job completion notification email arrive:

```
[student@tower ~]$ tail -f /var/mail/student
...output omitted...
From system@tower.lab.example.com Thu Apr 20 18:06:10 2017
Return-Path: <system@tower.lab.example.com>
```

```
X-Original-To: student@tower.lab.example.com
Delivered-To: student@tower.lab.example.com
Received: from tower.lab.example.com (localhost [IPv6:::1])
    by tower.lab.example.com (Postfix) with ESMTP id AAB30401FB3
    for <student@tower.lab.example.com>; Thu, 20 Apr 2017 18:06:10 -0400 (EDT)
MIME-Version: 1.0
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: 7bit
Subject: Workflow Job #50 'From Test to Prod' succeeded on Ansible Tower:
    https://tower.lab.example.com/#/workflows/50
From: system@tower.lab.example.com
To: student@tower.lab.example.com
Date: Thu, 20 Apr 2017 22:06:10 -0000
Message-ID: <20170420220610.2290.67752@tower.lab.example.com>

Workflow job summary:

- node #9 spawns job #51, "My Webservers TEST", which finished with status
  successful.
- node #10 spawns job #52, "TEST webservers setup", which finished with status
  successful.
- node #11 spawns job #54, "My Webservers PROD", which finished with status
  successful.
- node #12 spawns job #55, "PROD webservers setup", which finished with status
  successful.
```

8.3. Exit the console session on the `tower` system.

```
[student@tower ~]$ exit
```

9. Verify that the web servers have been updated on `serverc.lab.example.com`, `serverd.lab.example.com`, `servere.lab.example.com`, and `serverf.lab.example.com`.
 - 9.1. Open a web browser and navigate to `http://serverc.lab.example.com`, `http://serverd.lab.example.com`, `http://servere.lab.example.com`, and `http://serverf.lab.example.com` in separate tabs. You should see this line at the bottom of each page:

```
Deployment Version: v1.3
```

9.2. When ready, log out from the Ansible Tower web UI.

10. On workstation, run the `lab project-review grade` command to grade your exercise.

SUMMARY

In this chapter, you learned:

- Enabling fact caching can speed up job execution but requires management of fact gathering.
- You can add Surveys to a Job Template to enable single-step automation, by prompting users for the values of extra variables used by the playbook.
- Workflow Job Templates can launch several Job Templates in sequence, and can launch different Job Templates depending on whether the previous one succeeded or failed.
- You can configure Job Templates to launch jobs on a one-time or recurring schedule.
- You can use Notification Templates to send notifications when jobs succeed or fail. Red Hat Ansible Tower supports many different notification mechanisms.
- Ansible Tower provides a browsable REST API, which can easily be used to automate Ansible Tower operations and integrate it with third-party products.

CHAPTER 15

MANAGING ADVANCED INVENTORIES

GOAL

Manage inventories that are loaded from external files or generated dynamically from scripts or the Ansible Tower smart inventory feature.

OBJECTIVES

- Import a static inventory into Ansible Tower from an external file, and use static inventories managed in a Git repository.
- Create a dynamic inventory that uses a custom inventory script to set hosts and host groups.
- Create a smart inventory that is dynamically constructed from the other inventories on your Ansible Tower server using a filter.

SECTIONS

- Importing External Static Inventories (and Guided Exercise)
- Creating and Updating Dynamic Inventories (and Guided Exercise)
- Filtering Hosts with Smart Inventories (and Guided Exercise)

LAB

Managing Advanced Inventories

IMPORTING EXTERNAL STATIC INVENTORIES

OBJECTIVES

After completing this section, students should be able to import a static inventory into Ansible Tower from an external file, and use static inventories managed in a Git repository.

IMPORTING EXISTING STATIC INVENTORIES

If you are adding an existing Ansible project that you have been managing from a traditional control node into Red Hat Ansible Tower, you might already have a large static inventory file for that project. It can be inconvenient to add that static inventory manually through the web UI. You might also want to continue to manage that static inventory outside of Ansible Tower rather than through the web UI. There are a number of ways to handle these situations. In this section, you will learn about two of them.

The first solution is to import the static inventory into Ansible Tower so that it can be managed through the web UI. This has the advantage that after you do this, Ansible Tower users can manage the inventory normally through the Ansible Tower interface.

The second solution is to configure Ansible Tower to retrieve the static inventory file from a source control project. In this scenario, the inventory could be stored in a Git repository like a playbook, and is periodically synchronized to Ansible Tower like a Project update. The advantage of this is that you can continue to use a version control system to manage changes to your inventory file and inventory variables, just like you manage your playbooks. The disadvantage is that you will not be able to make changes to the inventory's contents through the Ansible Tower web UI.

UPLOADING A STATIC INVENTORY

Ansible Tower comes with the **awx-manage** command-line utility, which can be used to access detailed internal Ansible Tower information. The **awx-manage** command must be run as `root` or as the `awx` (Ansible Tower) user.

This utility is most commonly used to import an existing static inventory from a file directly into the Ansible Tower server. The variables set in the `group_vars` or `host_vars` directories that are associated with the inventory file will also be imported with the inventory file.

To use this feature of the **awx-manage** command, you must first set up a destination inventory in Ansible Tower. Ansible Tower imports the inventory file and associated inventory variables into this destination inventory.

Before importing your static inventory file and its host and group variables, you should organize the files you plan to import into a directory structure like the following example:

```
[root@towerhost ~]# tree inventory/
inventory/
|-- group_vars
|   '-- mygroup
|-- host_vars
|   '-- myhost
`-- hosts
```

Run **awx-manage inventory_import** to import these inventory hosts and variables into Ansible Tower. Specify the source directory containing your inventory files with the **--source** option, and the name of the existing destination inventory in Ansible Tower with the **--inventory-name** option.

```
[root@towerhost ~]# awx-manage inventory_import --source=inventory/ \
> --inventory-name="My Tower Inventory"
```

If your inventory is simply a single flat file, the **--source** option can point directly at the inventory file itself rather than to a directory:

```
[root@towerhost ~]# awx-manage inventory_import --source=./my_inventory_file \
> --inventory-name="My Tower Inventory"
```

If the destination inventory in Ansible Tower is not empty, then imported data does not overwrite the existing data by default, but is combined with it. This default behavior adds any new variables from the imported inventory to any variable already in the imported inventory. You can overwrite any existing data by specifying the **--overwrite_vars** option.

```
[root@towerhost ~]# awx-manage inventory_import --source=inventory/ \
> --inventory-name="My Tower Inventory" \
> --overwrite
```



IMPORTANT

Remember, if the destination inventory is not empty, by default **awx-manage inventory_import** combines the existing information in that inventory in Ansible Tower with the data that you're uploading. This could be really convenient, or make a big mess if you do it by accident.

STORING AN INVENTORY IN A PROJECT

Red Hat Ansible Tower can use inventory files that you manage in a source code management (SCM) repository. This allows you to continue to store your inventory in a Git repository, and use commits, pull requests, and other features of Git or your repository server to manage updates. You can also use any other SCM type supported by Projects in Ansible Tower.

To configure this, you start by setting up a Project that points to your Git repository. You do this in exactly the same way as a Project that you're setting up to be part of a Job Template. Specify the **SCM Type** (such as Git, Mercurial, or SVN) and **SCM URL** of the repository containing the inventory file or files. You might also need to specify the **SCM Credential** that contains the authentication information for that repository. You can specify a particular branch, tag, or commit to use, and whether the contents of the Project should **Update Revision on Launch** just like any other Project.

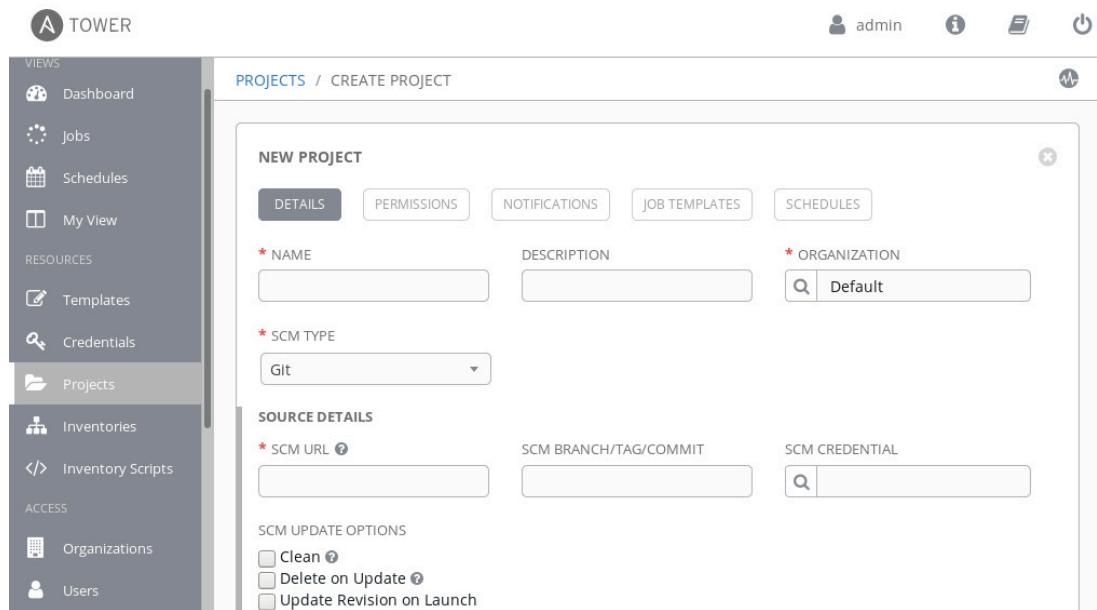


Figure 15.1: Configuring Git-based projects

Once you have set up the Project, you can configure the inventory in Ansible Tower. Create the inventory normally and open it for editing. Click on the SOURCES button, and then click the + button to add a new source.

On the CREATE SOURCE page, give the source a name and then under SOURCE select Sourced from a Project. This will add a few fields to the page. Specify the new PROJECT that contains your inventory. Select the inventory file from the INVENTORY FILE combo box, typing in the name of the file if it does not appear on the combo box's drop-down list.

You can enter an SCM credential in the Credential field if one is needed to access the Project repository. You can also check the Update on Project Change check box to refresh the inventory after every project update that involves a Git revision update.

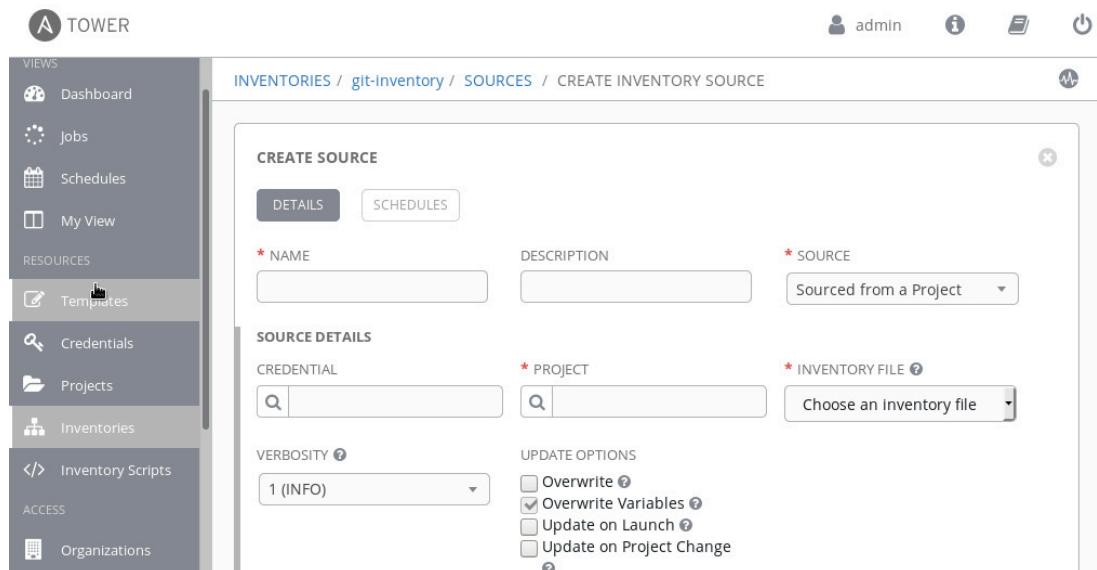


Figure 15.2: Inventory sourced from a project



REFERENCES

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide/>

Ansible Tower Administration Guide

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

IMPORTING EXTERNAL STATIC INVENTORIES

In this exercise, you will import an existing static inventory file into Ansible Tower, and you will also use an existing static inventory file stored in a Git repository.

OUTCOMES

- Use the **awx-manage** utility to import an existing inventory file.
- Use an existing static inventory file stored in a Git repository.

BEFORE YOU BEGIN

Ensure that the **workstation** and **tower** virtual machines are started.

Log in to **workstation** as **student** using **student** as the password.

From **workstation**, run **lab inventory-static setup**, to verify that Ansible Tower services are running and all required resources are available.

```
[student@workstation ~]$ lab inventory-static setup
```

- 1. Import the static inventory file **/root/example-inventory** into Ansible Tower with the **awx-manage** command.

- 1.1. On **workstation**, open a terminal. Use **ssh** to log in to the **tower** server as **root**.

```
[student@workstation ~]$ ssh root@tower
[root@tower ~]#
```

- 1.2. On **tower**, list the contents of the **root** user home directory, to confirm that the static inventory file for Ansible Tower is available.

```
[root@tower ~]# ls /root
...output omitted...
example-inventory
...output omitted...
```

- 1.3. Import the **example-inventory** static inventory file using the command **awx-manage inventory_import**. Use the existing inventory named **Exercise** as the destination for the import. After the import has completed, log out of the **tower** system.

```
[root@tower ~]# awx-manage inventory_import \
> --source=/root/example-inventory \
> --inventory-name="Exercise"
2.467 INFO      Updating inventory 5: Exercise
```

```

2.623 INFO    Reading Ansible inventory source: /root/example-inventory
3.960 INFO    Processing JSON output...
3.961 INFO    Loaded 2 groups, 5 hosts
4.192 INFO    Inventory import completed for (Exercise - 11) in 1.7s
[root@tower ~]# exit

```

- 1.4. In the Ansible Tower web interface, click Inventories in the left navigation bar to display the list of Inventories. You should see an Inventory named **Exercise**, which was used in a previous step for the import.
 - 1.5. Click the Exercise link to view the details of the imported static Inventory. Look at the available GROUPS and HOSTS sections. You should see that the inventory is composed of multiple groups and hosts, confirming that the static inventory has been successfully imported and is accessible.
- ▶ 2. Create a new project named **MyProjectGit** using the Git repository located at `git.lab.example.com/home/git/inventory.git`.
- 2.1. Click Projects in the left navigation bar.
 - 2.2. Click **+** to add a new Project.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	MyProjectGit
DESCRIPTION	Project Based on Git
ORGANIZATION	Default
SCM TYPE	Git
SCM URL	<code>ssh://git@git.lab.example.com/home/git/inventory.git</code>
SCM CREDENTIAL	student-git

- 2.4. Check the Update Revision on Launch checkbox.
 - 2.5. Click **SAVE** to add the inventory.
- ▶ 3. Create a new inventory named **ExerciseGit** with the **git-inventory** file available through the **MyProjectGit** project.
- 3.1. Click Inventories in the left navigation bar.
 - 3.2. Click **+**, and select **Inventory** to add a new Inventory.
 - 3.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	ExerciseGit
DESCRIPTION	Static Inventory from Git

FIELD	VALUE
ORGANIZATION	Default

- 3.4. Click **SAVE** to add the inventory.
- 3.5. Go to the **SOURCES** section, and click **+** to add a new source.
- 3.6. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	git-inventory
DESCRIPTION	Static Inventory from Git
SOURCE	Sourced from a Project
PROJECT	MyProjectGit
INVENTORY FILE	git-inventory



NOTE

Make sure you enter manually **git-inventory** in the INVENTORY FILE field. The file may not display in the drop-down menu because of a known product issue.

- 3.7. Select the **Update on Project Change** checkbox.
 - 3.8. Click **SAVE** to add the inventory.
 - 3.9. Scroll down to verify that the cloud icon next to **git-inventory** is static and green.
 - 3.10. Verify that the **filesrv** host group is available in the **GROUPS** section, and the **serverf.lab.example.com** host is available under the **HOSTS** section.
- 4. Add the **serverf.lab.example.com** host to the **filesrv** host group in the Git-based inventory.
- 4.1. On **workstation**, clone the **git.lab.example.com/home/git/inventory.git** Git repository. When done, go to the **inventory** directory.

```
[student@workstation ~]$ git clone \
> ssh://git@git.lab.example.com/home/git/inventory.git
[student@workstation ~]$ cd inventory
```

- 4.2. Add the **serverf.lab.example.com** host to the **filesrv** host group in the **git-inventory** file. When done, commit the changes.

```
[student@workstation inventory]$ vi git-inventory
[filesrv]
servere.lab.example.com
serverf.lab.example.com
[student@workstation inventory]$ git commit -m "adding serverf" git-inventory
```

```
[student@workstation inventory]$ git push
```

- 4.3. In the Ansible Tower web interface, click Projects in the left navigation bar
- 4.4. Click the double-arrow icon in the row for MyProjectGit to get the latest version of the Git-based inventory file. Wait until the dot icon next to MyProjectGit is static.
- 4.5. Click Inventories in the left navigation bar
- 4.6. Click on the ExerciseGit link and go to SOURCES.
- 4.7. Click the double-arrow icon in the row for the git-inventory source to retrieve the changes. Wait until the cloud icon next to git-inventory is static and green.
- 4.8. Verify that the serverf.lab.example.com host is available under the HOSTS section.
- 4.9. Click the Log Out icon to log out of the Tower web interface.

Cleanup

From workstation, run the **lab inventory-static cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab inventory-static cleanup
```

This concludes the guided exercise.

CREATING AND UPDATING DYNAMIC INVENTORIES

OBJECTIVES

After completing this section, students should be able to create a dynamic inventory that uses a custom inventory script to set hosts and host groups.

Figure 15.2: Creating and Updating Dynamic Inventories

DYNAMIC INVENTORIES

When Ansible runs a playbook, it uses an inventory to help determine against which hosts the plays should be run. Both Ansible and Ansible Tower make it easy to set up a static inventory of hosts which are explicitly specified in the inventory by the administrator.

However, these static lists require manual administration to keep them up to date. This can be inconvenient or challenging, especially when an organization wants to run the playbooks against hosts which are dynamically created in a virtualization or cloud computing environment.

In that scenario, it is useful to use a *dynamic inventory*. Dynamic inventories are scripts that, when run, dynamically determine which hosts and host groups should be in the inventory based on information from some external source. These can include the API for cloud providers, Cobbler, LDAP directories, or other third party CMDB software. Using a dynamic inventory is a recommended practice in a large and fast-changing IT environment, where systems are frequently deployed, tested and then removed.

By default, Ansible Tower comes with built-in dynamic inventory support for a number of external inventory sources (or cloud inventory sources), including:

- Amazon Web Services EC2
- Google Compute Engine
- Microsoft Azure Resource Manager
- VMware vCenter
- Red Hat Satellite 6
- Red Hat CloudForms
- Red Hat Virtualization
- OpenStack

In addition, it is possible to use custom dynamic inventory scripts in Ansible Tower to access inventory information from other sources. Ansible Tower also allows retrieving those scripts from a project which uses a repository like Git as a source.

The remainder of this section looks at three examples of dynamic inventory configuration in Ansible Tower. The first looks at the built-in support for OpenStack. The second briefly examines the built-in support for getting information from a Red Hat Satellite 6 server. The third investigates how custom dynamic inventory scripts may be used.

OPENSTACK DYNAMIC INVENTORIES

Cloud technologies like OpenStack bring many changes to the server lifecycle. Hosts come and go over time, and they can be created and started by external applications. Maintaining an accurate static inventory file is challenging over any length of time. Therefore, having the inventory update dynamically based on information provided directly from OpenStack is very helpful.

The basic process for configuring dynamic inventories using any of the built-in cloud sources is similar for each of them:

1. Create a credential to authenticate to the cloud data source you intend to use, using a credential type matching the source
2. Create a new inventory to provide dynamic inventory information.
3. In that new inventory, create a Source with one of the built-in dynamic inventory sources (instead of Manual). It should also use the new Credential to authenticate to that source. You may also want to set other options, like having it automatically Update on Launch.
4. Update the source in the inventory for the first time.

This process works for OpenStack dynamic inventories. This is an example of how to create credentials for use by an OpenStack dynamic inventory:

The screenshot shows the Ansible Tower interface. On the left is a sidebar with navigation links: Views, Dashboard, Jobs, Schedules, My View, Resources, Templates, Credentials (which is selected), Projects, Inventories, Inventory Scripts, Access, and Organizations. The main area has a header 'CREDENTIALS / CREATE CREDENTIAL'. Below it is a 'NEW CREDENTIAL' form. The 'DETAILS' tab is active. It contains fields for 'NAME' (with a placeholder 'OpenStack'), 'DESCRIPTION', and 'ORGANIZATION' (with a placeholder 'SELECT AN ORGANIZATION'). The 'PERMISSIONS' tab is also visible. Below the tabs is a section titled 'TYPE DETAILS' with fields for 'USERNAME', 'PASSWORD (API KEY)' (with a 'SHOW' button), 'HOST (AUTHENTICATION URL)', 'PROJECT (TENANT NAME)', and 'DOMAIN NAME'.

Figure 15.3: Cloud credentials creation

As you can see in this screenshot, there are a number of items you need to provide. You have used some of these with other objects in Ansible Tower: Name, Description and Organization. The only new item is the Credential Type. You have to choose the appropriate one for the product you are using. In this example, it is OpenStack.

An OpenStack Credential needs some additional information:

Username

The user who can access the required resources

Password

For that user, or API key

Host

The authentication URL of the host to authenticate with, for example `https://demo.lab.example.com/v2.0/`

Project

The name of the project (tenant) you want to use

Domain Name

Needed only with Keystone v3, defines administrative boundaries

Those newly created credentials are going to be used by the Ansible Tower Inventory synchronization mechanism. After creating the credentials, you can switch to the Inventories link in the left navigation bar. As you can see in the example below, you need to create a new Inventory:

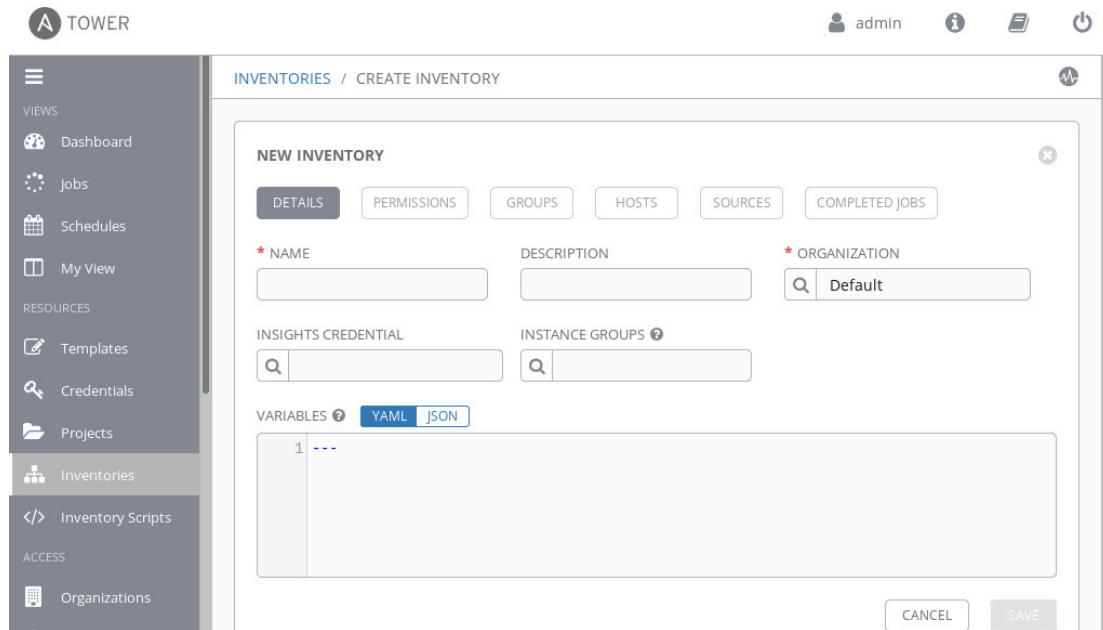


Figure 15.4: Cloud inventory creation

That new inventory needs a unique NAME and you need to assign it to an existing ORGANIZATION. When the new inventory configuration is saved, go to SOURCES within the inventory to create a new source for the inventory. Ansible Tower uses that source in conjunction with the existing OpenStack scripts and the previously created credentials. The screenshot below shows an example of such a source:

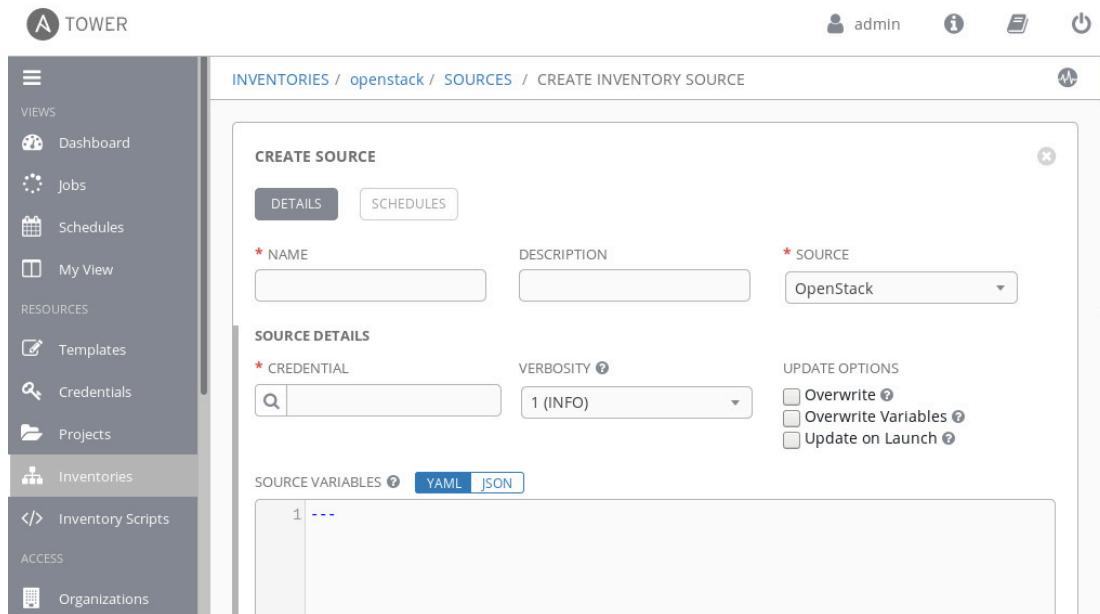


Figure 15.5: Cloud group creation

This new source uses the built-in Ansible Tower support for OpenStack as SOURCE and the new credentials for your OpenStack environment as CREDENTIAL. There are three UPDATE OPTIONS available to choose:

Overwrite

When this option is activated, the inventory update process deletes all child groups and hosts from the local inventory, not found on the external source. By default it is not active, it means that all child hosts and groups not found on the external source remain untouched.

Overwrite Variables

When not activated, a merge combining local variables with those found on the external source, is performed. Otherwise, when activated all variables not found on the external source are removed.

Update on Launch

When activated, each time a job runs using this inventory, a refresh of this inventory from the external source is performed, before the job tasks are executed.

In the source list for the inventory, the small cloud icon to the left of the source name shows the status of the dynamic inventory synchronization for that source. When it is gray, no status is available. To start the synchronization process you can click the two-arrows icon. When the synchronization finishes, the status of the cloud icon changes to green if synchronization has been successful, or red if it failed.

After a successful synchronization with the external source, you can review the child groups and hosts which have been created in Tower using the information from the external source. The child groups contain the hosts visible on the HOSTS lists. You can review each child group by clicking on the group name. This displays the content of that group, giving you a list of the hosts associated with that group. This inventory is updated every time you synchronize it with the external source. That action can be performed manually, scheduled using the Tower mechanisms or performed automatically each time a job runs using that inventory.

RED HAT SATELLITE 6 DYNAMIC INVENTORIES

Another example of a built-in dynamic inventory uses information about hosts that are registered with a Red Hat Satellite 6 server. Workflow for deployment and configuration of a new bare-metal server using Red Hat Satellite 6 in conjunction with Ansible Tower might look like this:

1. The new server uses some combination of PXE, DHCP, and TFTP to boot from the network or a boot ISO to prepare for either an unattended installation from the Satellite server or installation through Satellite's Discovery service.
2. The new server performs a Kickstart installation from materials provided by the Satellite server and registers itself to the Satellite server.
3. The new server appears in the dynamic inventory generated from Red Hat Satellite information once it's registered. Ansible Tower can now be used to launch various jobs using that inventory to ensure that the new server is provisioned correctly.



NOTE

The *Provisioning Callbacks* feature of Ansible Tower is particularly useful for triggering initial provisioning jobs when a new server is deployed. More information on this feature is available in the documentation at http://docs.ansible.com/ansible-tower/latest/html/userguide/job_templates.html#provisioning-callbacks/.

Configuration of an Ansible Tower dynamic inventory using Red Hat Satellite 6 is very similar to the OpenStack scenario.

The first step is to create a new credential. The Type of the credential in this case will be Red Hat Satellite 6. It requires three pieces of additional information:

Satellite 6 URL

The URL for the Satellite server, such as `https://satellite.example.com`

Username

For a user on the Satellite server

Password

Password for the Satellite user

Next, create a source in an inventory to synchronize inventory data from the Satellite server. The source should be set to Red Hat Satellite 6. The source's credential should be set to the credential you just created for Satellite. Just like the OpenStack dynamic inventory configuration, the three UPDATE OPTIONS (Overwrite, Overwrite Variables, and Update on Launch) may be selected as desired.

The final step is to synchronize the source with the Red Hat Satellite inventory source, in exactly the same way as discussed in the section on the OpenStack inventory source. When the synchronization finishes, all information gathered from the external source are visible in Ansible Tower web interface as groups in the inventory, and hosts associated with those groups.

CUSTOM DYNAMIC INVENTORY SCRIPTS

Ansible allows you to write custom scripts to generate a dynamic inventory. While Ansible Tower offers built-in support for a number of dynamic inventory sources, custom dynamic inventory scripts can still be used with Ansible Tower.

Writing or Obtaining Custom Inventory Scripts

Ansible Tower supports custom inventory scripts written in any dynamic language installed on the Ansible Tower server. This should include Python and Bash at a minimum. These scripts run as the awx user and have limited access to the Tower server. The script must start with an appropriate shebang line (for example, `#!/usr/bin/python` for a Python script).

Many examples of custom inventory scripts for use with various external sources have been contributed by the community the Git repository for Ansible at <https://github.com/ansible/ansible/tree/devel/contrib/inventory/>.

If you want to write your own custom inventory script, information is available at [Developing Dynamic Inventory Sources \[http://docs.ansible.com/ansible/dev_guide/developing_inventory.html\]](http://docs.ansible.com/ansible/dev_guide/developing_inventory.html) in the *Ansible Developer Guide*. When the dynamic inventory script is called with the `--list` option, it must output the inventory in JSON format.

This is example output from a custom dynamic inventory script:

```
{
  "databases" : {
    "vars" : {
      "example_db" : true
    },
    "hosts" : [
      "db1.demo.example.com",
      "db2.demo.example.com"
    ]
  },
  "webservers" : [
    "web1.demo.example.com",
    "web2.demo.example.com"
  ],
  "boston" : {
    "children" : [
      "backup",
      "ipa"
    ],
    "vars" : {
      "example_host" : false
    },
    "hosts" : [
      "server1.demo.example.com",
      "server2.demo.example.com",
      "server3.demo.example.com"
    ]
  },
  "backup" : [
    "server4.demo.example.com"
  ],
  "ipa" : [
    "server5.demo.example.com"
  ]
}
```

As you can see in the preceding example, each group may contain a list of hosts, potential child groups, possible group variables or a list of hosts.

**NOTE**

When called with the option `--host hostname`, the script must print a JSON hash/dictionary of the variables for the specified host (potentially an empty JSON hash or dictionary if there are no variables provided).

Optionally, if the `--list` option returns a top-level element called `_meta`, it is possible to return all host variables in one script call, which improves script performance. In that case, `--host` calls will not be made.

See the previously referenced documentation in the Developing Dynamic Inventory Sources section of the Ansible Developer Guide for more information.

Using Custom Inventory Scripts in Ansible Tower

When you have created or downloaded the appropriate custom inventory script, you need to import it into Ansible Tower and configure the inventory. Below is a procedure how to accomplish this task:

To upload a custom inventory script into Tower, go to Inventories Scripts in the left navigation bar on the Ansible Tower web interface. Click the + button to add a new custom inventory script.

The screenshot shows the 'Inventory Scripts / CREATE INVENTORY SCRIPT' dialog in Ansible Tower. On the left is a sidebar with 'Inventories Scripts' selected. The main dialog has a title 'NEW CUSTOM INVENTORY'. It contains four fields: 'NAME' (with a required asterisk), 'DESCRIPTION', 'ORGANIZATION' (set to 'Default'), and 'CUSTOM SCRIPT' (a large text area). At the bottom are 'CANCEL' and 'SAVE' buttons.

Figure 15.6: Ansible Tower Inventory Script

Define a new name for the custom inventory script in the NAME field, select an organization in the ORGANIZATION field, and copy and paste the actual script into the CUSTOM SCRIPT text box. Click the SAVE button.

Once you have configured the dynamic inventory script in Ansible Tower, configure it just like any of the built-in dynamic inventories:

1. Create a new source in an inventory for the dynamic inventory. Set its SOURCE to Custom Script, and CUSTOM INVENTORY SCRIPT to the name you set for the custom script that you just imported into Ansible Tower.
2. Synchronize the source with the inventory source, as discussed for the OpenStack and Red Hat Satellite 6 dynamic inventory sources.

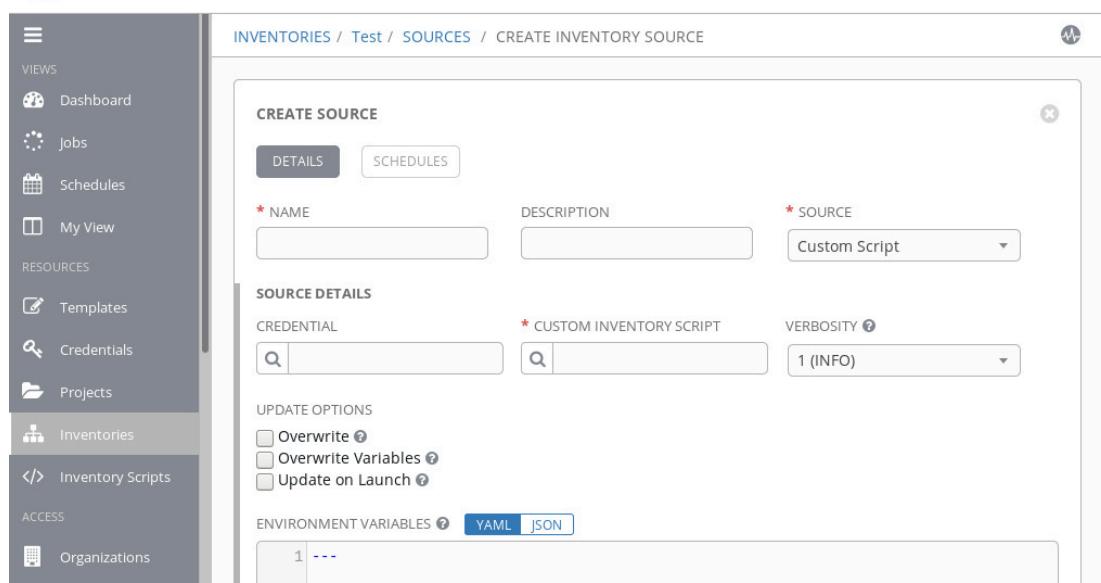


Figure 15.7: Adding new Inventory Group for custom dynamic inventory



REFERENCES

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide/>

Ansible Tower Administration Guide

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

CREATING AND UPDATING DYNAMIC INVENTORIES

In this exercise, you will add a custom inventory script and use it to manage a Dynamic Inventory managed on an IdM server.

OUTCOMES

You should be able to add a custom inventory script and use it to populate a Dynamic Inventory in Tower.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run `lab inventory-dynamic setup`. This setup script prepares the IdM server for this exercise.

```
[student@workstation ~]$ lab inventory-dynamic setup
```

- ▶ 1. Log in to the Ansible Tower web interface running on the `tower` system using the `admin` account and the `redhat` password.
- ▶ 2. Add the `ldap-freeipa.py` custom inventory script to Tower.
 - 2.1. Click Inventory Scripts in the left navigation bar.
 - 2.2. Click the + button to add a new custom inventory script.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	<code>ldap-freeipa.py</code>
DESCRIPTION	Dynamic Inventory for IdM Server
ORGANIZATION	Default

- 2.4. Copy the contents of the `ldap-freeipa.py` script located at `http://materials.example.com/classroom/ansible/ipa-setup/ldap-freeipa.py` into the CUSTOM SCRIPT field.
- 2.5. Click SAVE to add the custom inventory script.
- ▶ 3. Create a new Inventory called Dynamic Inventory.
 - 3.1. Click Inventories in the left navigation bar.

- 3.2. Click the + button to add a new inventory, and select Inventory in the drop-down menu.
- 3.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Dynamic Inventory
DESCRIPTION	Dynamic Inventory from IPA server
ORGANIZATION	Default

- 3.4. Click SAVE to create the Inventory.
 - ▶ 4. Add the **ldap-freeipa.py** script as a new source for the Inventory.
 - 4.1. Click SOURCES in the top bar for the Inventory details pane.
 - 4.2. Click the + button to add a new source.
 - 4.3. On the next screen, fill in the details as follows:
- | FIELD | VALUE |
|-------------------------|-------------------------------------|
| NAME | Custom Script |
| DESCRIPTION | Custom Script for Dynamic Inventory |
| SOURCE | Custom Script |
| CUSTOM INVENTORY SCRIPT | ldap-freeipa.py |
- 4.4. Under the UPDATE OPTIONS section, check the checkbox next to the Overwrite option.
 - 4.5. Click SAVE to create the Source.
 - ▶ 5. Update the Dynamic Inventory.
 - 5.1. Scroll down to the lower pane, and click the double-arrow button in the row for **Custom Script**, and wait until the cloud becomes green, and static.
 - 5.2. Click GROUPS, and observe that this inventory now contains four Groups: **development**, **ipaservers**, **production**, and **testing**. Each of these groups contains hosts.

This concludes the guided exercise.

FILTERING HOSTS WITH SMART INVENTORIES

OBJECTIVES

After completing this section, students should be able to create a smart inventory that is dynamically constructed from the other inventories on your Ansible Tower server using a filter.

CONFIGURING SMART INVENTORIES

So far, you have learned several ways to manage static and dynamic inventories in Red Hat Ansible Tower:

- You can manually create a static inventory in the web UI
- You can import a static inventory file into Ansible Tower and then manage it in the web UI
- You can configure Ansible Tower to use a Project to get the inventory from files stored in version control and manage it in the version control system
- You can configure a dynamic inventory to get host information from an external service or by using a custom inventory script

Red Hat Ansible Tower 3.2 added a way to dynamically construct a new inventory from inventories already existing in Ansible Tower. A *smart inventory* generates its information by applying a *host filter* to the union of all static and dynamic inventories configured on the Ansible Tower server. This host filter usually checks if a particular Ansible fact has a particular value for each host. Hosts that match the host filter are included in the smart inventory. This provides more flexibility to manage a subset of the hosts defined by the static and dynamic inventories.

Smart inventories use Ansible Tower's fact cache to apply the smart host filter. This means that you need to periodically populate the fact cache with a Job Template that is configured with the Use Fact Cache checkbox selected and that gathers facts. You can do this by running a normal playbook that has `gather_facts: yes` enabled (normally set implicitly by default) or that runs the `setup` module as a task. A minimal playbook to do this could read:

```
- name: Refresh fact cache
  hosts: all
  gather_facts: yes
```

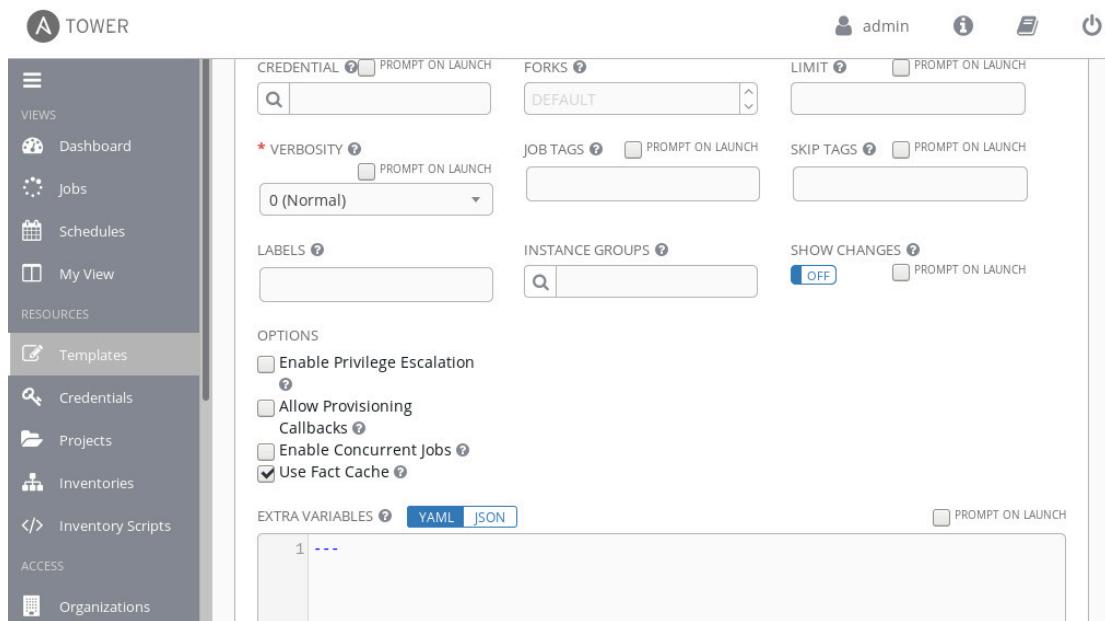


Figure 15.8: Enabling "Use Fact Cache" in a job template

To create a smart inventory, go to Inventories in the left navigation bar of the Ansible Tower Web UI. Click the + button, and select Smart Inventory to open the New Smart Inventory page. In that page, you need to specify at least a name for the smart inventory, assign it to an organization, and specify the smart host filter for the smart inventory.

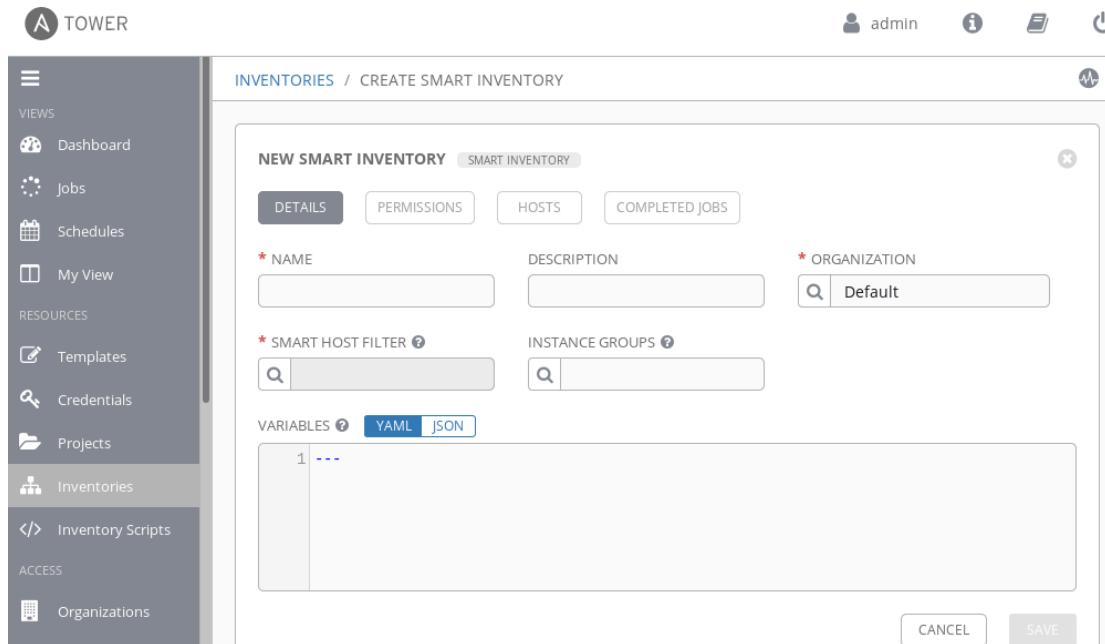


Figure 15.9: New Smart Inventory page

Defining Smart Host Filters

To define the host filter for the smart inventory, click the magnifying glass icon next to the SMART HOST FILTER field on the New Smart Inventory page. If no smart filter is set, all hosts are part of the smart inventory.

When you click on the icon, a new window opens titled DYNAMIC HOSTS. Enter your host filter or filters in the search field, and click on the magnifying glass next to the field to apply the filter. The hosts matching the filter will be displayed at the bottom of the window.

The syntax for defining a host filter based on an Ansible fact can be a bit confusing. The filter should start with the string **ansible_facts** and then is followed by the name of the Ansible fact in the old format (fact injected as a variable name), a colon, and then the value you want to match exactly. There must be no space after the colon and before the value.

For example, to match hosts that have the value **RedHat** for the fact `ansible_distribution`, you would use the host filter **ansible_facts.ansible_distribution:RedHat**.

You can also create host filters based on group membership or by host name and host description instead of using facts. For more information, see "host_filter Search" [<https://docs.ansible.com/ansible-tower/latest/html/userguide/inventories.html#host-filter-search>] in the *Ansible Tower User Guide*.



IMPORTANT

The current syntax for host filters is potentially confusing, because it looks like it is referring to a fact namespaced under `ansible_facts`, but it is *not* actually doing that. The **ansible_facts** string in the host filter indicates that you want to match a fact, not a host name or something else.

Until Ansible 2.5, all facts were "injected as variables" into the same namespace as other variables. Starting with Ansible 2.5, they were instead defined under the `ansible_facts` variable. For example, the old `ansible_distribution` fact is now officially referred to by the name `ansible_facts.distribution`. For backward compatibility, the "injected" names are still available if Ansible has the configuration setting `inject_facts_as_vars: true` set. This is still the default setting in Ansible 2.7.

But host filters for smart inventories do not support the modern naming for facts yet, and worse yet *look* like a modern fact name in the filter. So you must say `ansible_facts.ansible_distribution` and `not ansible_facts.ansible_facts.distribution` (or even `ansible_facts.distribution`).

The other challenge with host filter syntax is that there must be no whitespace between the colon and the value that you want to match. This is easy to forget and somewhat non-intuitive.

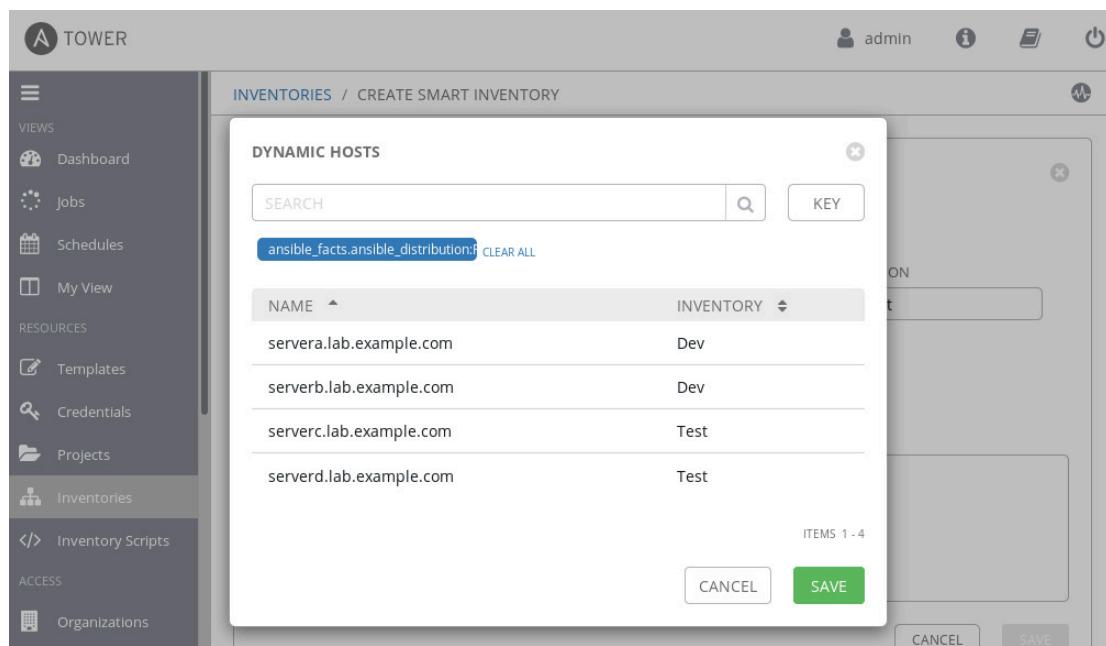


Figure 15.10: Configuring a smart host filter



REFERENCES

Ansible Tower User Guide

<https://docs.ansible.com/ansible-tower/latest/html/userguide/>

► GUIDED EXERCISE

FILTERING HOSTS WITH SMART INVENTORIES

In this exercise, you will create a smart inventory and explore how smart inventories work.

OUTCOMES

You should be able to create a smart inventory and automatically add more hosts to it.

BEFORE YOU BEGIN

Ensure that the workstation and tower virtual machines are started.

Log in to workstation as student using student as the password.

From workstation, run **lab inventory-smart setup**, which verifies that Ansible Tower services are running and all the required resources are available.

```
[student@workstation ~]$ lab inventory-smart setup
```

- ▶ 1. Verify that the facts for servera and serverb in the Dev host group are available in the Ansible Tower's cache. These two systems' facts are available in Ansible Tower's cache because in a previous exercise we executed a job on those managed hosts with a job template that had fact caching enabled.
 - 1.1. Click Inventories in the left navigation bar.
 - 1.2. Click on the link for the Dev host group, and go to HOSTS.
 - 1.3. Click on the link for servera.lab.example.com, and go to FACTS.
 - 1.4. Verify that the facts for servera.lab.example.com are available.
 - 1.5. Repeat the same steps to verify that the facts for serverb.lab.example.com are also available in Ansible Tower's cache.

- ▶ 2. Create a smart inventory, named Smart, which includes the Red Hat Enterprise Linux-based systems available in the Dev host group. The `ansible_distribution` fact will have the value **RedHat** for those systems.
 - 2.1. Click Inventories in the left navigation bar.
 - 2.2. Click +, and select Smart Inventory to create a new smart inventory.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Smart

FIELD	VALUE
DESCRIPTION	Smart Inventory
ORGANIZATION	Default
SMART HOST FILTER	<code>ansible_facts.ansible_distribution: RedHat</code>

**NOTE**

To enter the smart host filter, click the magnifying glass icon in SMART HOST FILTER. In the new window, enter the filter value in the top search box, and click the magnifying glass icon again. This filter displays both `servera.lab.example.com` and `serverb.lab.example.com` systems in the Dev host group. Click SAVE to set the smart host filter.

- 2.4. Click SAVE to create the smart inventory.
- 2.5. Click HOSTS, and verify that both `servera.lab.example.com` and `serverb.lab.example.com` are available.
- ▶ 3. Confirm that the host list for the Smart smart inventory matches more hosts when Ansible Tower adds facts for new hosts that match the host filter to its fact cache.
 - 3.1. Click Templates in the left navigation bar.
 - 3.2. Click on the link for the `TEST webservers setup` job template.
 - 3.3. Verify that you have selected the Use Fact Cache checkbox. Click SAVE to update the job template. You should see that this job template uses hosts from the Test inventory, which contains both the `serverc.lab.example.com`, and the `serverd.lab.example.com` hosts.
 - 3.4. Launch a job with the `TEST webservers setup` job template. Scroll down and click the rocket icon next to that template. When the job runs, it retrieves facts for the hosts `serverc.lab.example.com` and `serverd.lab.example.com`.
 - 3.5. Wait until the job's status is **Successful**, and then click Inventories in the left navigation bar.
 - 3.6. Click on the link for the Test inventory, and click the HOSTS button.
 - 3.7. Click on the link for `serverc.lab.example.com`. Go to FACTS and verify that the facts for that host are available. You can also verify that the value for the `ansible_distribution` fact is **RedHat** for that host.
 - 3.8. Repeat the previous step to verify that the facts for `serverd.lab.example.com` are available, and check that the value for the `ansible_distribution` fact is **RedHat**.
 - 3.9. Click Inventories in the left navigation bar, then click on the link for the Smart inventory.
 - 3.10. Go to HOSTS, and verify that both `serverc.lab.example.com` and `serverd.lab.example.com` are now available in this smart inventory. This is true because they meet the smart host filter condition.

- ▶ **4.** Log out from Ansible Tower.

This concludes the guided exercise.

► LAB

MANAGING ADVANCED INVENTORIES

PERFORMANCE CHECKLIST

In this lab, you will import a static inventory from a file, configure an inventory sourced from a project, create a dynamic inventory, and then create a smart inventory that selects its hosts from those inventories based on the value of a host variable.

OUTCOMES

- Use the **awx-manage** utility to import an existing inventory file.
- Use an existing static inventory file stored in a Git repository.
- Add a custom inventory script and use it to populate a Dynamic Inventory in Ansible Tower.
- Create a smart inventory.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run **lab inventory-review setup**. This setup script prepares the IdM server for this exercise.

```
[student@workstation ~]$ lab inventory-review setup
```

1. Import the existing static inventory available in the **/root/lab-example-inventory** file from the CLI.
2. Create a new project, named `LabProjectGit` that uses the Git repository located at `git.lab.example.com/home/git/inventory.git`.
3. Create a new inventory named `LabGit` with the **git-inventory-lab** inventory file available through the `LabProjectGit` project.
4. Create an inventory script, named **ldap-idm.py**, located at `http://materials.example.com/classroom/ansible/ipa-setup/ldap-idm.py`.
5. Create a new Inventory called `Lab Dynamic Inventory`.
6. Add the **ldap-idm.py** inventory script as a new source for the `Lab Dynamic Inventory` inventory.
7. Modify the `Demo Job Template` job template to use the `LabGit` inventory, and launch a job with it to trigger the caching of its associated managed hosts facts in the fact cache.
8. Create a smart inventory named `LabSmart`, which includes the Red Hat-based systems available in the `LabGit` host group.
9. Run the command **lab inventory-review grade** on `workstation` to grade your exercise.

This concludes the lab.

► SOLUTION

MANAGING ADVANCED INVENTORIES

PERFORMANCE CHECKLIST

In this lab, you will import a static inventory from a file, configure an inventory sourced from a project, create a dynamic inventory, and then create a smart inventory that selects its hosts from those inventories based on the value of a host variable.

OUTCOMES

- Use the **awx-manage** utility to import an existing inventory file.
- Use an existing static inventory file stored in a Git repository.
- Add a custom inventory script and use it to populate a Dynamic Inventory in Ansible Tower.
- Create a smart inventory.

BEFORE YOU BEGIN

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the `tower` system.

Log in as the student user on `workstation` and run **lab inventory-review setup**. This setup script prepares the IdM server for this exercise.

```
[student@workstation ~]$ lab inventory-review setup
```

1. Import the existing static inventory available in the **/root/lab-example-inventory** file from the CLI.
 - 1.1. On `workstation`, open a terminal. To use the **awx-manage** command, log in to the `tower` server as root user using SSH.

```
[student@workstation ~]$ ssh root@tower
[root@tower ~]#
```

- 1.2. On the `tower` server, list the contents of the `root` user home directory to ensure that the static inventory file for Ansible Tower is available.

```
[root@tower ~]# ls /root
...output omitted...
lab-example-inventory
...output omitted...
```

- 1.3. Using the **awx-manage** command and the **inventory_import** subcommand, import the **lab-example-inventory** static inventory file containing an inventory. Use the existing **Lab** name as the destination for the import. After the import has completed, log out of the `tower` system.

```
[root@tower ~]# awx-manage inventory_import \
> --source=/root/lab-example-inventory \
> --inventory-name="Lab"
2.467 INFO      Updating inventory 5: Lab
2.623 INFO      Reading Ansible inventory source: /root/lab-example-inventory
3.960 INFO      Processing JSON output...
3.961 INFO      Loaded 1 groups, 2 hosts
4.192 INFO      Inventory import completed for (Lab - 11) in 1.7s
[root@tower ~]# exit
```

- 1.4. Log into the Ansible Tower web UI as **admin**. Use **redhat** as a password.
- 1.5. In the Ansible Tower web UI, click **Inventories** in the left navigation bar to display the list of inventories. You should see an inventory named **Lab** which was used in a previous step for the import.
- 1.6. Click the **Lab** link to view the details of the imported static Inventory. Look at the available GROUPS and HOSTS sections, you should see that the inventory has the host group **virtualization** and **serverc.lab.example.com**, and **serverd.lab.example.com** as hosts. This confirms that the static inventory has been successfully imported and is accessible.
2. Create a new project, named **LabProjectGit** that uses the Git repository located at **git.lab.example.com/home/git/inventory.git**.
 - 2.1. Click **Projects** in the left navigation bar.
 - 2.2. Click **+** to add a new Project.
 - 2.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	LabProjectGit
DESCRIPTION	Project Based on Git
ORGANIZATION	Default
SCM TYPE	Git
SCM URL	ssh://git@git.lab.example.com/home/git/inventory.git
SCM CREDENTIAL	student-git

- 2.4. Select the **Update Revision on Launch** checkbox.
- 2.5. Click **SAVE** to add the inventory.

3. Create a new inventory named LabGit with the **git-inventory-lab** inventory file available through the LabProjectGit project.
 - 3.1. Click Inventories in the left navigation bar.
 - 3.2. Click +, and select Inventory to add a new inventory.
 - 3.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	LabGit
DESCRIPTION	Static Inventory from Git
ORGANIZATION	Default

- 3.4. Click SAVE to add the inventory.
- 3.5. Go to the SOURCES section, and click + to add a new source.
- 3.6. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	git-inventory-lab
DESCRIPTION	Static Inventory from Git
SOURCE	Sourced from a Project
PROJECT	LabProjectGit
INVENTORY FILE	git-inventory-lab

**NOTE**

Make sure you manually enter **git-inventory-lab** in the INVENTORY FILE combo box. The file may not display in the combo box's drop-down list because of a known product issue.

- 3.7. Select the Update on Project Change checkbox.
- 3.8. Click SAVE to add the inventory.
- 3.9. Scroll down to verify that the cloud icon next to **git-inventory-lab** is static and green.
- 3.10. Verify that the storage host group is available in the GROUPS section, and the `servere.lab.example.com`, and the `serverf.lab.example.com` hosts are available under the HOSTS section.

4. Create an inventory script, named **ldap-idm.py**, located at <http://materials.example.com/classroom/ansible/ipa-setup/ldap-idm.py>.
- 4.1. Click **Inventory Scripts** in the left navigation bar.
 - 4.2. Click the **+** button to add a new custom inventory script.
 - 4.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	ldap-idm.py
DESCRIPTION	Dynamic Inventory for IdM
ORGANIZATION	Default

- 4.4. Copy the contents of the **ldap-idm.py** script located at <http://materials.example.com/classroom/ansible/ipa-setup/ldap-idm.py> into the **CUSTOM SCRIPT** field.
 - 4.5. Click **SAVE** to add the custom inventory script.
5. Create a new Inventory called **Lab Dynamic Inventory**.
- 5.1. Click **Inventories** in the left navigation bar.
 - 5.2. Click the **+** button to add a new inventory, and select **Inventory** in the drop-down menu.
 - 5.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Lab Dynamic Inventory
DESCRIPTION	Dynamic Inventory from IdM server
ORGANIZATION	Default

- 5.4. Click **SAVE** to create the Inventory.
6. Add the **ldap-idm.py** inventory script as a new source for the **Lab Dynamic Inventory** inventory.
- 6.1. Click **SOURCES** in the top bar for the Inventory details pane.
 - 6.2. Click the **+** button to add a new source.
 - 6.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	Custom Script
DESCRIPTION	Custom Script for Lab Dynamic Inventory
SOURCE	Custom Script

FIELD	VALUE
CUSTOM INVENTORY SCRIPT	ldap-idm.py

- 6.4. Under the UPDATE OPTIONS section, check the checkbox next to the Overwrite option.
- 6.5. Click SAVE to create the Source.
- 6.6. Scroll down to the lower pane, and click the double-arrow button in the row for Custom Script. Wait until the cloud becomes green and static.
- 6.7. Use the top breadcrumb menu to navigate back to the Lab Dynamic Inventory inventory details pane, and click the GROUPS button. Observe that it now contains four Groups: development, ipaservers, testing, and production. Each of these groups contains hosts.
7. Modify the Demo Job Template job template to use the LabGit inventory, and launch a job with it to trigger the caching of its associated managed hosts facts in the fact cache.
 - 7.1. Click Templates in the left navigation bar.
 - 7.2. Click on the link for the Demo Job Template job template.
 - 7.3. Select the LabGit inventory on INVENTORY.
 - 7.4. On CREDENTIAL, select **Operations** credential under the **Machine** credential type.
 - 7.5. Select the Use Fact Cache checkbox, and click SAVE to update the job template. See that this job template uses the LabGit inventory, which contains both servere.lab.example.com, and serverf.lab.example.com.
 - 7.6. Scroll down, and click the rocket icon for the Demo Job Template job template to launch a job with that job template. During its execution, this job retrieves the facts for both servere.lab.example.com, and serverf.lab.example.com.
 - 7.7. Wait until the job status is **Successful**, and click Inventories in the left navigation bar.
 - 7.8. Click on the link for the LabGit inventory, and go to HOSTS.
 - 7.9. Click on the link for servere.lab.example.com, go to FACTS, and verify that the facts for that host are available. You can also verify in the fact lists that the value for the ansible_distribution fact is **RedHat**.
 - 7.10. Repeat the previous step to verify that the facts for serverf.lab.example.com are available, and check that the value for the ansible_distribution fact is **RedHat**

8. Create a smart inventory named **LabSmart**, which includes the Red Hat-based systems available in the **LabGit** host group.
 - 8.1. Click **Inventories** in the left navigation bar.
 - 8.2. Click **+**, and select **Smart Inventory** to create a new smart inventory.
 - 8.3. On the next screen, fill in the details as follows:

FIELD	VALUE
NAME	LabSmart
DESCRIPTION	Lab Smart Inventory
ORGANIZATION	Default
SMART HOST FILTER	ansible_facts.ansible_distribution: RedHat



NOTE

To enter the host filter, click the magnifying glass icon in SMART HOST FILTER. A new window opens. Enter the host filter's value in the combo box at the top of the window, and click the magnifying glass icon. If you have entered it correctly, the filter should display both `servere.lab.example.com` and `serverf.lab.example.com` systems in the **LabGit** host group at the bottom of the window. Click **SAVE** to create the smart host filter.

- 8.4. Click **SAVE** to create the smart inventory.
- 8.5. Click **HOSTS**, and verify that both `servere.lab.example.com` and `serverf.lab.example.com` are available.
9. Run the command **lab inventory-review grade** on **workstation** to grade your exercise.

SUMMARY

In this chapter, you learned:

- You can use the **awx-manage inventory_import** command to import an existing static inventory into Red Hat Ansible Tower.
- You can use an inventory that is managed externally in a Git repository by configuring the repository as a Project and setting up the inventory source as Sourced from Project.
- Red Hat Ansible Tower includes built-in support for some dynamic inventories, and you can also configure it to use your own custom dynamic inventory script.
- Red Hat Ansible Tower also provides smart inventories, which allow you to build an inventory from all the other inventories on the Ansible Tower server by filtering hosts based on Ansible facts or other criteria.

CHAPTER 16

PERFORMING MAINTENANCE AND ROUTINE ADMINISTRATION OF ANSIBLE TOWER

GOAL

Perform routine maintenance and administration of Ansible Tower.

OBJECTIVES

- Describe the low-level components of Red Hat Ansible Tower, locate and examine relevant log files, control Ansible Tower services, and perform basic troubleshooting.
- Replace the default TLS certificate for Ansible Tower with an updated certificate obtained from a certificate authority.
- Back up and restore the Ansible Tower database and configuration files.

SECTIONS

- Performing Basic Troubleshooting of Ansible Tower (and Guided Exercise)
- Configuring TLS/SSL for Ansible Tower (and Guided Exercise)
- Backing Up and Restoring Ansible Tower (and Guided Exercise)

QUIZ

- Maintaining Ansible Tower

PERFORMING BASIC TROUBLESHOOTING OF ANSIBLE TOWER

OBJECTIVES

After completing this section, students should be able to describe the low-level components of Red Hat Ansible Tower, locate and examine relevant log files, control Ansible Tower services, and perform basic troubleshooting.

Figure 16.0: Basic troubleshooting of Ansible Tower infrastructure

ANSIBLE TOWER COMPONENTS

Red Hat Ansible Tower is a web application made up of a number of cooperating processes and services. Four main network services are enabled, which start the rest of the components of Ansible Tower:

- Nginx provides the web server that hosts the Ansible Tower application and supports the web UI and the API.
- PostgreSQL is the database that stores most Ansible Tower data, configuration, and history.
- Supervisord is a process control system that itself manages the various components of the Ansible Tower application to do perform operations such as schedule and run jobs, listen for callbacks from running jobs, and so on.
- Rabbitmq-server is an AMQP message broker that supports signaling for the Ansible Tower application components.

A fifth component also used by Ansible Tower is the memcached memory object caching daemon, which is used as a local caching service.

These network services communicate with each other using normal network protocols. For a normal self-contained Ansible Tower server, the main ports that need to be exposed outside the system are 80/tcp and 443/tcp, to allow clients to access the web UI and API.

However, the other services may also expose ports to external clients unless specifically protected. For example, the PostgreSQL service listens for connections from anywhere on 5432/tcp, and the RabbitMQ server **beam** listens for connections on 5672/tcp, 15672/tcp, and 25672/tcp. If only the local Ansible Tower services need to be able to connect to these ports, it may be desirable to block access to them using the local firewall.



WARNING

This is one reason why setting good passwords for the PostgreSQL and RabbitMQ services in the **inventory** file used to install Ansible Tower is important. These services can be contacted by internet clients directly by default, and weak passwords may leave them vulnerable to remote attack.

Starting, Stopping, and Restarting Ansible Tower

Ansible Tower ships with a **/usr/bin/ansible-tower-service** script, which can start, stop, restart, and give the status of the major Ansible Tower services, including the database and message queue components.

```
[root@tower ~]# ansible-tower-service status
Showing Tower Status
● postgresql-9.6.service - PostgreSQL 9.6 database server
  Loaded: loaded (/usr/lib/systemd/system/postgresql-9.6.service; enabled; vendor
  preset: disabled)
  ...output omitted...
Status of node rabbitmq@localhost ...
[{"pid":1554},
 {"running_applications,
  ...output omitted...
● nginx.service - The nginx HTTP and reverse proxy server
  Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; vendor preset:
  disabled)
  Active: active (running) since czw 2018-11-21 03:57:56 EDT; 3h 58min ago
  ...output omitted...
● supervisord.service - Process Monitoring and Control Daemon
  Loaded: loaded (/usr/lib/systemd/system/supervisord.service; enabled; vendor
  preset: disabled)
  Active: active (running) since czw 2018-11-21 03:57:57 EDT; 3h 58min ago
```

To access the list of available options, run the **ansible-tower-service** command without any options:

```
[root@tower ~]# ansible-tower-service
Usage: /usr/bin/ansible-tower-service {start|stop|restart|status}
```

The following example demonstrates restarting the Ansible Tower infrastructure:

```
[root@tower ~]# ansible-tower-service restart
Restarting Tower
Redirecting to /bin/systemctl stop postgresql-9.6.service
Stopping rabbitmq-server (via systemctl): [ OK ]
Redirecting to /bin/systemctl stop nginx.service
Redirecting to /bin/systemctl stop supervisord.service
Redirecting to /bin/systemctl start postgresql-9.6.service
Starting rabbitmq-server (via systemctl): [ OK ]
Redirecting to /bin/systemctl start nginx.service
Redirecting to /bin/systemctl start supervisord.service
```

Supervisord Components

`supervisord` is a process control system often used to control Django-based applications. It is used to manage and monitor long-running processes or daemons, and to automatically restart them as needed. In Ansible Tower, `supervisord` manages important components of the Ansible Tower application itself.

You can use the **supervisorctl status** command to see the list of Ansible Tower processes controlled by the `supervisord` service:

```
[root@tower ~]# supervisorctl status
exit-event-listener          RUNNING    pid 15382, uptime 0:34:22
tower-processes:awx-callback-receiver  RUNNING    pid 15387, uptime 0:34:22
tower-processes:awx-celeryd        RUNNING    pid 15389, uptime 0:34:22
tower-processes:awx-celeryd-beat   RUNNING    pid 15388, uptime 0:34:22
```

tower-processes:awx-channels-worker	RUNNING	pid 15383, uptime 0:34:22
tower-processes:awx-daphne	RUNNING	pid 15386, uptime 0:34:22
tower-processes:awx-fact-cache-receiver	RUNNING	pid 15385, uptime 0:34:22
tower-processes:awx-uwsgi	RUNNING	pid 15384, uptime 0:34:22

As you can see in the preceding output, supervisord controls a number of processes owned by the awx user. One of them is the awx-celeryd daemon, which is used as a real time, distributed message-passing task and job queue.

ANSIBLE TOWER CONFIGURATION AND LOG FILES

Configuration Files

The main configuration files for Ansible Tower are kept in the `/etc/tower` directory. These include settings files for the Ansible Tower application which are outside the PostgreSQL database, the TLS certificate for `nginx` and other key files.

Perhaps the most important of these files for the Ansible Tower application is the `/etc/tower/settings.py` file, which specifies the locations for job output, project storage, and other directories.

The other individual services may have service-specific configuration files elsewhere on the system, such as the `/etc/nginx` files used by the web server.

Log Files

The Ansible Tower application log files are stored in one of two centralized locations:

- `/var/log/tower/`
- `/var/log/supervisor/`

Ansible Tower server errors are logged in the `/var/log/tower/` directory. Some key files in the `/var/log/tower/` directory include:

- `/var/log/tower/tower.log`, the main log for the Ansible Tower application.
- `/var/log/tower/setup*.log`, which are logs of runs of the `setup.sh` script to install, back up, or restore the Ansible Tower server.
- `/var/log/tower/task_system.log`, which logs various system housekeeping tasks (such as the removal of the record of old job runs).

The `/var/log/supervisor/` directory stores log files for services, daemons, and applications managed by supervisord. The `supervisord.log` file in this directory is the main log file for the service that controls all of these daemons. The other files contain log information about the activity of those daemons.

Ansible Tower can also send detailed logs to external log aggregation services. Log aggregation can offer insight into Ansible Tower technical trends or usage. The data can be used to monitor for anomalies, analyze events, and correlate events. Splunk, Elastic stack/logstash (formerly ELK), Loggly, and Sumologic are all log aggregation and data analysis systems that can be used with Ansible Tower.

More information on how to configure such services is located in the *Ansible Tower Administration Guide* at <https://docs.ansible.com/ansible-tower/latest/html/administration/>.

**IMPORTANT**

This discussion has focused on looking at the log files to troubleshoot problems with the Ansible Tower server itself.

If you encounter errors running playbooks which do not appear to be related to actual errors in the Ansible Tower configuration, remember to look at the output of your launched jobs in the Ansible Tower web UI or the API.

Other Ansible Tower Files

A number of other key files for Ansible Tower are kept in the `/var/lib/awx` directory. This directory includes:

- `/var/lib/awx/public/static`: for static root directory (this is the location of your Django based application files).
- `/var/lib/awx/projects`: projects root directory (in the subdirectories of this directory Ansible Tower will store project based files - for example `git` repository files).
- `/var/lib/awx/job_status`: Job status output from playbooks is stored in this file.

COMMON TROUBLESHOOTING SCENARIOS

Problems running playbooks

The default configuration confines playbooks to the `/tmp` directory and limits what the playbook can access locally on the Ansible Tower server. This can impact tasks that the playbook may delegate to the local system rather than the target host.

Review your license status and the number of unique hosts you have managed by the Ansible Tower server. If the license has expired, or too many hosts are registered, you will not be able to launch jobs.

Problems connecting to your host

If you encounter problems with connectivity errors while running playbooks, try the following:

- Verify that you can establish an `ssh` or `winrm` connection with the managed host. Ansible depends upon `ssh` (or `winrm` for Microsoft Windows systems) to access the servers you are managing.
- Review your inventory file. Review the host names and IP addresses.

Playbooks are not accessible in the Job Template drop-down

If your playbooks are not showing up in the Job Template list, review these items:

- Review the playbook's YAML syntax and make sure that it can be parsed by Ansible.
- Make sure the permissions and ownership of the project path (`/var/lib/awx/projects/`) are configured correctly so that the awx system user can view the files.

Playbook stays in pending

When you are trying to run a Job and it stays in the Pending state, try the following:

- Ensure that the Ansible Tower server has enough memory available and that the services governed by `supervisord` are running. Run the `supervisorctl status` command.

- Ensure that the partition where the `/var/` directory is located has more than 1 GB of space available. Jobs will not complete when there is insufficient free space on the `/var/` partition.
- Restart the Ansible Tower infrastructure using the `ansible-tower-service restart` command.

Error: provided hosts list is empty

If you encounter the error message `Skipping: No Hosts Matched` when you are trying to run a playbook through Ansible Tower, review the following:

- Review and make sure that the host patterns used by the `hosts` declaration in your play matches the group or host names in the inventory. The host patterns are case sensitive.
- Make sure your group names have no spaces and modify them to use underscores or no spaces to ensure that the groups are correctly recognized.
- If you have specified a limit in the Job Template, make sure that it is a valid limit and that it matches something in your inventory.

PERFORMING COMMAND-LINE MANAGEMENT

Ansible Tower ships with the `awx-manage` command-line utility, which can be used to access detailed internal Ansible Tower information. The `awx-manage` command must be run as `root` or as the `awx` (Ansible Tower) user. This utility is most commonly used to reset the Ansible Tower's `admin` password and to import an existing static inventory file into the Ansible Tower server.

Changing the Ansible Tower Admin Password

The password for the built-in Ansible Tower System Administrator account, `admin`, is initially set when the Ansible Tower server is installed. The `awx-manage` command offers a way to change the administrator password from the command line. To do this, as the `root` or `awx` user on the Ansible Tower server, use the `changepassword` option:

```
[root@tower ~]# awx-manage changepassword admin
Changing password for user 'admin'
Password: new_password
Password (again): new_password
Password changed successfully for user 'admin'
```

After entering the new password twice, the password you have entered will be the `admin` password in the Ansible Tower web UI.

You can also create a new Ansible Tower superuser, with administrative privileges if needed. To create a new superuser you can use `awx-manage` with the `createsuperuser` option.

```
[root@tower ~]# awx-manage createsuperuser
Username (leave blank to use 'root'): admin3
Email address: admin@demo.example.com
Password: new_password
Password (again): new_password
Superuser created successfully.
```



REFERENCES

Ansible Tower Administration Guide

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

PERFORMING BASIC TROUBLESHOOTING OF ANSIBLE TOWER

In this exercise, you will identify the locations of Ansible Tower log files and use them for basic troubleshooting.

OUTCOMES

You should be able to:

- Start, stop, and restart Ansible Tower services.
- Use the log files to troubleshoot Ansible Tower.
- Use the **awx-manage** utility to reset the **admin** password.

Ensure that the **workstation** and **tower** virtual machines are running.

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run **lab admin-troubleshoot setup**, which prepares the **tower.lab.example.com** server for this exercise.

```
[student@workstation ~]$ lab admin-troubleshoot setup
```

- 1. On **workstation**, open Firefox and try to log in to the Ansible Tower web UI at <https://tower.lab.example.com/>.
This should fail. This exercise investigates that failure.

Server Error
A server error has occurred.

- 2. On **workstation**, open a terminal and use **ssh** to log in to **tower** as **root**.

```
[student@workstation ~]$ ssh root@tower  
Last login: Wed Nov 21 06:27:17 2018 from workstation.lab.example.com  
[root@tower ~]#
```

- 3. Ensure that services making up the main components of Ansible Tower are all running and that the server's firewall is not blocking communications.
- 3.1. To eliminate potential connection errors caused by firewall rules, review the current **firewalld** configuration by using **firewall-cmd** command with the **--list-ports** option.

```
[root@tower ~]# firewall-cmd --list-ports
```

```
443/tcp 80/tcp
```

As you can see in the output, port 80 and 443 are not blocked. This is not a surprise, because you received a response from the Ansible Tower web server earlier, even though the response was a Server Error.

- 3.2. Next, determine the status of the services that make up the Ansible Tower infrastructure. Use the **ansible-tower-service** script to run **ansible-tower-service status**.

```
[root@tower ~]# ansible-tower-service status
Showing Tower Status
● postgresql-9.6.service - PostgreSQL 9.6 database server
  Loaded: loaded (/usr/lib/systemd/system/postgresql-9.6.service; enabled; vendor
  preset: disabled)
  Active: inactive (dead) since Wed 2018-11-21 04:20:43 EDT; 24min ago
    ...output omitted...
● nginx.service - The nginx HTTP and reverse proxy server
  Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; vendor preset:
  disabled)
  Active: active (running) since Wed 2018-11-21 04:20:49 EDT; 24min ago
    ...output omitted...
● supervisord.service - Process Monitoring and Control Daemon
  Loaded: loaded (/usr/lib/systemd/system/supervisord.service; enabled; vendor
  preset: disabled)
  Active: active (running) since Wed 2018-11-21 04:20:55 EDT; 23min ago
    Process: 909 ExecStart=/usr/bin/supervisord -c /etc/supervisord.conf
    (code=exited, status=0/SUCCESS)
  Main PID: 1101 (code=exited, status=0/SUCCESS)
```

As you can see in the output, one of the services required for the Ansible Tower infrastructure is marked as **inactive (dead)**. The services directly started by **ansible-tower-service** for the Ansible Tower infrastructure to work properly are **postgresql-9.6**, **nginx**, **supervisord**, and **rabbitmq-server**.

- 3.3. Restart the Ansible Tower infrastructure using the **ansible-tower-service admin** utility script.

```
[root@tower ~]# ansible-tower-service restart
Restarting Tower
Redirecting to /bin/systemctl stop postgresql-9.6.service
Stopping rabbitmq-server (via systemctl): [ OK ]
Redirecting to /bin/systemctl stop nginx.service
Redirecting to /bin/systemctl stop supervisord.service
Redirecting to /bin/systemctl start postgresql-9.6.service
Starting rabbitmq-server (via systemctl): [ OK ]
Redirecting to /bin/systemctl start nginx.service
Redirecting to /bin/systemctl start supervisord.service
```

- 3.4. To confirm that the Ansible Tower infrastructure has started, go back to **workstation**, open Firefox and log in to the Ansible Tower web UI.

- 4. The next step is to determine why the PostgreSQL database was not running.

- 4.1. The **ansible-tower-service** script apparently controls PostgreSQL as a **postgresql-9.6.service** unit through **systemctl**, so you can investigate further using that tool:

```
[root@tower ~]# systemctl status postgresql-9.6 -l
● postgresql-9.6.service - PostgreSQL 9.6 database server
  Loaded: loaded (/usr/lib/systemd/system/postgresql-9.6.service; enabled; vendor
  preset: disabled)
  Active: active (running) since Wed 2018-11-21 04:52:45 EDT; 2min ago
    Process: 28501 ExecStop=/usr/pgsql-9.6/bin/pg_ctl stop -D ${PGDATA} -s -m fast
               (code=exited, status=0/SUCCESS)
    Process: 29737 ExecStart=/usr/pgsql-9.6/bin/pg_ctl start -D ${PGDATA} -s -w -t
               300 (code=exited, status=0/SUCCESS)
    Process: 29732 ExecStartPre=/usr/pgsql-9.6/bin/postgresql94-check-db-dir
               ${PGDATA} (code=exited, status=0/SUCCESS)
  Main PID: 29741 (postgres)
     CGroup: /system.slice/postgresql-9.6.service
             ├─29741 /usr/pgsql-9.6/bin/postgres -D /var/lib/pgsql/9.6/data
             ├─29742 postgres: logger process
             ├─29744 postgres: checkpointer process
             ├─29745 postgres: writer process
             ├─29746 postgres: wal writer process
             ├─29747 postgres: autovacuum launcher process
             ├─29748 postgres: stats collector process
             ├─30579 postgres: awx awx 127.0.0.1(46134) idle
             ├─30582 postgres: awx awx 127.0.0.1(46146) idle
             ├─30583 postgres: awx awx 127.0.0.1(46148) idle
             └─30598 postgres: awx awx 127.0.0.1(46166) idle

Mar 28 04:52:44 tower.lab.example.com systemd[1]: Starting PostgreSQL 9.6 database
server...
Mar 28 04:52:44 tower.lab.example.com pg_ctl[29737]: < 2018-11-21 04:52:44.342 EDT
>LOG:  redirecting log output to logging collector process
Mar 28 04:52:44 tower.lab.example.com pg_ctl[29737]: < 2018-11-21 04:52:44.342 EDT
>HINT:  Future log output will appear in directory "pg_log".
Mar 28 04:52:45 tower.lab.example.com systemd[1]: Started PostgreSQL 9.6 database
server.
```

The PostgreSQL server appears to be logging to a **pg_log** directory. The **find** command illustrates where that is:

```
[root@tower ~]# find / -name pg_log
/var/lib/pgsql/9.6/data/pg_log
```

- 4.2. Change to the **/var/lib/pgsql/9.6/data/pg_log** directory and inspect the log files.

```
[root@tower ~]# cd /var/lib/pgsql/9.6/data/pg_log
[root@tower pg_log]# ls -l
...output omitted...
```

```
-rw----- 1 postgres postgres 68055 Mar 28 04:55 postgresql-Tue.log
```

- 4.3. Examine the log file that recorded messages near the time that PostgreSQL stopped. In this example, this was at Wed 2018-11-21 04:20:43 EDT based on the output reported the first time we ran the **ansible-tower-service status** command and saw the failure. The log file and time you should actually use may be different from the one in this example, depending on when you do this exercise.

```
[root@tower pg_log]# less postgresql-Wed.log
...output omitted...
< 2018-11-21 04:20:42.601 EDT >LOG:  received fast shutdown request
< 2018-11-21 04:20:42.601 EDT >LOG:  aborting any active transactions
< 2018-11-21 04:20:42.601 EDT >FATAL:  terminating connection due to administrator
command
< 2018-11-21 04:20:42.602 EDT >FATAL:  terminating connection due to administrator
command
< 2018-11-21 04:20:42.603 EDT >FATAL:  terminating connection due to administrator
command
< 2018-11-21 04:20:42.603 EDT >FATAL:  terminating connection due to administrator
command
< 2018-11-21 04:20:42.604 EDT >FATAL:  terminating connection due to administrator
command
< 2018-11-21 04:20:42.604 EDT >FATAL:  terminating connection due to administrator
command
< 2018-11-21 04:20:42.605 EDT >LOG:  autovacuum launcher shutting down
< 2018-11-21 04:20:42.606 EDT >LOG:  shutting down
< 2018-11-21 04:20:42.744 EDT >LOG:  database system is shut down
...output omitted...
```

It looks like someone manually stopped the service.

- 5. Now that you have solved the mystery of the Server Error, the remainder of this exercise explores other log files and useful tools for troubleshooting Ansible Tower.

The **supervisord** service is responsible for running a collection of programs that control the main logic of Ansible Tower. This includes the **awx-celeryd** worker queues that run jobs and the **awx-callback-receiver** processes that receive job events from running jobs.

A number of the log files for the supervisord service are in the **/var/log/supervisor** directory on the Ansible Tower server. In that directory, display the end of the file **supervisord.log**.

```
[root@tower pg_log]# cd /var/log/supervisor
[root@tower supervisor]# tail -n 50 supervisord.log
...output omitted...
2018-11-21 05:14:07,247 ERRO pool exit-event-listener event buffer overflowed,
discarding event 20
2018-11-21 05:14:07,247 INFO stopped: awx-daphne (exit status 0)
2018-11-21 05:14:07,318 ERRO pool exit-event-listener event buffer overflowed,
discarding event 21
2018-11-21 05:14:07,318 INFO stopped: awx-celeryd-beat (exit status 0)
2018-11-21 05:14:08,280 ERRO pool exit-event-listener event buffer overflowed,
discarding event 22
```

```
2018-11-21 05:14:08,280 INFO stopped: awx-uwsgi (exit status 0)
2018-11-21 05:14:09,282 INFO waiting for awx-callback-receiver to die
2018-11-21 05:14:11,285 WARN killing 'awx-callback-receiver' (14267) with SIGKILL
2018-11-21 05:14:25,348 CRIT Supervisor running as root (no user in config file)
2018-11-21 05:14:25,349 WARN Included extra file "/etc/supervisord.d/tower.ini"
  during parsing
2018-11-21 05:14:25,349 INFO Increased RLIMIT_NOFILE limit to 4096
2018-11-21 05:14:25,396 INFO RPC interface 'supervisor' initialized
...output omitted...
2018-11-21 05:14:25,397 CRIT Server 'unix_http_server' running without any HTTP
  authentication checking
2018-11-21 05:14:25,403 INFO daemonizing the supervisord process
2018-11-21 05:14:25,409 INFO supervisord started with pid 16949
2018-11-21 05:14:26,422 INFO spawned: 'exit-event-listener' with pid 16950
2018-11-21 05:14:26,430 INFO spawned: 'awx-channels-worker' with pid 16951
2018-11-21 05:14:26,605 INFO spawned: 'awx-celeryd' with pid 16957
2018-11-21 05:14:27,812 ERRO pool exit-event-listener event buffer overflowed,
  discarding event 2
2018-11-21 05:14:27,812 INFO success: awx-daphne entered RUNNING state, process
  has stayed up for > than 1 seconds (startsecs)
2018-11-21 05:14:27,813 ERRO pool exit-event-listener event buffer overflowed,
  discarding event 3
2018-11-21 05:14:27,813 INFO success: awx-callback-receiver entered RUNNING state,
  process has stayed up for > than 1 seconds (startsecs)
2018-11-21 05:14:27,813 ERRO pool exit-event-listener event buffer overflowed,
  discarding event 4
2018-11-21 05:14:27,813 INFO success: awx-celeryd-beat entered RUNNING state,
  process has stayed up for > than 1 seconds (startsecs)
2018-11-21 05:14:27,813 ERRO pool exit-event-listener event buffer overflowed,
  discarding event 5
2018-11-21 05:14:27,813 INFO success: awx-celeryd entered RUNNING state, process
  has stayed up for > than 1 seconds (startsecs)
...output omitted...
```

As you can see in the output, some services governed by the `supervisord` service had also been stopped. After executing the `ansible-tower-service restart` command to restart the stopped PostgreSQL server, they were also successfully spawned by `supervisord`, which allowed you to log in to the Ansible Tower web UI.

- 6. You can also determine the status of the processes managed by `supervisord` by using the `supervisorctl` command.

```
[root@tower supervisor]# supervisorctl status
exit-event-listener                  RUNNING    pid 4111, uptime 0:42:55
tower-processes:awx-callback-receiver RUNNING    pid 4116, uptime 0:42:55
tower-processes:awx-celeryd           RUNNING    pid 4118, uptime 0:42:55
tower-processes:awx-celeryd-beat     RUNNING    pid 4117, uptime 0:42:55
tower-processes:awx-channels-worker   RUNNING    pid 4112, uptime 0:42:55
tower-processes:awx-daphne          RUNNING    pid 4115, uptime 0:42:55
tower-processes:awx-uwsgi            RUNNING    pid 4113, uptime 0:42:55
```

- 7. Other log files relevant to Ansible Tower are kept in the `/var/log/tower` directory. For example, if there is a problem with job execution using Ansible Tower, one file to examine

is the **/var/log/tower/tower.log** file. This log contains useful information about the status of executed jobs and changes in Ansible Tower Inventories or Job Templates.

```
[root@tower supervisor]# less /var/log/tower/tower.log
...output omitted...
  File "/var/lib/awx/venv/tower/lib/python2.7/site-packages/psycopg2/__init__.py",
line 164, in connect
    conn = _connect(dsn, connection_factory=connection_factory, async=async)
OperationalError: could not connect to server: Connection refused
    Is the server running on host "127.0.0.1" and accepting
    TCP/IP connections on port 5432?

2018-11-21 05:02:44,375 WARNING awx.api.generics status 404 received by
user admin attempting to access /api/v2/job_templates/55555/launch/ from
172.25.250.254
2018-11-21 05:05:59,920 WARNING awx.api.generics status 404 received by
user admin attempting to access /api/v2/job_templates/55555/launch/ from
172.25.250.254
2018-11-21 05:06:26,066 ERROR     awx.main.tasks Failed to update ProjectUpdate
after 5 retries.
2018-11-21 05:06:51,108 ERROR     awx.main.tasks Failed to update ProjectUpdate
after 5 retries.
2018-11-21 05:08:24,920 WARNING awx.api.generics status 404 received by
user admin attempting to access /api/v2/job_templates/55555/launch/ from
172.25.250.254
2018-11-21 05:08:27,523 ERROR     awx.main.scheduler Task
awx.main.scheduler.partial.ProjectUpdateDict object at 0x6705710 appears
orphaned... marking as failed
...output omitted...
```

The log messages here indicate that something is trying to launch a nonexistent Job Template.

- ▶ 8. Those attempts by **admin** to use the API to launch a nonexistent Job Template with ID 55555 from 172.25.250.254 should also show up in the **nginx** web server's **access.log** file.

The logs for **nginx** are located in the **/var/log/nginx** directory. Look at the **access.log** file for events that happened at the same time as the **WARNING** in the **tower.log** file:

```
[root@tower supervisor]# less /var/log/nginx/access.log
...output omitted...
172.25.250.254 - admin [26/Nov/2018:05:02:44 -0400] "POST /api/v2/job_templates/
55555/launch/ HTTP/1.1" 404 34 "-" "curl/7.47.1" "-"
...output omitted...
```

- ▶ 9. You have resolved the troubleshooting problem. In the following steps you will change the Ansible Tower **admin** user password by using the **awx-manage** command. You already have a shell prompt as **root** on your **tower** server.

- 10. Create a new System Administrator user with the use of **awx-manage** command and **createsuperuser** subcommand.

```
[root@tower ~]# awx-manage createsuperuser
```

- 10.1. Use **admin2** as the name of the new superuser.

```
Username (leave blank to use 'root'): admin2
```

- 10.2. Leave the email address blank.

```
Username (leave blank to use 'root'): admin2
```

```
Email address:
```

- 10.3. Enter **redhat** as password.

```
Username (leave blank to use 'root'): admin2
```

```
Email address:
```

```
Password: redhat
```

- 10.4. Repeat previous step.

```
Username (leave blank to use 'root'): admin2
```

```
Email address:
```

```
Password:
```

```
Password (again): redhat
```

```
Superuser created successfully.
```

- 11. Using the **awx-manage** command and the **changepassword** subcommand, reset the **admin2** user password. When prompted, type **redhat2** as the new password twice.

```
[root@tower ~]# awx-manage changepassword admin2
Changing password for user 'admin2'
Password: redhat2
Password (again): redhat2
Password changed successfully for user 'admin2'
```

- 12. To confirm the change, open Firefox on workstation and log in to Ansible Tower as **admin2** user with the new **redhat2** password.

Cleanup

On workstation, run the **lab admin-troubleshoot cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab admin-troubleshoot cleanup
```

CONFIGURING TLS/SSL FOR ANSIBLE TOWER

OBJECTIVES

After completing this section, students should be able to:

- Replace the default TLS certificate for Ansible Tower with an updated certificate obtained from a certificate authority.

NGINX WEB SERVER ON ANSIBLE TOWER

The Ansible Tower web UI is provided by an Nginx web server running on the Tower server. When installed, Ansible Tower creates a self-signed TLS certificate and matching private key file which Nginx uses for HTTPS communication.

The main configuration files for Nginx are in the `/etc/nginx` directory, the most important of which is `/etc/nginx/nginx.conf`. Access logs for the Nginx web server hosting the Ansible Tower web UI are located in `/var/log/nginx/access.log` and error logs are in `/var/log/nginx/error.log`. Both log files are periodically rotated, and `gzip` compressed archives of older versions of those files may be found in the `/var/log` directory.

In general, no changes should be necessary to the `/etc/nginx/nginx.conf` configuration file. However, one scenario in which changes might be useful is if the server's default HTTPS configuration needs adjustment.

DEFAULT TLS CONFIGURATION

There are two reasons an administrator might need to know how to find the TLS configuration on Ansible Tower's TLS service. The first is to locate the TLS certificate and private key so that they can be replaced with versions signed by a Certificate Authority trusted by the browsers accessing Ansible Tower. The second would be if customization of the TLS configuration becomes necessary, particularly removing ciphers if vulnerabilities in their algorithms are found.

The TLS configuration for the Nginx web server is defined in the `/etc/nginx/nginx.conf` Nginx configuration file. The `server` block, which listens for SSL connections on port 443, contains the relevant configuration directives. In particular, this shows that the TLS certificate is `/etc/tower/tower.cert` and the matching private key is `/etc/tower/tower.key`:

```
server {
    listen 443 default_server ssl;
    listen 127.0.0.1:80 default_server;
    listen [::1]:80 default_server;

    # If you have a domain name, this is where to add it
    server_name _;
    keepalive_timeout 65;

    ssl_certificate /etc/tower/tower.cert;
    ssl_certificate_key /etc/tower/tower.key;
    ssl_session_cache shared:SSL:50m;
    ssl_session_timeout 1d;
    ssl_session_tickets off;
```

```
# intermediate configuration
ssl_protocols TLSv1.2;
ssl_ciphers 'ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-
ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES128-GCM-
SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256';
ssl_prefer_server_ciphers on;
```

REPLACING THE TLS CERTIFICATE AND KEY

Most organizations will want to replace the Ansible Tower self-signed certificate with one signed by a TLS Certificate Authority (CA) that is trusted by the organization's web browsers. This might be a public CA, or an internal corporate CA.

Either way, a correct CA-signed TLS certificate in PEM format needs to be obtained for the server, as well as a matching private key in PEM format. Assuming this has been done, they need to be used to replace the existing self-signed certificate and key as follows:

- Save the CA-signed TLS certificate in PEM format to `/etc/tower/tower.cert`.
- Save the matching private key in PEM format to `/etc/tower/tower.key`.
- Both files must be readable and writable only by the `awx` user (the Ansible Tower user), and must also be owned by the `awx` group:

```
[root@tower tower]# ls -l /etc/tower/tower.*
-rw----- 1 awx awx 1281 Mar 31 23:32 tower.cert
-rw----- 1 awx awx 1704 Mar 31 23:32 tower.key
```

- Use the `ansible-tower-service restart` command to restart Ansible Tower.
- Test the connection from a browser that trusts the CA used to sign the Ansible Tower server's certificate. Review the certificate details presented by your browser and whether your browser considers the connection secure. The details on how to do this will vary depending on which web browser you use.



IMPORTANT

In the lab, you will use the classroom FreeIPA server as the CA. This allows you to use `ipa-getcert` to request a TLS certificate, which will be automatically renewed by the `certmonger` daemon when it expires.

However, Ansible Tower 3.3.1 labels all files in `/etc/tower` with the SELinux type `etc_t`. Both `tower.cert` and `tower.key` need to be labeled `cert_t` for the files to be managed correctly by the FreeIPA tools. Fortunately, Nginx can read TLS certificates labeled `cert_t` correctly.

To persistently set the SELinux type on these two files, you need to make sure the `semanage` command is available. It is provided by the `policycoreutils-python` package. Run the `semanage fcontext -a -t cert_t "/etc/tower/tower.(.*)"` command to set the default context for those files in the system policy. Finally, run `restorecon -FvvR /etc/tower/` to correct the SELinux contexts in that directory based on the current policy settings.



REFERENCES

Ansible Tower Administration Guide

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

CONFIGURING TLS/SSL FOR ANSIBLE TOWER

In this exercise, you will replace the existing TLS/SSL certificate with a valid one provided by the **utility** server.

OUTCOMES

You should be able to:

- Replace the default TLS/SSL certificate used by Ansible Tower with a provided certificate appropriate for the server's host name.
- Ensure that the **workstation**, **tower**, and **utility** virtual machines are started.
- Log in to **workstation** as **student** using **student** as the password.
- On **workstation**, run **lab admin-cert setup**, which verifies that the Ansible Tower services are running and all the required resources are available.

```
[student@workstation ~]$ lab admin-cert setup
```

- 1. On **workstation**, open a terminal. In the following steps, you will replace the Ansible Tower TLS/SSL certificate with a provided certificate appropriate for the server's host name.
- 2. Use **ssh** to log in to the **tower** server as **root**.

```
[student@workstation ~]$ ssh root@tower
Last login: Wed Nov 21 06:27:17 2018 from workstation.lab.example.com
[root@tower ~]#
```

- 3. Ensure that the system policy will set the SELinux type to **cert_t** on **/etc/tower/tower.cert** and **/etc/tower/tower.key** in the system policy. Run **restorecon** on those files to make sure that the SELinux type is set on those files. This is needed for **certmonger** to write to those files when requested by **ipa-getcert** in the next step.

```
[root@tower ~]# semanage fcontext -a -t cert_t "/etc/tower(/.*)?"
[root@tower ~]# restorecon -FvvR /etc/tower/
```

- 4. Remove the existing **/etc/tower/tower.cert** and **/etc/tower/tower.key** files.

```
[root@tower ~]# rm /etc/tower/tower.*
rm: remove regular file '/etc/tower/tower.cert'? y
rm: remove regular file '/etc/tower/tower.key'? y
```

- ▶ 5. Using the **ipa-getcert** command, get a certificate for **tower.lab.example.com** that is signed by the organization's FreeIPA-based CA on **utility.lab.example.com**.

```
[root@tower ~]# ipa-getcert request -f /etc/tower/tower.cert \
> -k /etc/tower/tower.key
New signing request "20181121155831" added.
```



NOTE

Do not worry too much about why this works if you are not familiar with FreeIPA. The important part of this step is that you have a CA-signed TLS certificate and key for **tower.lab.example.com** that have been copied into the right locations on your Ansible Tower server.

- ▶ 6. Use the **ansible-tower-service** command to restart the Ansible Tower infrastructure and then exit the console session on the tower system.

```
[root@tower ~]# ansible-tower-service restart
Restarting Tower
Redirecting to /bin/systemctl stop postgresql-9.6.service
Stopping rabbitmq-server (via systemctl): [ OK ]
Redirecting to /bin/systemctl stop nginx.service
Redirecting to /bin/systemctl stop supervisord.service
Redirecting to /bin/systemctl start postgresql-9.6.service
Starting rabbitmq-server (via systemctl): [ OK ]
Redirecting to /bin/systemctl start nginx.service
Redirecting to /bin/systemctl start supervisord.service
[root@tower ~]# exit
```

- ▶ 7. Open Firefox and connect to the Ansible Tower web UI at <https://tower.lab.example.com>.

The new SSL certificate is signed by a trusted CA, which is why you do not receive any SSL certificate warnings. To review the new certificate details, click the padlock symbol in the browser address bar. The new TLS/SSL certificate includes the **tower.lab.example.com** hostname and the **LAB.EXAMPLE.COM** organization.

Cleanup

On workstation, run the **lab admin-cert cleanup** script to clean up this exercise.

```
[student@workstation ~]$ lab admin-cert cleanup
```

BACKING UP AND RESTORING ANSIBLE TOWER

OBJECTIVES

After completing this section, students should be able to:

- Back up and restore the Ansible Tower database and configuration files.

Figure 16.0: Backing up and restoring Ansible Tower

BACKING UP ANSIBLE TOWER

The ability to manually back up and restore a Red Hat Ansible Tower installation is integrated into Ansible Tower's installation software. You can then use other tools to automate the backup process and make sure that the backup files are stored in a safe and secure location separate from the Ansible Tower server.

The procedure uses the same **setup.sh** script and the **inventory** file that you used to install Ansible Tower.



IMPORTANT

If you have deleted the original installation directory, you can still set up backups by unpacking the **tar** archive containing the installer for the same version of Ansible Tower that you are using.

You also need to edit the installer's **inventory** file to contain the current passwords for your Ansible Tower services (`admin_password`, `pg_password`, and `rabbitmq_password`). If you made any other edits to the **inventory** file before installing Ansible Tower, you must make those edits now as well.

The actual backup is started by running `./setup.sh -b` in the installation directory on the Ansible Tower server as root. This creates the backup as a **tar** archive in the installer's directory named **tower-backup-*DATE*.tar.gz**, where *DATE* is in **date +%F-%T** format. It also creates a symlink, **tower-backup-latest.tar.gz**, pointing to the most recent backup archive in the directory.

The backup archive consists of the following files and directories:

- **tower.db**: PostgreSQL database dump file.
- **./conf**: The configuration directory, containing files from the **/etc/tower/** directory.
- **./job_status**: The directory for job output files.
- **./projects**: The directory for manual projects.
- **./static**: The directory for web UI customization, such as custom logos.

As with every backup procedure, you should review the amount of disk space available to ensure that there is enough free space to store the backup. Note that this procedure backs up the Ansible Tower configuration, but not its logs or the programs installed by the Ansible Tower installer.

**WARNING**

The manual backup procedure using **setup.sh -b** only creates the backup archive file. You are responsible for setting up a system that periodically runs the command to create the backups and store the backup archives in a safe place.

Backup Procedure

The following procedure creates a new backup of the running Ansible Tower configuration.

1. As the root user, locate the Ansible Tower installation directory and change into that directory.

```
[root@tower ~]# find / -name ansible*
/root/ansible-tower-setup-bundle-3.3.1-1.el7
[root@tower ~]# cd /root/ansible-tower-setup-bundle-3.3.1-1.el7
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]#
```

2. As the root user, run the **setup.sh** script with the **-b** option to initiate the Ansible Tower configuration and database backup process.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh -b
...output omitted...

RUNNING HANDLER [backup : Remove the backup tarball.] ****
changed: [localhost] => {"changed": true, "path": "/var/backups/tower/tower-
backup-2017-03-29-08:34:43.tar.gz", "state": "absent"}

PLAY RECAP ****
localhost : ok=24    changed=16    unreachable=0    failed=0

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2017-03-29-08:34:34.log
```

3. List the current directory to ensure that the backup archive has been created and is accessible.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ls -l tower-*
-rw-r--r--. 1 root root 164317 03-29 08:34 tower-
backup-2017-03-29-08:34:43.tar.gz
lrwxrwxrwx. 1 root root      84 03-29 08:35 tower-backup-latest.tar.gz -> /root/
ansible-tower-setup-bundle-3.3.1-1.el7/tower-backup-2017-03-29-08:34:43.tar.gz
```

**NOTE**

Notice the symbolic link **tower-backup-latest.tar.gz** pointing to the latest created backup archive. This link is by default used to recover the Ansible Tower infrastructure from backup.

RESTORING ANSIBLE TOWER FROM BACKUP

The **setup.sh** script is used with the **-r** option to restore Ansible Tower from a backup archive. You need the backup, the installer for the same version of Ansible Tower that was used to create the backup, and an **inventory** file for the installer.

Should you have multiple backup archives available, be sure that the **tower-backup-latest.tar.gz** symbolic link points to the exact backup file from which you want to restore. If you need to use an older backup file, delete the existing **tower-backup.latest.tar.gz** symbolic link and create a new link pointing to the correct backup archive.



WARNING

When restoring a backup, be sure to use the same version of Ansible Tower that was used to create the backup.

Restore Procedure

The following is the procedure for restoring the Ansible Tower infrastructure from existing backup archive.

- As the root user, locate the Ansible Tower installation directory and change into that directory.

```
[root@tower ~]# find / -name ansible*
/root/ansible-tower-setup-bundle-3.3.1-1.el7
[root@tower ~]# cd /root/ansible-tower-setup-bundle-3.3.1-1.el7
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]#
```

- Ensure that the backup archive is in that directory.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ls -l tower-
-rw-r--r--. 1 root root 164317 03-29 08:34 tower-
backup-2017-03-29-08:34:43.tar.gz
lrwxrwxrwx. 1 root root      84 03-29 08:35 tower-backup-latest.tar.gz -> /root/
ansible-tower-setup-bundle-3.3.1-1.el7/tower-backup-2017-03-29-08:34:43.tar.gz
```



IMPORTANT

Remember that the symbolic link **tower-backup-latest.tar.gz** points to the latest backup archive. This link is used to recover the Ansible Tower infrastructure from backup. If you need to restore from an older archive, you have to recreate that link by pointing to the correct archive.

- As the root user, run **setup.sh -r** to start restoring the Ansible Tower configuration and database.



IMPORTANT

Do not forget the **-r** option to the **setup.sh** command. Without this option, the Ansible Tower installer will start a new installation process from the beginning.

If that happens, wait until the installation process finishes, and then restore the backup.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh -r
...output omitted...
00ms", "Result": "success", "RootDirectoryStartOnly": "no",
"RuntimeDirectoryMode": "0755", "SameProcessGroup": "no", "SecureBits": "0",
"SendSIGHUP": "no", "SendSIGKILL": "yes", "Slice": "system.slice",
"StandardError": "inherit", "StandardInput": "null", "StandardOutput": "journal",
"StartLimitAction": "none", "StartLimitBurst": "5", "StartLimitInterval": "10000000",
"StartupBlockIOWeight": "18446744073709551615", "StartupCPUShares": "18446744073709551615",
>StatusErrno": "0", "StopWhenUnneeded": "no",
"SubState": "dead", "SyslogLevelPrefix": "yes", "SyslogPriority": "30",
"SystemCallErrorMessage": "0", "TTYReset": "no", "TTYVHangup": "no",
"TTYVTDisallocate": "no", "TimeoutStartUsec": "1min 30s", "TimeoutStopUsec": "1min 30s",
"TimerSlackNSec": "50000", "Transient": "no", "Type": "forking",
"UMask": "0022", "UnitFilePreset": "disabled", "UnitFileState": "enabled",
"WantedBy": "multi-user.target", "Wants": "system.slice",
"WatchdogTimestampMonotonic": "0", "WatchdogUsec": "0"}, "warnings": []}

PLAY RECAP ****
localhost : ok=26    changed=18    unreachable=0    failed=0

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2017-03-30-04:19:05.log
```

4. Log in to the Ansible Tower web UI and verify that the server has been restored from backup correctly.



REFERENCES

Ansible Tower Administration Guide

<https://docs.ansible.com/ansible-tower/latest/html/administration/>

► GUIDED EXERCISE

BACKING UP AND RESTORING ANSIBLE TOWER

In this exercise, you will back up the Red Hat Ansible Tower database and configuration files.

OUTCOMES

You should be able to:

- Back up the existing Red Hat Ansible Tower installation.
- Restore the Red Hat Ansible Tower configuration and database from an existing backup.

Ensure that the `workstation` and `tower` virtual machines are started.

Log in to `workstation` as `student` using `student` as the password.

On `workstation`, run `lab admin-recovery setup`, which verifies that the Ansible Tower services are running and all the required resources are available.

```
[student@workstation ~]$ lab admin-recovery setup
```

- ▶ 1. On `workstation`, open a terminal. In the following steps you will create a backup of Ansible Tower from the CLI.
- ▶ 2. To perform the backup, log in to the `tower` server as `root` user using SSH.

```
[student@workstation ~]$ ssh root@tower
Last login: Wed Nov 21 06:27:17 2018 from workstation.lab.example.com
[root@tower ~]#
```

- ▶ 3. List the contents of the `root` user home directory, to ensure that the installation directory of Ansible Tower is available.

```
[root@tower ~]# ls /root
anaconda-ks.cfg  ansible-tower-setup-bundle-3.3.1-1.el7  original-ks.cfg
```

- ▶ 4. Use the `ansible-tower-service status` command to ensure that the Ansible Tower services are running.

```
[root@tower ~]# ansible-tower-service status
(...output omitted...)
Redirecting to /bin/systemctl status supervisord.service
● supervisord.service - Process Monitoring and Control Daemon
    Loaded: loaded (/usr/lib/systemd/system/supervisord.service; enabled; vendor
    preset: disabled)
```

```
Active: active (running) since wto 2018-11-23 04:25:50 EDT; 2h 33min ago
Process: 4145 ExecStart=/usr/bin/supervisord -c /etc/supervisord.conf
          (code=exited, status=0/SUCCESS)
          (...output omitted...)
```

- 5. Change to the /root/ansible-tower-setup-bundle-3.3.1-1.el7 directory.

```
[root@tower ~]# cd /root/ansible-tower-setup-bundle-3.3.1-1.el7
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]#
```

- 6. Use the **setup.sh -b** command to back up the Ansible Tower configuration and database.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh -b
(...output omitted...)
RUNNING HANDLER [backup : Remove the backup directory.]
*****
changed: [localhost] => {"changed": true, "path": "/var/backups/tower/2018-11-29-13:38:56/", "state": "absent"}

RUNNING HANDLER [backup : Remove common directory.]
*****
changed: [localhost] => {"changed": true, "path": "/var/backups/tower/common/", "state": "absent"}

RUNNING HANDLER [backup : Remove the backup tarball.]
*****
changed: [localhost] => {"changed": true, "path": "/var/backups/tower/localhost.tar.gz", "state": "absent"}

RUNNING HANDLER [backup : Remove the common tarball.]
*****
changed: [localhost] => {"changed": true, "path": "/var/backups/tower/common.tar.gz", "state": "absent"}

RUNNING HANDLER [backup : Remove backup dest stage directory.]
*****
changed: [localhost] => {"changed": true, "path": "/root/ansible-tower-setup-bundle-3.3.1-1.el7/2018-11-29-13:38:56", "state": "absent"}

PLAY RECAP *****
localhost                  : ok=33    changed=25    unreachable=0    failed=0

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2018-11-23-07:11:21.log
```

- 7. Verify that the backup was created in the current working directory by issuing the **ls** command.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ls
backup.yml  install.yml           inventory_cluster  restore.yml
bundle      inventory             licenses            roles
tower-backup-2018-11-23-07:11:28.tar.gz
```

```
group_vars inventory.1521.2017-03-20@11:31:45~ README.md           setup.sh
      tower-backup-latest.tar.gz
```

- 8. Change the Ansible Tower `admin` superuser password to `redhat2` using the `awx-manage` command.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# awx-manage changepassword
  admin
Changing password for user 'admin'
Password: redhat2
Password (again): redhat2
Password changed successfully for user 'admin'
```

- 9. On workstation, open a browser and log in to Ansible Tower as the `admin` user using the new `redhat2` password. Then click the Log Out icon to log out of the Ansible Tower web UI.
- 10. Go back to `tower` server and restore the backup from the file created in previous steps. Use the `setup.sh` command with the `-r` as option. When the restore operation completes, exit the console session on the `tower` system.



IMPORTANT

Do not forget the `-r`, otherwise Ansible Tower will start the installation process from the beginning. If that happens, wait until the installation process finishes and then restore the backup.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh -r
(...output omitted...)
PLAY RECAP ****
localhost                  : ok=26    changed=18    unreachable=0    failed=0

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2018-11-23-11:53:32.log
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# exit
```

- 11. Verify that the backup has been restored by logging in to the Ansible Tower web UI as `admin` with the old `redhat` password. Then click the Log Out icon to log out of the Ansible Tower web UI.

Cleanup

On workstation, run the `lab admin-recovery cleanup` script to clean up this exercise.

```
[student@workstation ~]$ lab admin-recovery cleanup
```

► QUIZ

PERFORMING MAINTENANCE AND ROUTINE ADMINISTRATION OF ANSIBLE TOWER

Choose the correct answers to the following questions:

- ▶ 1. Which of the following commands can be used to determine the status of Ansible Tower services?
 - a. `firewall-cmd --list-services`
 - b. `systemctl status ansible-tower`
 - c. `ansible-tower-service status`
 - d. `service tower status`
- ▶ 2. Which of the following commands can be used to change the Red Hat Ansible Tower admin password?
 - a. `tower-manage changepassword admin`
 - b. `awx-manage createsuperuser`
 - c. `awx-manage changepassword admin`
 - d. `awx-manage update_password admin`
- ▶ 3. Where is the default location of Ansible Tower TLS/SSL certificate defined?
 - a. `/etc/tower/tower.cert`
 - b. `/etc/httpd/conf/ssl.conf`
 - c. `/etc/nginx/nginx.conf`
 - d. `/etc/nginx/conf.d/nginx.conf`
- ▶ 4. What is the default Ansible Tower log directory location?
 - a. `/etc/tower/logs/`
 - b. `/var/log/ansible/`
 - c. `/var/log/tower/`
 - d. `/var/awx/log/tower/`

► SOLUTION

PERFORMING MAINTENANCE AND ROUTINE ADMINISTRATION OF ANSIBLE TOWER

Choose the correct answers to the following questions:

- ▶ 1. Which of the following commands can be used to determine the status of Ansible Tower services?
 - a. `firewall-cmd --list-services`
 - b. `systemctl status ansible-tower`
 - c. `ansible-tower-service status`
 - d. `service tower status`

- ▶ 2. Which of the following commands can be used to change the Red Hat Ansible Tower admin password?
 - a. `tower-manage changepassword admin`
 - b. `awx-manage createsuperuser`
 - c. `awx-manage changepassword admin`
 - d. `awx-manage update_password admin`

- ▶ 3. Where is the default location of Ansible Tower TLS/SSL certificate defined?
 - a. `/etc/tower/tower.cert`
 - b. `/etc/httpd/conf/ssl.conf`
 - c. `/etc/nginx/nginx.conf`
 - d. `/etc/nginx/conf.d/nginx.conf`

- ▶ 4. What is the default Ansible Tower log directory location?
 - a. `/etc/tower/logs/`
 - b. `/var/log/ansible/`
 - c. `/var/log/tower/`
 - d. `/var/awx/log/tower/`

SUMMARY

In this chapter, you learned:

- Red Hat Ansible Tower integrates four main network services as its components: Nginx as the web server for the application, PostgreSQL as its database, `supervisord` as the process control system, and `rabbitmq-server` as an AMQP message broker for internal signaling.
- You should use the `ansible-tower-service` command instead of directly running `systemctl` when manually stopping, starting, or restarting Red Hat Ansible Tower services.
- The Ansible Tower application's configuration files are in `/etc/tower`.
- Log files for Ansible Tower are located in `/var/log/tower` and `/var/log/supervisor`.
- You can change the built-in `admin` user's password with `awx-manage changepassword admin`.
- The TLS certificate and private key for the Ansible Tower web server can be customized by replacing `/etc/tower/tower.cert` and `/etc/tower/tower.key`.
- You can back up the Ansible Tower database, configuration files, and local projects and job output with the installation script by running `setup.sh -b`.
- You can restore the backup files on a new server by putting the backup archive into the unpacked installation directory and running `setup.sh -r` to install and restore Ansible Tower.

CHAPTER 17

COMPREHENSIVE REVIEW: AUTOMATION WITH ANSIBLE AND ANSIBLE TOWER

OVERVIEW

GOAL Review tasks from *Automation with Ansible and Ansible Tower*

OBJECTIVES • Review tasks from *Automation with Ansible and Ansible Tower*

SECTIONS • Comprehensive Review

LAB • Deploying Ansible
• Creating Playbooks
• Creating Roles and Using Dynamic Inventory
• Restoring Ansible Tower from Backup
• Adding Users and Teams
• Creating a Custom Dynamic Inventory
• Configuring Job Templates
• Configuring Workflow Job Templates, Surveys, and Notifications
• Testing the Prepared Environment

► LAB

DEPLOYING ANSIBLE

In this review, you will install Ansible on **workstation** and use it as a control node and configure it for connections to the managed hosts **servera** and **serverb**. Use ad hoc commands to perform actions on managed hosts.

OUTCOMES

You should be able to:

- Install Ansible.
- Use ad hoc commands to perform actions on managed hosts.

Log in as the **student** user on **workstation** and run **lab review-deploy setup**. This script ensures that the managed hosts, **servera** and **serverb**, are reachable on the network. The script creates a lab subdirectory named **review-deploy** in the student's home directory.

```
[student@workstation ~]$ lab review-deploy setup
```

Instructions

Install and configure Ansible on **workstation**. Demonstrate that you can construct the ad hoc commands specified in the list of criteria in order to modify the managed hosts and verify that the modifications work as expected:

- Install Ansible on **workstation** so that it can serve as the control node.
- On the control node, create an inventory file, **/home/student/review-deploy/inventory**, containing a group called **dev**. This group should consist of the managed hosts **servera.lab.example.com** and **serverb.lab.example.com**.
- Create the Ansible configuration file in **/home/student/review-deploy/ansible.cfg**. The configuration file should point to the inventory file **/home/student/review-deploy/inventory**.
- Execute an ad hoc command using privilege escalation to modify the contents of the **/etc/motd** file on **servera** and **serverb** so that it contains the string **Managed by Ansible\n**. Use **devops** as the remote user.
- Execute an ad hoc command to verify that the contents of the **/etc/motd** file on **servera** and **serverb** are identical.

Evaluation

From **workstation**, run the **lab review-deploy** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

► SOLUTION

DEPLOYING ANSIBLE

In this review, you will install Ansible on **workstation** and use it as a control node and configure it for connections to the managed hosts **servera** and **serverb**. Use ad hoc commands to perform actions on managed hosts.

OUTCOMES

You should be able to:

- Install Ansible.
- Use ad hoc commands to perform actions on managed hosts.

Log in as the **student** user on **workstation** and run **lab review-deploy setup**. This script ensures that the managed hosts, **servera** and **serverb**, are reachable on the network. The script creates a lab subdirectory named **review-deploy** in the student's home directory.

```
[student@workstation ~]$ lab review-deploy setup
```

Instructions

Install and configure Ansible on **workstation**. Demonstrate that you can construct the ad hoc commands specified in the list of criteria in order to modify the managed hosts and verify that the modifications work as expected:

- Install Ansible on **workstation** so that it can serve as the control node.
 - On the control node, create an inventory file, **/home/student/review-deploy/inventory**, containing a group called **dev**. This group should consist of the managed hosts **servera.lab.example.com** and **serverb.lab.example.com**.
 - Create the Ansible configuration file in **/home/student/review-deploy/ansible.cfg**. The configuration file should point to the inventory file **/home/student/review-deploy/inventory**.
 - Execute an ad hoc command using privilege escalation to modify the contents of the **/etc/motd** file on **servera** and **serverb** so that it contains the string **Managed by Ansible\n**. Use **devops** as the remote user.
 - Execute an ad hoc command to verify that the contents of the **/etc/motd** file on **servera** and **serverb** are identical.
1. Install Ansible on **workstation** so that it can serve the control node.

```
[student@workstation ~]$ sudo yum install ansible
[sudo] password for student:
Loaded plugins: langpacks, search-disabled-repos
Resolving Dependencies
--> Running transaction check
---> Package ansible.noarch 0:2.7.1-1.el7ae will be installed
```

```
--> Processing Dependency: sshpass for package: ansible-2.7.1-1.el7ae.noarch
...output omitted...
Dependencies Resolved

=====
Package           Arch    Version      Repository   Size
=====
Installing:
ansible          noarch  2.7.1-1.el7ae  ansible      11 M
Installing for dependencies:
...output omitted...
Is this ok [y/d/N]: y
Downloading packages:
...output omitted...

Installed:
ansible.noarch 0:2.7.1-1.el7ae
...output omitted...
```

2. On the control node, create an inventory file, **/home/student/review-deploy/inventory**, containing a group called dev. This group should consist of the managed hosts servera.lab.example.com and serverb.lab.example.com.
- 2.1. Use the Vim text editor to create and edit the inventory file **/home/student/review-deploy/inventory**.

```
[student@workstation ~]$ cd /home/student/review-deploy
[student@workstation review-deploy]$ vim inventory
```

- 2.2. Add the following entries to the file to create the dev host group and members servera.lab.example.com and serverb.lab.example.com. Save the changes and exit the text editor.
- ```
[dev]
servera.lab.example.com
serverb.lab.example.com
```
3. Create the Ansible configuration file in **/home/student/review-deploy/ansible.cfg**. The configuration file should point to the inventory file **/home/student/review-deploy/inventory**.
- 3.1. Use the Vim text editor to create and edit the ansible configuration file **/home/student/review-deploy/ansible.cfg**.

```
[student@workstation review-deploy]$ vim ansible.cfg
```

- 3.2. Add the following entries to configure the inventory file **./inventory** as the inventory source. Save the changes and exit the text editor.

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

4. Execute an ad hoc command using privilege escalation to modify the contents of the `/etc/motd` file on `servera` and `serverb` so that it contains the string `Managed by Ansible\n`. Use `devops` as the remote user.

- 4.1. From the project directory `/home/student/review-deploy`, execute an ad hoc command using privilege escalation to modify the contents of the `/etc/motd` file on `servera` and `serverb`, so that it contains the string `Managed by Ansible\n`. Use `devops` as the remote user.

```
[student@workstation review-deploy]$ ansible dev -m copy \
> -a 'content="Managed by Ansible\n" dest=/etc/motd' -b -u devops
servera.lab.example.com | CHANGED => {
 "changed": true,
 "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
 "dest": "/etc/motd",
 "gid": 0,
 "group": "root",
 "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
 "mode": "0644",
 "owner": "root",
 "secontext": "system_u:object_r:etc_t:s0",
 "size": 19,
 "src": "/home/devops/.ansible/tmp/...output omitted...",
 "state": "file",
 "uid": 0
}
serverb.lab.example.com | CHANGED => {
 "changed": true,
 "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
 "dest": "/etc/motd",
 "gid": 0,
 "group": "root",
 "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
 "mode": "0644",
 "owner": "root",
 "secontext": "system_u:object_r:etc_t:s0",
 "size": 19,
 "src": "/home/devops/.ansible/tmp/...output omitted...",
 "state": "file",
 "uid": 0
}
```

5. Execute an ad hoc command to verify that the contents of the `/etc/motd` file on `servera` and `serverb` are identical.

```
[student@workstation review-deploy]$ ansible dev -m command -a "cat /etc/motd"
servera.lab.example.com | CHANGED | rc=0 >>
Managed by Ansible

serverb.lab.example.com | CHANGED | rc=0 >>
Managed by Ansible
```

## Evaluation

From workstation, run the **lab review-deploy** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

## ▶ LAB

# CREATING PLAYBOOKS

In this review, you will create three playbooks in the Ansible project directory, `/home/student/review-playbooks`. One playbook will ensure that `lftp` is installed on systems that should be FTP clients, one playbook will ensure that `vsftpd` is installed and configured on systems that should be FTP servers, and one playbook (`site.yml`) will run both of the other playbooks.

## OUTCOMES

You should be able to:

- Create and execute playbooks to perform tasks on managed hosts.
- Utilize Jinja2 templates, variables, and handlers in playbooks.

Set up your computers for this exercise by logging into `workstation` as `student`, and run the following command:

```
[student@workstation ~]$ lab review-playbooks setup
```

## Instructions

Create a static inventory in `review-playbooks/inventory` with `serverc.lab.example.com` in the group `ftpclients`, and `serverb.lab.example.com` and `serverd.lab.example.com` in the group `ftpservers`. Create an `review-playbooks/ansible.cfg` file which configures your Ansible project to use this inventory. You may find it useful to look at the system's `/etc/ansible/ansible.cfg` file for help with syntax.

Configure your Ansible project to connect to hosts in the inventory using the remote user, `devops`, and the `sudo` method for privilege escalation. You have SSH keys to log in as `devops` already configured. The `devops` user does not need a password for privilege escalation with `sudo`.

Create a playbook named `ftpclients.yml` in the `review-playbooks` directory that contains a play targeting hosts in the inventory group `ftpclients`. It should make sure the `lftp` package is installed.

Create a second playbook named `ansible-vsftpd.yml` in the `review-playbooks` directory that contains a play targeting hosts in the inventory group `ftpservers`. It should be written as follows:

- You have a configuration file for `vsftpd` generated from a Jinja2 template. Create a directory for templates, `review-playbooks/templates`, and copy the provided `vsftpd.conf.j2` file into it. Also create the directory `review-playbooks/vars`. Copy into that directory the provided `defaults-template.yml` file, which contains default variable settings used to complete that template when it is deployed.
- Create a variable file, `review-playbooks/vars/vars.yml`, that sets three variables:

| VARIABLE           | VALUE                   |
|--------------------|-------------------------|
| vsftpd_package     | vsftpd                  |
| vsftpd_service     | vsftpd                  |
| vsftpd_config_file | /etc/vsftpd/vsftpd.conf |

- In your **ansible-vsftpd.yml** playbook, make sure that you use **vars\_files** to include the files of variables in the **review-playbooks/vars** directory in your play.
- In the play in **ansible-vsftpd.yml**, create tasks which:
  1. Ensure that the package listed by the variable `{{ vsftpd_package }}` is installed.
  2. Ensure that the service listed by the variable `{{ vsftpd_service }}` is started and enabled to start at boot time.
  3. Use the template module to deploy the **templates/vsftpd.conf.j2** template to the location defined by the `{{ vsftpd_config_file }}` variable. The file should be owned by user `root`, group `root`, have octal file permissions `0600`, and an SELinux type of `etc_t`. Notify a handler that restarts **vsftpd** if this task cause a change.
  4. Ensure that the `firewalld` package is installed and that the service is started and enabled. Ensure that `firewalld` has been configured to immediately and permanently allow connections to the ftp service.
- In your **ansible-vsftpd.yml** playbook, create a handler to restart the services listed by the variable `{{ vsftpd_service }}` when notified.

Create a third playbook, **site.yml**, in the **review-playbooks** directory. This playbook should only import the other two playbooks.

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

Remember to use the **ansible-doc** command to help you find modules and information on how to use them.

When done, you should use **ansible-playbook site.yml** to check your work before running the grading script. You may also run the individual playbooks separately to make sure that they function.



### IMPORTANT

If you are having trouble with your **site.yml** playbook, make sure that both **ansible-vsftpd.yml** and **ftpclients.yml** have indentation consistent with each other.

## Evaluation

As the student user on `workstation`, run the **lab review-playbooks grade** command to confirm success of this exercise. Correct any reported failures and rerun the script until successful.

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

## Cleanup

Run the **lab review-playbooks cleanup** command to clean up the lab tasks on serverb, serverc, and serverd.

```
[student@workstation ~]$ lab review-playbooks cleanup
```

## ► SOLUTION

# CREATING PLAYBOOKS

In this review, you will create three playbooks in the Ansible project directory, `/home/student/review-playbooks`. One playbook will ensure that `lftp` is installed on systems that should be FTP clients, one playbook will ensure that `vsftpd` is installed and configured on systems that should be FTP servers, and one playbook (`site.yml`) will run both of the other playbooks.

### OUTCOMES

You should be able to:

- Create and execute playbooks to perform tasks on managed hosts.
- Utilize Jinja2 templates, variables, and handlers in playbooks.

Set up your computers for this exercise by logging into `workstation` as `student`, and run the following command:

```
[student@workstation ~]$ lab review-playbooks setup
```

### Instructions

Create a static inventory in `review-playbooks/inventory` with `serverc.lab.example.com` in the group `ftpclients`, and `serverb.lab.example.com` and `serverd.lab.example.com` in the group `ftpservers`. Create an `review-playbooks/ansible.cfg` file which configures your Ansible project to use this inventory. You may find it useful to look at the system's `/etc/ansible/ansible.cfg` file for help with syntax.

Configure your Ansible project to connect to hosts in the inventory using the remote user, `devops`, and the `sudo` method for privilege escalation. You have SSH keys to log in as `devops` already configured. The `devops` user does not need a password for privilege escalation with `sudo`.

Create a playbook named `ftpclients.yml` in the `review-playbooks` directory that contains a play targeting hosts in the inventory group `ftpclients`. It should make sure the `lftp` package is installed.

Create a second playbook named `ansible-vsftpd.yml` in the `review-playbooks` directory that contains a play targeting hosts in the inventory group `ftpservers`. It should be written as follows:

- You have a configuration file for `vsftpd` generated from a Jinja2 template. Create a directory for templates, `review-playbooks/templates`, and copy the provided `vsftpd.conf.j2` file into it. Also create the directory `review-playbooks/vars`. Copy into that directory the provided `defaults-template.yml` file, which contains default variable settings used to complete that template when it is deployed.
- Create a variable file, `review-playbooks/vars/vars.yml`, that sets three variables:

| VARIABLE           | VALUE                   |
|--------------------|-------------------------|
| vsftpd_package     | vsftpd                  |
| vsftpd_service     | vsftpd                  |
| vsftpd_config_file | /etc/vsftpd/vsftpd.conf |

- In your **ansible-vsftpd.yml** playbook, make sure that you use **vars\_files** to include the files of variables in the **review-playbooks/vars** directory in your play.
- In the play in **ansible-vsftpd.yml**, create tasks which:
  1. Ensure that the package listed by the variable `{{ vsftpd_package }}` is installed.
  2. Ensure that the service listed by the variable `{{ vsftpd_service }}` is started and enabled to start at boot time.
  3. Use the **template** module to deploy the **templates/vsftpd.conf.j2** template to the location defined by the `{{ vsftpd_config_file }}` variable. The file should be owned by user **root**, group **root**, have octal file permissions **0600**, and an SELinux type of **etc\_t**. Notify a handler that restarts **vsftpd** if this task cause a change.
  4. Ensure that the **firewalld** package is installed and that the service is started and enabled. Ensure that **firewalld** has been configured to immediately and permanently allow connections to the ftp service.
- In your **ansible-vsftpd.yml** playbook, create a handler to restart the services listed by the variable `{{ vsftpd_service }}` when notified.

Create a third playbook, **site.yml**, in the **review-playbooks** directory. This playbook should only import the other two playbooks.

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

Remember to use the **ansible-doc** command to help you find modules and information on how to use them.

When done, you should use **ansible-playbook site.yml** to check your work before running the grading script. You may also run the individual playbooks separately to make sure that they function.



### IMPORTANT

If you are having trouble with your **site.yml** playbook, make sure that both **ansible-vsftpd.yml** and **ftpclients.yml** have indentation consistent with each other.

1. As the student user on workstation, create the inventory file **/home/student/review-playbooks/inventory**, containing `serverc.lab.example.com` in the group **ftpclients**, and `serverb.lab.example.com` and `serverd.lab.example.com` in the group **ftpservers**.
  - 1.1. Change directory into the Ansible project directory, **/home/student/review-playbooks**, created by the setup script.

```
[student@workstation ~]$ cd /home/student/review-playbooks
```

- 1.2. Create the static inventory file, **inventory**, by opening it with a text editor.

```
[student@workstation review-playbooks]$ vim inventory
```

- 1.3. Populate the **inventory** file with the following contents:

```
[ftpservers]
serverb.lab.example.com
serverd.lab.example.com

[ftpclients]
serverc.lab.example.com
```

- 1.4. Save the changes to the newly created inventory file.

2. Create the Ansible configuration file, **/home/student/review-playbooks/ansible.cfg**, and populate it with the necessary entries to meet these requirements:

- Configure the Ansible project to use the newly created inventory
- Connect to managed hosts as the devops user
- Utilize privilege escalation using **sudo** as the **root** user
- Escalate privileges for each task by default

- 2.1. Create the Ansible configuration file, **/home/student/review-playbooks/ansible.cfg**, by opening it with a text editor.

```
[student@workstation review-playbooks]$ vim ansible.cfg
```

- 2.2. Configure the inventory, remote user, and privilege escalation method and user for the Ansible project by adding the following entries in the **ansible.cfg** configuration file.

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

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

- 2.3. Save the changes to the newly created Ansible configuration file.

3. Create the playbook, **/home/student/review-playbooks/ftpclients.yml**, containing a play targeting the hosts in the `ftpclients` inventory group and ensures that the `lftp` is installed.
  - 3.1. Create the playbook file, **/home/student/review-playbooks/ftpclients.yml**, by opening it with a text editor.

```
[student@workstation review-playbooks]$ vim ftpclients.yml
```

- 3.2. Populate the new playbook file with a play to ensure that the `lftp` package is installed on the hosts in the `ftpclients` inventory group by adding the following entries.

```

- name: Ensure FTP Client Configuration
 hosts: ftpclients

 tasks:
 - name: latest version of lftp is installed
 yum:
 name: lftp
 state: latest
```

- 3.3. Save the changes to the newly created playbook file.
4. Place the provided vsftpd configuration file in the **templates** subdirectory.
  - 4.1. Create the **templates** subdirectory.

```
[student@workstation review-playbooks]$ mkdir -v templates
mkdir: created directory 'templates'
```

- 4.2. Move the **vsftpd.conf.j2** file to the newly created **templates** subdirectory.
5. Place the provided **defaults-template.yml** file in the **vars** subdirectory.

- 5.1. Create the **vars** subdirectory.

```
[student@workstation review-playbooks]$ mkdir -v vars
mkdir: created directory 'vars'
```

- 5.2. Move the **defaults-template.yml** file to the newly created **vars** subdirectory.

```
[student@workstation review-playbooks]$ mv -v defaults-template.yml vars
'defaults-template.yml' -> 'vars/defaults-template.yml'
```

6. Create a **vars.yml** variable definition file in the **vars** subdirectory to define the following three variables and their values.

| VARIABLE           | VALUE                   |
|--------------------|-------------------------|
| vsftpd_package     | vsftpd                  |
| vsftpd_service     | vsftpd                  |
| vsftpd_config_file | /etc/vsftpd/vsftpd.conf |

- 6.1. Create the **/home/student/review-playbooks/vars/vars.yml** file.

```
[student@workstation review-playbooks]$ vim vars/vars.yml
```

- 6.2. Populate the **vars.yml** file with the following variable definitions.

```
vsftpd_package: vsftpd
vsftpd_service: vsftpd
vsftpd_config_file: /etc/vsftpd/vsftpd.conf
```

- 6.3. Save the changes to the newly created variable definition file.

7. Using the previously created Jinja2 template and variable definition files, create a second playbook, **/home/student/review-playbooks/ansible-vsftpd.yml**, to configure the vsftpd service on the hosts in the **ftpservers** inventory group.

- 7.1. Create the playbook file, **/home/student/review-playbooks/ansible-vsftpd.yml** by opening it with a text editor.

```
[student@workstation review-playbooks]$ vim ansible-vsftpd.yml
```

- 7.2. Populate the new playbook file with the following entries in order to configure the vsftpd service on the hosts in the **ftpservers** inventory group.

```

- name: FTP server is installed
 hosts:
 - ftpservers
 vars_files:
 - vars/defaults-template.yml
 - vars/vars.yml

 tasks:
 - name: Packages are installed
 yum:
 name: '{{ vsftpd_package }}'
 state: present

 - name: Ensure service is started
 service:
 name: '{{ vsftpd_service }}'
 state: started
 enabled: true
```

```

- name: Configuration file is installed
 template:
 src: templates/vsftpd.conf.j2
 dest: '{{ vsftpd_config_file }}'
 owner: root
 group: root
 mode: '0600'
 setype: etc_t
 notify: restart vsftpd

- name: firewalld is installed
 yum:
 name: firewalld
 state: present

- name: firewalld is started and enabled
 service:
 name: firewalld
 state: started
 enabled: yes

- name: FTP port is open
 firewalld:
 service: ftp
 permanent: true
 state: enabled
 immediate: yes

handlers:
- name: restart vsftpd
 service:
 name: "{{ vsftpd_service }}"
 state: restarted

```

7.3. Save the changes to the newly created playbook file.

8. Create a third playbook, **/home/student/review-playbooks/site.yml**, and include the plays from the two playbooks created previously, **ftpclients.yml** and **ansible-vsftpd.yml**.

8.1. Create the playbook file, **/home/student/review-playbooks/site.yml**, by opening it with a text editor.

```
[student@workstation review-playbooks]$ vim site.yml
```

8.2. Populate the new playbook file with the following entries in order to include the plays from the other two playbooks.

```

FTP Servers playbook
- import_playbook: ansible-vsftpd.yml

FTP Clients playbook

```

```
- import_playbook: ftpclients.yml
```

8.3. Save the changes to the newly created playbook file.

9. Execute the **/home/student/review-playbooks/site.yml** playbook to verify that it performs the desired tasks on the managed hosts.

```
[student@workstation review-playbooks]$ ansible-playbook site.yml
```

## Evaluation

As the student user on workstation, run the **lab review-playbooks grade** command to confirm success of this exercise. Correct any reported failures and rerun the script until successful.

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

## Cleanup

Run the **lab review-playbooks cleanup** command to clean up the lab tasks on serverb, serverc, and serverd.

```
[student@workstation ~]$ lab review-playbooks cleanup
```

## ► LAB

# CREATING ROLES AND USING DYNAMIC INVENTORY

In this review, you will convert the **ansible-vsftpd.yml** playbook into a role, and then use that role in a new playbook that will also run some additional tasks. You will also install and use a dynamic inventory script, which will be provided to you.

## OUTCOMES

You should be able to:

- Create a role to configure the vsftpd service using tasks from an existing playbook.
- Include a role in a playbook, and execute the playbook.
- Use a dynamic inventory when executing a playbook.

Log in to workstation as student using student as the password.

On workstation, run the **lab review-roles setup** script. This script ensures that the remote hosts are reachable on the network. The script also checks that Ansible is installed on workstation, creates a directory structure for the lab environment, and installs required lab files.

```
[student@workstation ~]$ lab review-roles setup
```

## Instructions

Configure your Ansible project to use the dynamic inventory script **crlinventory.py** and the static inventory file **inventory**.

Convert the **ansible-vsftpd.yml** playbook into the role **ansible-vsftpd**, as specified below:

- Use the **ansible-galaxy** command to create the directory structure for the role **ansible-vsftpd** in the **review-roles/roles** directory of your Ansible project.
- The file **defaults-template.yml** contains default variables for the role. It should be moved to an appropriate location in the role directory structure.
- The file **vars.yml** contains regular variables for the role. It should be moved to an appropriate location in the role directory structure.
- The template **vsftpd.conf.j2** should be moved to an appropriate location in the role directory structure.
- The tasks and handlers in the **ansible-vsftpd.yml** playbook should be appropriately installed in the role.
- You may edit the role's **meta/main.yml** file to set the author, description, and license fields (use BSD for the license). You may also edit the **README.md** file as you wish for completeness.
- Remove any subdirectories in the role that you are not using.

Create a new playbook, **vsftpd-configure.yml**, in the **review-roles** directory. It should be written as follows:

- It should contain a play targeting hosts in the inventory group **ftpservers**.
- The play should set the following variables:

| VARIABLE          | VALUE      |
|-------------------|------------|
| vsftpd_anon_root  | /mnt/share |
| vsftpd_local_root | /mnt/share |

- The play should apply the role **ansible-vsftpd**.
- The play should include the following tasks in the specified order:
  1. Use the **command** module to create a GPT disk label on **/dev/vdb**, that starts 1 MiB from the beginning of the device and ends at the end of the device. Use the **ansible-doc** command to learn how to use the **creates** argument to skip this task if **/dev/vdb1** has already been created. This avoids destructive repartitioning of the device. Use the following command to create the partition: **parted --script /dev/vdb mklabel gpt mkpart primary 1MiB 100%**
  2. Ensure a **/mnt/share** directory exists for use as a mount point.
  3. Use **ansible-doc -l** to find a module that can make a file system on a block device. Use **ansible-doc** to learn how to use that module. Add a task to the playbook that uses it to create an XFS file system on **/dev/vdb1**. Do not force creation of that file system if one exists already.
  4. Add a task to ensure that **/etc/fstab** mounts the device **/dev/vdb1** on **/mnt/share** at boot, and that it is currently mounted. (Use **ansible-doc** to find a module that can help with this.) If this task changes, notify the **ansible-vsftpd** role's handler that restarts vsftpd.
  5. Add a task that ensures that the **/mnt/share** directory is owned by the **root** user and the **root** group, has the SELinux type defined in the **{{ vsftpd\_setype }}** variable from the role, and has octal permissions of 0755. This has to be done after the file system is mounted to set the permissions on the mounted file system and not on the placeholder mount point directory.
  6. Make sure that a file named **README** exists in the directory specified by **{{ vsftpd\_anon\_root }}** containing the string **Welcome to the FTP server at serverX.lab.example.com** where **serverX.lab.example.com** is the actual fully-qualified hostname for that server. This file should have octal permissions of 0644 and the SELinux type specified by the **{{ vsftpd\_setype }}** variable. (Hint: look at the **copy** or **template** modules and the available Ansible facts in order to solve this problem.)

**IMPORTANT**

You may find it useful to debug your role by testing it in a playbook that does not contain the extra tasks or playbook variables listed above, but only contains a play that targets hosts in the group `ftpservers`, and applies the role.

Once you have confirmed that a simplified playbook using only the role works just like the original `ansible-vsftpd.yml` playbook, you can build the complete `vsftpd-configure.yml` playbook by adding the additional variables and tasks specified above.

Change the `review-roles/site.yml` playbook to use the new `vsftpd-configure.yml` playbook instead of `ansible-vsftpd.yml`.

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

When done, use `ansible-playbook site.yml` to check your work before running the grading script. You may also run the individual playbooks separately to make sure they function.

**IMPORTANT**

If you are having trouble with your `site.yml` playbook, make sure that both `vsftpd-configure.yml` and `ftpclients.yml` have indentation consistent with each other.

## Evaluation

From workstation, run the `lab review-roles grade` command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

## Cleanup

Run the `lab review-roles cleanup` command to clean up the lab tasks on servera and serverb.

```
[student@workstation ~]$ lab review-roles cleanup
```

## ► SOLUTION

# CREATING ROLES AND USING DYNAMIC INVENTORY

In this review, you will convert the **ansible-vsftpd.yml** playbook into a role, and then use that role in a new playbook that will also run some additional tasks. You will also install and use a dynamic inventory script, which will be provided to you.

## OUTCOMES

You should be able to:

- Create a role to configure the vsftpd service using tasks from an existing playbook.
- Include a role in a playbook, and execute the playbook.
- Use a dynamic inventory when executing a playbook.

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab review-roles setup** script. This script ensures that the remote hosts are reachable on the network. The script also checks that Ansible is installed on **workstation**, creates a directory structure for the lab environment, and installs required lab files.

```
[student@workstation ~]$ lab review-roles setup
```

## Instructions

Configure your Ansible project to use the dynamic inventory script **cinventory.py** and the static inventory file **inventory**.

Convert the **ansible-vsftpd.yml** playbook into the role **ansible-vsftpd**, as specified below:

- Use the **ansible-galaxy** command to create the directory structure for the role **ansible-vsftpd** in the **review-roles/roles** directory of your Ansible project.
- The file **defaults-template.yml** contains default variables for the role. It should be moved to an appropriate location in the role directory structure.
- The file **vars.yml** contains regular variables for the role. It should be moved to an appropriate location in the role directory structure.
- The template **vsftpd.conf.j2** should be moved to an appropriate location in the role directory structure.
- The tasks and handlers in the **ansible-vsftpd.yml** playbook should be appropriately installed in the role.
- You may edit the role's **meta/main.yml** file to set the author, description, and license fields (use BSD for the license). You may also edit the **README.md** file as you wish for completeness.
- Remove any subdirectories in the role that you are not using.

Create a new playbook, **vsftpd-configure.yml**, in the **review-roles** directory. It should be written as follows:

- It should contain a play targeting hosts in the inventory group **ftpservers**.
- The play should set the following variables:

| VARIABLE                       | VALUE                   |
|--------------------------------|-------------------------|
| <code>vsftpd_anon_root</code>  | <code>/mnt/share</code> |
| <code>vsftpd_local_root</code> | <code>/mnt/share</code> |

- The play should apply the role **ansible-vsftpd**.
- The play should include the following tasks in the specified order:
  1. Use the **command** module to create a GPT disk label on **/dev/vdb**, that starts 1 MiB from the beginning of the device and ends at the end of the device. Use the **ansible-doc** command to learn how to use the **creates** argument to skip this task if **/dev/vdb1** has already been created. This avoids destructive repartitioning of the device. Use the following command to create the partition: **parted --script /dev/vdb mklabel gpt mkpart primary 1MiB 100%**
  2. Ensure a **/mnt/share** directory exists for use as a mount point.
  3. Use **ansible-doc -l** to find a module that can make a file system on a block device. Use **ansible-doc** to learn how to use that module. Add a task to the playbook that uses it to create an XFS file system on **/dev/vdb1**. Do not force creation of that file system if one exists already.
  4. Add a task to ensure that **/etc/fstab** mounts the device **/dev/vdb1** on **/mnt/share** at boot, and that it is currently mounted. (Use **ansible-doc** to find a module that can help with this.) If this task changes, notify the **ansible-vsftpd** role's handler that restarts vsftpd.
  5. Add a task that ensures that the **/mnt/share** directory is owned by the **root** user and the **root** group, has the SELinux type defined in the `{{ vsftpd_setype }}` variable from the role, and has octal permissions of 0755. This has to be done after the file system is mounted to set the permissions on the mounted file system and not on the placeholder mount point directory.
  6. Make sure that a file named **README** exists in the directory specified by `{{ vsftpd_anon_root }}` containing the string **Welcome to the FTP server at serverX.lab.example.com** where `serverX.lab.example.com` is the actual fully-qualified hostname for that server. This file should have octal permissions of 0644 and the SELinux type specified by the `{{ vsftpd_setype }}` variable. (Hint: look at the **copy** or **template** modules and the available Ansible facts in order to solve this problem.)

**IMPORTANT**

You may find it useful to debug your role by testing it in a playbook that does not contain the extra tasks or playbook variables listed above, but only contains a play that targets hosts in the group `ftpservers`, and applies the role.

Once you have confirmed that a simplified playbook using only the role works just like the original `ansible-vsftpd.yml` playbook, you can build the complete `vsftpd-configure.yml` playbook by adding the additional variables and tasks specified above.

Change the `review-roles/site.yml` playbook to use the new `vsftpd-configure.yml` playbook instead of `ansible-vsftpd.yml`.

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

When done, use `ansible-playbook site.yml` to check your work before running the grading script. You may also run the individual playbooks separately to make sure they function.

**IMPORTANT**

If you are having trouble with your `site.yml` playbook, make sure that both `vsftpd-configure.yml` and `ftpclients.yml` have indentation consistent with each other.

1. Log in to your workstation host as `student`. Change to the `review-roles` working directory.

```
[student@workstation ~]$ cd ~/review-roles
[student@workstation review-roles]$
```

2. Configure the Ansible project to use both the dynamic inventory file `crinventory.py` as well as the static inventory file `inventory`. Verify the inventory configuration using the `ansible-inventory` command.

- 2.1. Place the static inventory file in a directory named `inventory`.

```
[student@workstation review-roles]$ mv -v inventory inventory.tmp
'inventory' -> 'inventory.tmp'
[student@workstation review-roles]$ mkdir -v inventory
mkdir: created directory 'inventory'
[student@workstation review-roles]$ mv -v inventory.tmp inventory/inventory
'inventory.tmp' -> 'inventory/inventory'
```

- 2.2. Add the dynamic inventory script to the inventory directory. Ensure that the script is executable.

```
[student@workstation review-roles]$ chmod 755 -v crinventory.py
mode of 'crinventory.py' changed from 0644 (rw-r--r--) to 0755 (rwxr-xr-x)
[student@workstation review-roles]$ mv -v crinventory.py inventory
'crinventory.py' -> 'inventory/crinventory.py'
```

2.3. Configure the **inventory** directory as the source of inventory files for the project.

The **[defaults]** section of the **ansible.cfg** file looks like this:

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

2.4. Use the **ansible-inventory** command to verify the project inventory configuration:

```
[student@workstation review-roles]$ ansible-inventory --list all
{
 "_meta": {
 "hostvars": {
 "servera.lab.example.com": {},
 "serverb.lab.example.com": {},
 "serverc.lab.example.com": {},
 "serverd.lab.example.com": {}
 }
 },
 "all": {
 "children": [
 "ftpclients",
 "ftpservers",
 "ungrouped"
]
 },
 "ftpclients": {
 "hosts": [
 "servera.lab.example.com",
 "serverc.lab.example.com"
]
 },
 "ftpservers": {
 "hosts": [
 "serverb.lab.example.com",
 "serverd.lab.example.com"
]
 },
 "ungrouped": {}
}
```

3. Convert the **ansible-vsftpd.yml** playbook into the role **ansible-vsftpd**.

3.1. Create the **roles** subdirectory.

```
[student@workstation review-roles]$ mkdir -v roles
mkdir: created directory 'roles'
```

3.2. Using **ansible-galaxy**, create the directory structure for the new **ansible-vsftpd** role in the **roles** subdirectory.

```
[student@workstation review-roles]$ cd roles
[student@workstation roles]$ ansible-galaxy init ansible-vsftpd
```

```
- ansible-vsftpd was created successfully
[student@workstation roles]$ cd ..
[student@workstation review-roles]$
```

- 3.3. Using **tree**, verify the directory structure created for the new role.

```
[student@workstation review-roles]$ tree roles
roles
└── ansible-vsftpd
 ├── defaults
 │ └── main.yml
 ├── files
 ├── handlers
 │ └── main.yml
 ├── meta
 │ └── main.yml
 ├── README.md
 ├── tasks
 │ └── main.yml
 ├── templates
 ├── tests
 │ ├── inventory
 │ └── test.yml
 └── vars
 └── main.yml

9 directories, 8 files
```

- 3.4. Replace the **roles/ansible-vsftpd/defaults/main.yml** file with the variable definitions in the **defaults-template.yml** file.

```
[student@workstation review-roles]$ mv -v defaults-template.yml \
> roles/ansible-vsftpd/defaults/main.yml
'defaults-template.yml' -> 'roles/ansible-vsftpd/defaults/main.yml'
```

- 3.5. Replace the **roles/ansible-vsftpd/vars/main.yml** file with the variable definitions in the **vars.yml** file.

```
[student@workstation review-roles]$ mv -v vars.yml \
> roles/ansible-vsftpd/vars/main.yml
'vars.yml' -> 'roles/ansible-vsftpd/vars/main.yml'
```

- 3.6. Use the **templates/vsftpd.conf.j2** file as a template for the **ansible-vsftpd** role.

```
[student@workstation review-roles]$ mv -v vsftpd.conf.j2 \
> roles/ansible-vsftpd/templates/
'vsftpd.conf.j2' -> 'roles/ansible-vsftpd/templates/vsftpd.conf.j2'
```

- 3.7. Copy the tasks in the **ansible-vsftpd.yml** playbook into the **roles/ansible-vsftpd/tasks/main.yml** file. The value of the **src** keyword in the template module task no longer needs to reference the **templates** subdirectory. The **roles/**

**ansible-vsftpd/tasks/main.yml** file should have the following contents after you are done.

```

tasks file for ansible-vsftpd
- name: Packages are installed
 yum:
 name: '{{ vsftpd_package }}'
 state: present

- name: Ensure service is started
 service:
 name: '{{ vsftpd_service }}'
 state: started
 enabled: true

- name: Configuration file is installed
 template:
 src: vsftpd.conf.j2
 dest: '{{ vsftpd_config_file }}'
 owner: root
 group: root
 mode: '0600'
 setype: etc_t
 notify: restart vsftpd

- name: firewalld is installed
 yum:
 name: firewalld
 state: present

- name: firewalld is started and enabled
 service:
 name: firewalld
 state: started
 enabled: yes

- name: FTP port is open
 firewalld:
 service: ftp
 permanent: true
 state: enabled
 immediate: yes
```

- 3.8. Copy the handlers in the **ansible-vsftpd.yml** playbook into the **roles/ansible-vsftpd/handlers/main.yml** file. The **roles/ansible-vsftpd/handlers/main.yml** file should have the following contents after you are done.

```

handlers file for ansible-vsftpd
- name: restart vsftpd
 service:
 name: "{{ vsftpd_service }}"
 state: restarted
```

4. Update the contents of the **roles/ansible-vsftpd/meta/main.yml** file.

- 4.1. Change the value of the author entry to **Red Hat Training**.

```
author: Red Hat Training
```

- 4.2. Change the value of the **description** entry to "**example role for D0410**".

```
description: example role for D0410
```

- 4.3. Change the value of the **company** entry to "**Red Hat**".

```
company: Red Hat
```

- 4.4. Change the value of the **license:** entry to "**BSD**".

```
license: BSD
```

5. Modify the contents of the **roles/ansible-vsftpd/README.md** file so that it provides pertinent information regarding the role. After modification, the file should contain the following contents.

```
ansible-vsftpd
```

```
=====
```

```
Example ansible-vsftpd role from Red Hat's "Automation with Ansible and Ansible Tower" (D0410)
course.
```

```
Role Variables
```

```

```

```
* defaults/main.yml contains variables used to configure the vsftpd.conf template
* vars/main.yml contains the name of the vsftpd service, the name of the RPM
package, and the location of the service's configuration file
```

```
Dependencies
```

```

```

```
None.
```

```
Example Playbook
```

```

```

```
- hosts: servers
 roles:
 - ansible-vsftpd
```

```
License
```

```

```

```
BSD
```

```
Author Information
```

```

Red Hat (training@redhat.com)
```

6. Remove the unused directories from the new role.

```
[student@workstation review-roles]$ rm -rfv roles/ansible-vsftpd/tests
removed 'roles/ansible-vsftpd/tests/inventory'
removed 'roles/ansible-vsftpd/tests/test.yml'
removed directory: 'roles/ansible-vsftpd/tests/'
```

7. Create the new playbook **vsftpd-configure.yml**. It should contain the following contents.

```

- name: Install and configure vsftpd
 hosts: ftpservers
 vars:
 vsftpd_anon_root: /mnt/share/
 vsftpd_local_root: /mnt/share/

 roles:
 - ansible-vsftpd

 tasks:

 - name: /dev/vdb1 is partitioned
 command: >
 parted --script /dev/vdb mklabel gpt mkpart primary 1MiB 100%
 args:
 creates: /dev/vdb1

 - name: XFS file system exists on /dev/vdb1
 filesystem:
 dev: /dev/vdb1
 fstype: xfs
 force: no

 - name: anon_root mount point exists
 file:
 path: '{{ vsftpd_anon_root }}'
 state: directory

 - name: /dev/vdb1 is mounted on anon_root
 mount:
 name: '{{ vsftpd_anon_root }}'
 src: /dev/vdb1
 fstype: xfs
 state: mounted
 dump: '1'
 passno: '2'
 notify: restart vsftpd

 - name: Make sure permissions on mounted fs are correct
 file:
 path: '{{ vsftpd_anon_root }}'
 owner: root
```

```
group: root
mode: '0755'
setype: "{{ vsftpd_setype }}"
state: directory

- name: Copy README to the ftp anon_root
 copy:
 dest: '{{ vsftpd_anon_root }}/README'
 content: "Welcome to the FTP server at {{ ansible_fqdn }}\n"
 setype: '{{ vsftpd_setype }}'
```

8. Change the **site.yml** playbook to use the newly created **vsftpd-configure.yml** playbook instead of the **ansible-vsftpd.yml** playbook. The file should contain the following contents after you are done.

```
FTP Servers playbook
- import_playbook: vsftpd-configure.yml

FTP Clients playbook
- import_playbook: ftclients.yml
```

9. Verify that the **site.yml** playbook works as intended by executing it with **ansible-playbook**.

```
[student@workstation review-roles]$ ansible-playbook site.yml
```

## Evaluation

From workstation, run the **lab review-roles grade** command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

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

## Cleanup

Run the **lab review-roles cleanup** command to clean up the lab tasks on servera and serverb.

```
[student@workstation ~]$ lab review-roles cleanup
```

## ► LAB

# RESTORING ANSIBLE TOWER FROM BACKUP

In this review, you will restore the Ansible Tower configuration from an existing backup.

## OUTCOMES

You should be able to restore the Ansible Tower configuration from a backup.

## BEFORE YOU BEGIN

In this exercise, you will restore the configuration of your Ansible Tower server using a backup archive.



### IMPORTANT

Your work from earlier exercises in this course will be erased from Tower by this exercise. You will start this exercise with an empty Ansible Tower environment.

Set up your computers for this exercise by logging in to `workstation` as `student`, and run the following command:

```
[student@workstation ~]$ lab review-review setup
```

## INSTRUCTIONS

On `tower.lab.example.com`, use the archive available at `http://materials.example.com/classroom/ansible/tower-backup-latest.tar.gz` to restore the Ansible Tower database from backup.

## Evaluation

Log into your Tower as `admin` and verify that the configuration has been restored. One indication is that there should only be one Job Template, `Demo Job Template`, present after the restoration.

This concludes the comprehensive review.

## ► SOLUTION

# RESTORING ANSIBLE TOWER FROM BACKUP

In this review, you will restore the Ansible Tower configuration from an existing backup.

## OUTCOMES

You should be able to restore the Ansible Tower configuration from a backup.

## BEFORE YOU BEGIN

In this exercise, you will restore the configuration of your Ansible Tower server using a backup archive.



### IMPORTANT

Your work from earlier exercises in this course will be erased from Tower by this exercise. You will start this exercise with an empty Ansible Tower environment.

Set up your computers for this exercise by logging in to `workstation` as `student`, and run the following command:

```
[student@workstation ~]$ lab review-review setup
```

## INSTRUCTIONS

On `tower.lab.example.com`, use the archive available at `http://materials.example.com/classroom/ansible/tower-backup-latest.tar.gz` to restore the Ansible Tower database from backup.

1. Restoring the Ansible Tower infrastructure from backup.

- 1.1. Use `ssh` to log in as `root` on the `tower` server.

```
[student@workstation ~]$ ssh root@tower
[root@tower ~]#
```

- 1.2. Change directory to `/root/ansible-tower-setup-bundle-3.3.1-1.el7`.

```
[root@tower ~]# cd /root/ansible-tower-setup-bundle-3.3.1-1.el7
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]#
```

- 1.3. Use the `wget` command to download the backup archive from `http://materials.example.com/classroom/ansible/tower-backup-latest.tar.gz`.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# wget \
http://materials.example.com/classroom/ansible/tower-backup-latest.tar.gz
...output omitted...
Saving to: 'tower-backup-latest.tar.gz'

100%[=====] 44,787 --.-K/s in 0s

2018-05-17 07:01:46 (110 MB/s) - 'tower-backup-latest.tar.gz' saved [44787/44787]
```

- 1.4. Using the **setup.sh** command with the **-r** option, restore Ansible Tower from backup.

```
[root@tower ansible-tower-setup-bundle-3.3.1-1.el7]# ./setup.sh -r
...output omitted...
PLAY RECAP ****
localhost : ok=40 changed=21 unreachable=0 failed=0

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2018-05-17-07:05:15.log
```

## Evaluation

Log into your Tower as `admin` and verify that the configuration has been restored. One indication is that there should only be one Job Template, **Demo Job Template**, present after the restoration.

## ► LAB

# ADDING USERS AND TEAMS

In this review, you will create new users and a new team.

## OUTCOMES

You should be able to:

- Create normal users.
- Create a new team.
- Add users to a team.

## BEFORE YOU BEGIN

You must have completed the preceding exercise, which restored a specific Ansible Tower backup on your server.

## INSTRUCTIONS

Configure your Ansible Tower system at `tower.lab.example.com` based on the following specification:

- Add a Normal User to the Default organization. Use the following information:

| FIELD      | VALUE                |
|------------|----------------------|
| FIRST NAME | Anny                 |
| LAST NAME  | Mage                 |
| EMAIL      | anny@lab.example.com |
| USERNAME   | anny                 |
| PASSWORD   | redhat123            |

- Add a second Normal User to the Default organization. Use the following information:

| FIELD      | VALUE                  |
|------------|------------------------|
| FIRST NAME | Robert                 |
| LAST NAME  | Farnham                |
| EMAIL      | robert@lab.example.com |
| USERNAME   | robert                 |
| PASSWORD   | redhat123              |

- Within the Default organization, create a Team called Devops. Give the team a description of Devops Team.

- Add anny as Member of the Devops Team. Add robert as Admin of the Devops Team.

## Evaluation

Use the Tower interface to verify to your satisfaction that the users are correctly configured.



### NOTE

There is a grading script that you will run at the end of the comprehensive review to evaluate all your work in this chapter.

This concludes the comprehensive review.

## ► SOLUTION

# ADDING USERS AND TEAMS

In this review, you will create new users and a new team.

## OUTCOMES

You should be able to:

- Create normal users.
- Create a new team.
- Add users to a team.

## BEFORE YOU BEGIN

You must have completed the preceding exercise, which restored a specific Ansible Tower backup on your server.

## INSTRUCTIONS

Configure your Ansible Tower system at `tower.lab.example.com` based on the following specification:

- Add a Normal User to the Default organization. Use the following information:

| FIELD      | VALUE                |
|------------|----------------------|
| FIRST NAME | Anny                 |
| LAST NAME  | Mage                 |
| EMAIL      | anny@lab.example.com |
| USERNAME   | anny                 |
| PASSWORD   | redhat123            |

- Add a second Normal User to the Default organization. Use the following information:

| FIELD      | VALUE                  |
|------------|------------------------|
| FIRST NAME | Robert                 |
| LAST NAME  | Farnham                |
| EMAIL      | robert@lab.example.com |
| USERNAME   | robert                 |
| PASSWORD   | redhat123              |

- Within the Default organization, create a Team called Devops. Give the team a description of Devops Team.

- Add anny as Member of the Devops Team. Add robert as Admin of the Devops Team.

1. Log in to the Tower web interface as `admin` user, and then click **Users** in the left navigation bar.
  - 1.1. Click the **+** button to add a new User.
  - 1.2. On the next screen, fill in the details as follows:

| FIELD            | VALUE                |
|------------------|----------------------|
| FIRST NAME       | Anny                 |
| LAST NAME        | Mage                 |
| EMAIL            | anny@lab.example.com |
| USERNAME         | anny                 |
| ORGANIZATION     | Default              |
| PASSWORD         | redhat123            |
| CONFIRM PASSWORD | redhat123            |
| USER TYPE        | Normal User          |

- 1.3. Click **SAVE** to create the new User.
- 1.4. Scroll down the next screen and click the **+** button on the right to add another User.
- 1.5. On the next screen, fill in the details as follows:

| FIELD            | VALUE                  |
|------------------|------------------------|
| FIRST NAME       | Robert                 |
| LAST NAME        | Farnham                |
| EMAIL            | robert@lab.example.com |
| USERNAME         | robert                 |
| ORGANIZATION     | Default                |
| PASSWORD         | redhat123              |
| CONFIRM PASSWORD | redhat123              |
| USER TYPE        | Normal User            |

- 1.6. Click **SAVE** to create the new User.
2. To create the Devops Team:
  - 2.1. Click **Teams** in the left navigation bar to manage Teams.

- 2.2. Click + to add a new Team.
- 2.3. On the next screen, fill in the details as follows:

| FIELD        | VALUE       |
|--------------|-------------|
| NAME         | Devops      |
| DESCRIPTION  | Devops Team |
| ORGANIZATION | Default     |

- 2.4. Click SAVE to create the new Team.
3. To add Users to the Devops Team:
  - 3.1. Click Teams in the left navigation bar to manage Teams.
  - 3.2. Click the link for the Devops Team you created previously.
  - 3.3. Click USERS to manage the Team's Users.
  - 3.4. Click + to add a new User to the Team.
  - 3.5. In the first section on the screen, check the box next to anny and robert to select those Users. This adds anny and robert to the list of selected Users in the second section below.
  - 3.6. On the second section, assign the Member role to Anny Mage and the Admin role to Robert Farnham.
  - 3.7. Click SAVE button to save the changes.
  - 3.8. You can verify on the next screen that anny is displayed in the list as a Member and robert as an Admin.

## Evaluation

Use the Tower interface to verify to your satisfaction that the users are correctly configured.



### NOTE

There is a grading script that you will run at the end of the comprehensive review to evaluate all your work in this chapter.

## ▶ LAB

# CREATING A CUSTOM DYNAMIC INVENTORY

In this review, you will create a dynamic inventory using a custom script.

## OUTCOMES

You should be able to:

- Install custom inventory script.
- Create a dynamic inventory in Red Hat Ansible Tower.

## BEFORE YOU BEGIN

The previous exercises in this comprehensive review chapter must be completed before starting this exercise.

## INSTRUCTIONS

Configure your Ansible Tower server with a custom dynamic inventory script, **ldap-freeipa.py**, based on the following specification:

- Add the **ldap-freeipa.py** custom inventory script to Tower. The Python script can be downloaded from <http://materials.example.com/classroom/ansible/ipa-setup/ldap-freeipa.py>.
- Create a new Inventory called **Dynamic Inventory**. Using Default organization, give the script a Description of **Dynamic Inventory for IPA Server**.
- Create a Group called **Dynamic Group** within the Inventory **Dynamic Inventory**. Give it a Description of **Dynamic Group from IPA Server**. For Source, choose Custom Script.
- Update the dynamic inventory. When finished, review every group synchronized from the IPA Server to ensure that all groups contain hosts.
- Create a new Machine Credential called **Devops**. Use the following information:

| FIELD                       | VALUE             |
|-----------------------------|-------------------|
| NAME                        | Devops            |
| DESCRIPTION                 | Devops Credential |
| ORGANIZATION                | Default           |
| CREDENTIAL TYPE             | Machine           |
| USERNAME                    | devops            |
| PASSWORD                    | redhat            |
| PRIVILEGE ESCALATION METHOD | <b>sudo</b>       |

| FIELD                         | VALUE  |
|-------------------------------|--------|
| PRIVILEGE ESCALATION USERNAME | root   |
| PRIVILEGE ESCALATION PASSWORD | redhat |

- Grant the Admin role on the Devops Credential to the Devops Team.

## Evaluation

Use the Tower interface to verify to your satisfaction that the dynamic inventory script is correctly configured.



### NOTE

There is a grading script that you will run at the end of the comprehensive review to evaluate all your work in this chapter.

This concludes the comprehensive review.

## ► SOLUTION

# CREATING A CUSTOM DYNAMIC INVENTORY

In this review, you will create a dynamic inventory using a custom script.

## OUTCOMES

You should be able to:

- Install custom inventory script.
- Create a dynamic inventory in Red Hat Ansible Tower.

## BEFORE YOU BEGIN

The previous exercises in this comprehensive review chapter must be completed before starting this exercise.

## INSTRUCTIONS

Configure your Ansible Tower server with a custom dynamic inventory script, **ldap-freeipa.py**, based on the following specification:

- Add the **ldap-freeipa.py** custom inventory script to Tower. The Python script can be downloaded from <http://materials.example.com/classroom/ansible/ipa-setup/ldap-freeipa.py>.
- Create a new Inventory called **Dynamic Inventory**. Using Default organization, give the script a Description of **Dynamic Inventory for IPA Server**.
- Create a Group called **Dynamic Group** within the Inventory **Dynamic Inventory**. Give it a Description of **Dynamic Group from IPA Server**. For Source, choose Custom Script.
- Update the dynamic inventory. When finished, review every group synchronized from the IPA Server to ensure that all groups contain hosts.
- Create a new Machine Credential called **Devops**. Use the following information:

| FIELD                       | VALUE             |
|-----------------------------|-------------------|
| NAME                        | Devops            |
| DESCRIPTION                 | Devops Credential |
| ORGANIZATION                | Default           |
| CREDENTIAL TYPE             | Machine           |
| USERNAME                    | devops            |
| PASSWORD                    | redhat            |
| PRIVILEGE ESCALATION METHOD | <b>sudo</b>       |

| FIELD                         | VALUE  |
|-------------------------------|--------|
| PRIVILEGE ESCALATION USERNAME | root   |
| PRIVILEGE ESCALATION PASSWORD | redhat |

- Grant the Admin role on the Devops Credential to the Devops Team.

- To add the **ldap-freeipa.py** custom inventory script to Ansible Tower:
  - Log in to the Ansible Tower as **admin** user.
  - Click **Inventory Scripts** in the left navigation bar to manage custom inventory scripts.
  - Click **+** to add a custom inventory script.
  - On the next screen, fill in the details as follows:

| FIELD        | VALUE                            |
|--------------|----------------------------------|
| NAME         | ldap-freeipa.py                  |
| DESCRIPTION  | Dynamic Inventory for IPA Server |
| ORGANIZATION | Default                          |

  - Copy the contents of the **ldap-freeipa.py** script located at <http://materials.example.com/classroom/ansible/ipa-setup/ldap-freeipa.py> into the CUSTOM SCRIPT field.
  - Click **SAVE** to add the custom inventory script.
- Click the **Inventories** in the left quick navigation bar.
  - Click the **+** button to add a new inventory, and select **Inventory** from the drop-down list.
  - On the next screen, fill in the details as follows:

| FIELD        | VALUE                             |
|--------------|-----------------------------------|
| NAME         | Dynamic Inventory                 |
| DESCRIPTION  | Dynamic Inventory from IPA server |
| ORGANIZATION | Default                           |

  - Click **SAVE** to create the Inventory. This redirects you to the **Dynamic Inventory** details page.

- Within the inventory **Dynamic Inventory**, add the **ldap-freeipa.py** script as a new source for the Inventory.
  - Click **SOURCES** in the top bar for the Inventory details pane.
  - Click the **+** button to add a new source.

- 3.3. On the next screen, fill in the details as follows:

| FIELD                   | VALUE                               |
|-------------------------|-------------------------------------|
| NAME                    | Custom Script                       |
| DESCRIPTION             | Custom Script for Dynamic Inventory |
| SOURCE                  | Custom Script                       |
| CUSTOM INVENTORY SCRIPT | ldap-freeipa.py                     |

- 3.4. Under the UPDATE OPTIONS section, check the checkbox next to the Overwrite option.
- 3.5. Click SAVE to create the Source.
4. Update the dynamic inventory.
- 4.1. Scroll down to the lower pane and click the double-arrow button in the row for Custom Script. Wait until the cloud becomes green and static.
- 4.2. Use the top breadcrumb menu to navigate back to the Inventory details pane, and click GROUPS section. Observe that it now contains four Groups: development, ipaservers, production, and testing. Each of these groups contains hosts.
5. To create new Machine Credentials:
- 5.1. Click Credentials in the left navigation bar, to manage Credentials.
- 5.2. Click the + button to add a new Credential.
- 5.3. Create a new Credential, Devops, with the following information:

| FIELD                         | VALUE             |
|-------------------------------|-------------------|
| NAME                          | Devops            |
| DESCRIPTION                   | Devops Credential |
| ORGANIZATION                  | Default           |
| CREDENTIAL TYPE               | Machine           |
| USERNAME                      | devops            |
| PASSWORD                      | redhat            |
| PRIVILEGE ESCALATION METHOD   | <b>sudo</b>       |
| PRIVILEGE ESCALATION USERNAME | root              |
| PRIVILEGE ESCALATION PASSWORD | redhat            |

- 5.4. Leave the other fields untouched and click SAVE to create the new Credential.

6. To grant Admin role to Credential:
  - 6.1. Click Credentials in the left navigation bar, to manage Credentials.
  - 6.2. Click the Devops Credential to edit the Credential.
  - 6.3. On the next page, click PERMISSIONS to manage the Credential's permissions.
  - 6.4. Click the + button to add permissions.
  - 6.5. Click TEAMS to display the list of available Teams.
  - 6.6. In the first section, select the box next to the Devops Team. This causes the Team to display in the second section underneath the first one.
  - 6.7. In the second section below, select the Admin Role from the drop-down list.
  - 6.8. Click SAVE to finalize the role assignment. This redirects you to the list of permissions for the Devops Credential, which now shows that the Users, anny and robert, are assigned the Admin role on the Devops Credential.

## Evaluation

Use the Tower interface to verify to your satisfaction that the dynamic inventory script is correctly configured.



### NOTE

There is a grading script that you will run at the end of the comprehensive review to evaluate all your work in this chapter.

## ▶ LAB

# CONFIGURING JOB TEMPLATES

In this review, you will configure several Job Templates, as well as a supporting Project and Source Control credentials.

## OUTCOMES

You should be able to:

- Create a Source Control credential.
- Create a Project.
- Create a Job Template.

## BEFORE YOU BEGIN

The previous exercises in this comprehensive review chapter must be completed before starting this exercise.

## INSTRUCTIONS

Configure your Ansible Tower server with three new Job Templates and supporting materials based on the following specification:

- Create a new SCM Credential, needed to download the Ansible materials for the Project that the new Job Templates will use, based on the following information:

| FIELD           | VALUE                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------------|
| NAME            | student-git                                                                                                |
| DESCRIPTION     | Student Git Credential                                                                                     |
| ORGANIZATION    | Default                                                                                                    |
| CREDENTIAL TYPE | Source Control                                                                                             |
| USERNAME        | git                                                                                                        |
| SCM PRIVATE KEY | Copy the contents of the <b>/home/student/.ssh/lab_rsa</b> private key file on workstation into this field |

- Create a new Project based on the following information:

| FIELD        | VALUE                 |
|--------------|-----------------------|
| NAME         | My Full-Stack Project |
| DESCRIPTION  | Full Stack Project    |
| ORGANIZATION | Default               |
| SCM TYPE     | Git                   |

| FIELD          | VALUE                                        |
|----------------|----------------------------------------------|
| SCM URL        | git@git.lab.example.com:full-stack-setup.git |
| SCM CREDENTIAL | student-git                                  |

- Create a new Job Template called `Set up Databases` with the following configuration:

| FIELD       | VALUE                     |
|-------------|---------------------------|
| NAME        | Set up Databases          |
| DESCRIPTION | Set up all databases      |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <b>database-setup.yml</b> |
| CREDENTIAL  | Devops                    |

- Create a new Job Template called `Set up Web servers` with the following configuration:

| FIELD       | VALUE                      |
|-------------|----------------------------|
| NAME        | Set up Web servers         |
| DESCRIPTION | Set up all web servers     |
| JOB TYPE    | Run                        |
| INVENTORY   | Dynamic Inventory          |
| PROJECT     | My Full-Stack Project      |
| PLAYBOOK    | <b>webserver-setup.yml</b> |
| CREDENTIAL  | Devops                     |

- Create a new Job Template called `Set up Load Balancer` with the following configuration:

| FIELD       | VALUE                     |
|-------------|---------------------------|
| NAME        | Set up Load Balancer      |
| DESCRIPTION | Set up all load balancers |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |

| FIELD      | VALUE                 |
|------------|-----------------------|
| PROJECT    | My Full-Stack Project |
| PLAYBOOK   | <b>1b-setup.yml</b>   |
| CREDENTIAL | Devops                |

- The Devops Team should have Admin role on all three Job Templates (Set up Databases, Set up Web servers, and Set up Load Balancer) and on the Project (My Full-Stack Project).

## Evaluation

Use the Ansible Tower interface to verify to your satisfaction that the project materials update and that your Job Templates are configured correctly.



### NOTE

You will use these templates to launch jobs in an upcoming exercise, and there is a grading script at the comprehensive review that you will run to evaluate all your work in this chapter.

This concludes the comprehensive review.

## ► SOLUTION

# CONFIGURING JOB TEMPLATES

In this review, you will configure several Job Templates, as well as a supporting Project and Source Control credentials.

## OUTCOMES

You should be able to:

- Create a Source Control credential.
- Create a Project.
- Create a Job Template.

## BEFORE YOU BEGIN

The previous exercises in this comprehensive review chapter must be completed before starting this exercise.

## INSTRUCTIONS

Configure your Ansible Tower server with three new Job Templates and supporting materials based on the following specification:

- Create a new SCM Credential, needed to download the Ansible materials for the Project that the new Job Templates will use, based on the following information:

| FIELD           | VALUE                                                                                                            |
|-----------------|------------------------------------------------------------------------------------------------------------------|
| NAME            | student-git                                                                                                      |
| DESCRIPTION     | Student Git Credential                                                                                           |
| ORGANIZATION    | Default                                                                                                          |
| CREDENTIAL TYPE | Source Control                                                                                                   |
| USERNAME        | git                                                                                                              |
| SCM PRIVATE KEY | Copy the contents of the <code>/home/student/.ssh/lab_rsa</code> private key file on workstation into this field |

- Create a new Project based on the following information:

| FIELD        | VALUE                 |
|--------------|-----------------------|
| NAME         | My Full-Stack Project |
| DESCRIPTION  | Full Stack Project    |
| ORGANIZATION | Default               |
| SCM TYPE     | Git                   |

| FIELD          | VALUE                                        |
|----------------|----------------------------------------------|
| SCM URL        | git@git.lab.example.com:full-stack-setup.git |
| SCM CREDENTIAL | student-git                                  |

- Create a new Job Template called Set up Databases with the following configuration:

| FIELD       | VALUE                     |
|-------------|---------------------------|
| NAME        | Set up Databases          |
| DESCRIPTION | Set up all databases      |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <b>database-setup.yml</b> |
| CREDENTIAL  | Devops                    |

- Create a new Job Template called Set up Web servers with the following configuration:

| FIELD       | VALUE                      |
|-------------|----------------------------|
| NAME        | Set up Web servers         |
| DESCRIPTION | Set up all web servers     |
| JOB TYPE    | Run                        |
| INVENTORY   | Dynamic Inventory          |
| PROJECT     | My Full-Stack Project      |
| PLAYBOOK    | <b>webserver-setup.yml</b> |
| CREDENTIAL  | Devops                     |

- Create a new Job Template called Set up Load Balancer with the following configuration:

| FIELD       | VALUE                     |
|-------------|---------------------------|
| NAME        | Set up Load Balancer      |
| DESCRIPTION | Set up all load balancers |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |

DO410-RHAT-3.3-en-1-20190110

| FIELD      | VALUE                 |
|------------|-----------------------|
| PROJECT    | My Full-Stack Project |
| PLAYBOOK   | <b>lb-setup.yml</b>   |
| CREDENTIAL | Devops                |

- The Devops Team should have Admin role on all three Job Templates (Set up Databases, Set up Web servers, and Set up Load Balancer) and on the Project (My Full-Stack Project).

**1.** Steps to create the new Source Control Credential:

- 1.1. Click Credentials in the left navigation bar, to manage Credentials.
- 1.2. Click the + button to add a new Credential.
- 1.3. On the next screen, fill in the details as follows:

| FIELD           | VALUE                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------------|
| NAME            | student-git                                                                                                |
| DESCRIPTION     | Student Git Credential                                                                                     |
| ORGANIZATION    | Default                                                                                                    |
| CREDENTIAL TYPE | Source Control                                                                                             |
| USERNAME        | git                                                                                                        |
| SCM PRIVATE KEY | Copy the contents of the <b>/home/student/.ssh/lab_rsa</b> private key file on workstation into this field |

- 1.4. Leave the other fields untouched and click **SAVE** to create the new Credential.

**2.** To create the new Project:

- 2.1. Click Projects in the left quick navigation bar.
- 2.2. Click the + button to add a new Project.
- 2.3. On the next screen, fill in the details as follows:

| FIELD        | VALUE                 |
|--------------|-----------------------|
| NAME         | My Full-Stack Project |
| DESCRIPTION  | Full Stack Project    |
| ORGANIZATION | Default               |
| SCM TYPE     | Git                   |

| FIELD          | VALUE                                        |
|----------------|----------------------------------------------|
| SCM URL        | git@git.lab.example.com:full-stack-setup.git |
| SCM CREDENTIAL | student-git                                  |

- 2.4. Click **SAVE** to create the new Project. This automatically triggers the SCM update of the Project. Ansible Tower uses the values provided in the SCM URL and SCM CREDENTIAL fields to pull down a local copy of that repository.
3. Verify the success of the **My Full-Stack Project** automatic SCM update:
  - 3.1. Scroll down the page and wait a couple of seconds. In the list of Projects, there is a status icon to the left of **My Full-Stack Project**. This icon is white at the start, red with an exclamation mark when it fails, and green when it succeeds.
  - 3.2. Click on the status icon to show the detailed status page of the SCM update job. As you can see in the **DETAIL** window, the SCM update job runs like any other Ansible Playbook.
  - 3.3. Verify that the **STATUS** of the job in the **DETAILS** section shows **Successful**.
4. To give the **Devops Team** the **Admin** role on the new Project:
  - 4.1. Click **Projects** in the left navigation bar.
  - 4.2. On the same line as **My Full-Stack Project**, click the pencil icon on the right to edit the Project.
  - 4.3. On the next page, click **PERMISSIONS** to manage the Project's permissions.
  - 4.4. Click the **+** button on the right to add permissions.
  - 4.5. Click **TEAMS** to display the list of available Teams.
  - 4.6. In the first section, check the box next to the **Devops Team**. This causes the Team to display in the second section underneath the first one.
  - 4.7. In the second section below, select the **Admin** role from the drop-down list.
  - 4.8. Click **SAVE** to make the role assignment. This redirects you to the list of permissions for the Project, **My Full-Stack Project**, which now shows that all members of the **Devops Team** are assigned the **Admin** role on the Project.
5. To create new Job Template called **Set up Databases**:
  - 5.1. Click **Templates** in the left navigation bar.
  - 5.2. Click the **+** button to add a new Job Template.
  - 5.3. From the drop-down list, select **Job Template**.
  - 5.4. On the next screen, fill in the details as follows:

| FIELD | VALUE            |
|-------|------------------|
| NAME  | Set up Databases |

| FIELD       | VALUE                     |
|-------------|---------------------------|
| DESCRIPTION | Set up all databases      |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <b>database-setup.yml</b> |
| CREDENTIAL  | Devops                    |

- 5.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.
6. Grant the Devops Team Admin role on the **Set up Databases** Job Template:
- 6.1. Click **PERMISSIONS** to manage the Job Template's permissions.
  - 6.2. Click the **+** button on the right to add permissions.
  - 6.3. Click **TEAMS** to display the list of available Teams.
  - 6.4. In the first section, select the box next to **Devops Team**. This causes the Team to display in the second section underneath the first one.
  - 6.5. In the second section below, select the **Admin** role from the drop-down list.
  - 6.6. Click **SAVE** to make the role assignment. This redirects you to the list of permissions for the Job Template, **Set up Databases**, which now shows that all members of the Devops Team are assigned the **Admin** role on the Job Template.
7. Create new Job Template called **Set up Web servers**:
- 7.1. Click **Templates** in the left navigation bar.
  - 7.2. Click the **+** button to add a new Job Template.
  - 7.3. From the drop-down list, select **Job Template**.
  - 7.4. On the next screen, fill in the details as follows:

| FIELD       | VALUE                      |
|-------------|----------------------------|
| NAME        | Set up Web servers         |
| DESCRIPTION | Set up all Web servers     |
| JOB TYPE    | Run                        |
| INVENTORY   | Dynamic Inventory          |
| PROJECT     | My Full-Stack Project      |
| PLAYBOOK    | <b>webserver-setup.yml</b> |

| FIELD      | VALUE  |
|------------|--------|
| CREDENTIAL | Devops |

- 7.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.
- 8.** Grant the Devops Team Admin role on the **Set up Web servers** Job Template:
- 8.1. Click **PERMISSIONS** to manage the Job Template's permissions.
  - 8.2. Click the **+** button on the right to add permissions.
  - 8.3. Click **TEAMS** to display the list of available Teams.
  - 8.4. In the first section, check the box next to Devops Team. This causes the Team to display in the second section underneath the first one.
  - 8.5. In the second section below, select the **Admin** role from the drop-down list.
  - 8.6. Click **SAVE** to make the role assignment. This redirects you to the list of permissions for the Job Template, **Set up Web servers**, which now shows that all members of the Devops Team are assigned the Admin role on the Job Template.
- 9.** To create new Job Template called **Set up Load Balancer**:
- 9.1. Click **Templates** in the left navigation bar.
  - 9.2. Click the **+** button to add a new Job Template.
  - 9.3. From the drop-down list, select **Job Template**.
  - 9.4. On the next screen, fill in the details as follows:

| FIELD       | VALUE                    |
|-------------|--------------------------|
| NAME        | Set up Load Balancer     |
| DESCRIPTION | Set up all loadbalancers |
| JOB TYPE    | Run                      |
| INVENTORY   | Dynamic Inventory        |
| PROJECT     | My Full-Stack Project    |
| PLAYBOOK    | <b>lb-setup.yml</b>      |
| CREDENTIAL  | Devops                   |

- 9.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.

10. Grant the Devops Team Admin role on Set up Load Balancer Job Template:
- 10.1. Click PERMISSIONS to manage the Job Template's permissions.
  - 10.2. Click the + button on the right to add permissions.
  - 10.3. Click TEAMS to display the list of available Teams.
  - 10.4. In the first section, check the box next to Devops Team. This causes the Team to display in the second section underneath the first one.
  - 10.5. In the second section below, select the Admin role from the drop-down list.
  - 10.6. Click SAVE to make the role assignment. This redirects you to the list of permissions for the Job Template, Set up Load Balancer, which now shows that all members of the Devops Team are assigned the Admin role on the Job Template.

## Evaluation

Use the Ansible Tower interface to verify to your satisfaction that the project materials update and that your Job Templates are configured correctly.



### NOTE

You will use these templates to launch jobs in an upcoming exercise, and there is a grading script at the comprehensive review that you will run to evaluate all your work in this chapter.

## ► LAB

# CONFIGURING WORKFLOW JOB TEMPLATES, SURVEYS, AND NOTIFICATIONS

In this review, you will create a Workflow Job Template which uses a survey and sends email notifications.

## OUTCOMES

You should be able to:

- Create a Workflow Job Template.
- Add a Survey to an existing Workflow Job Template.
- Create and activate an email Notification Template.

## BEFORE YOU BEGIN

The previous exercises in this comprehensive review must be completed before starting this exercise.

## INSTRUCTIONS

Configure your Ansible Tower server with a Workflow Job Template using your existing Job Templates, a new Survey, and a new Notification Template, based on the following specification:

- Create a Workflow Job Template in the **Default** organization called **Full Stack Deployment**. Its **Description** should be **Deploy the Full Stack**. Define an Extra Variable **environment\_name** with the value "*Development*".

The Workflow Job Template should include the following steps:

1. Sync the Inventory, **Dynamic Inventory**.
2. Upon success of the previous step, sync the Project, **My Full-Stack Project**.
3. Upon success of the previous step, launch a job using the Job Template, **Set up Databases**.
4. Upon success of the previous step, launch a job using the Job Template, **Set up Web servers**.
5. Upon success of the previous step, launch a job using the Job Template, **Set up Load Balancer**.

The Devops Team should have the **Admin** role on the Workflow Job Template.

- Add a Survey to the **Full Stack Deployment** Workflow Job Template. Configure the Survey based on the following specification:

| FIELD                | VALUE                                                             |
|----------------------|-------------------------------------------------------------------|
| PROMPT               | What environment are you deploying?                               |
| DESCRIPTION          | The environment will be displayed at the bottom of the index page |
| ANSWER VARIABLE NAME | environment_name                                                  |
| ANSWER TYPE          | Text                                                              |
| MINIMUM LENGTH       | 1                                                                 |
| MAXIMUM LENGTH       | 40                                                                |
| DEFAULT ANSWER       | Development                                                       |
| REQUIRED             | Enabled                                                           |

- Create an **email** Notification Template named **Notify on Job Success and Failure**, based on the following specification:

| FIELD          | VALUE                                          |
|----------------|------------------------------------------------|
| NAME           | Notify on Job Success and Failure              |
| DESCRIPTION    | Sends an email to notify the status of the Job |
| ORGANIZATION   | Default                                        |
| TYPE           | Email                                          |
| HOST           | localhost                                      |
| RECIPIENT LIST | student@lab.example.com                        |
| SENDER EMAIL   | system@lab.example.com                         |
| PORT           | 25                                             |

- For the Workflow Job Template, **Full Stack Deployment**, activate the **SUCCESS** and the **FAILURE** Notifications using the Notification Template, **Notify on Job Success and Failure**.

## Evaluation

Use the Ansible Tower interface to confirm to your satisfaction that everything is configured as specified.



**NOTE**

The next activity in this chapter will instruct you to launch the Workflow Job Template and otherwise test your work in this chapter.

This concludes the comprehensive review.

## ► SOLUTION

# CONFIGURING WORKFLOW JOB TEMPLATES, SURVEYS, AND NOTIFICATIONS

In this review, you will create a Workflow Job Template which uses a survey and sends email notifications.

## OUTCOMES

You should be able to:

- Create a Workflow Job Template.
- Add a Survey to an existing Workflow Job Template.
- Create and activate an email Notification Template.

## BEFORE YOU BEGIN

The previous exercises in this comprehensive review must be completed before starting this exercise.

## INSTRUCTIONS

Configure your Ansible Tower server with a Workflow Job Template using your existing Job Templates, a new Survey, and a new Notification Template, based on the following specification:

- Create a Workflow Job Template in the `Default` organization called `Full Stack Deployment`. Its `Description` should be `Deploy the Full Stack`. Define an Extra Variable `environment_name` with the value "`Development`".

The Workflow Job Template should include the following steps:

1. Sync the Inventory, `Dynamic Inventory`.
2. Upon success of the previous step, sync the Project, `My Full-Stack Project`.
3. Upon success of the previous step, launch a job using the Job Template, `Set up Databases`.
4. Upon success of the previous step, launch a job using the Job Template, `Set up Web servers`.
5. Upon success of the previous step, launch a job using the Job Template, `Set up Load Balancer`.

The Devops Team should have the `Admin` role on the Workflow Job Template.

- Add a Survey to the `Full Stack Deployment` Workflow Job Template. Configure the Survey based on the following specification:

| FIELD                | VALUE                                                             |
|----------------------|-------------------------------------------------------------------|
| PROMPT               | What environment are you deploying?                               |
| DESCRIPTION          | The environment will be displayed at the bottom of the index page |
| ANSWER VARIABLE NAME | environment_name                                                  |
| ANSWER TYPE          | Text                                                              |
| MINIMUM LENGTH       | 1                                                                 |
| MAXIMUM LENGTH       | 40                                                                |
| DEFAULT ANSWER       | Development                                                       |
| REQUIRED             | Enabled                                                           |

- Create an **email** Notification Template named **Notify on Job Success and Failure**, based on the following specification:

| FIELD          | VALUE                                          |
|----------------|------------------------------------------------|
| NAME           | Notify on Job Success and Failure              |
| DESCRIPTION    | Sends an email to notify the status of the Job |
| ORGANIZATION   | Default                                        |
| TYPE           | Email                                          |
| HOST           | localhost                                      |
| RECIPIENT LIST | student@lab.example.com                        |
| SENDER EMAIL   | system@lab.example.com                         |
| PORT           | 25                                             |

- For the Workflow Job Template, **Full Stack Deployment**, activate the **SUCCESS** and the **FAILURE** Notifications using the Notification Template, **Notify on Job Success and Failure**.

1. To create a Workflow Job Template called **Full Stack Deployment**:
  - 1.1. Click **Templates** in the left quick navigation bar.
  - 1.2. Click the **+** button to add a new Workflow Job Template.
  - 1.3. From the drop-down list, select **Workflow Template**.
  - 1.4. On the next screen, fill in the details as follows:

| FIELD           | VALUE                              |
|-----------------|------------------------------------|
| NAME            | Full Stack Deployment              |
| DESCRIPTION     | Deploy the Full Stack              |
| ORGANIZATION    | Default                            |
| EXTRA VARIABLES | environment_name:<br>"Development" |

- 1.5. Click SAVE to create the new Workflow Job Template.
2. To configure the Workflow for the **Full Stack Deployment** Workflow Job Template:
- 2.1. Click WORKFLOW VISUALIZER to open the Workflow VISUALIZER.
  - 2.2. Click START to add the first action to be performed. This will display, in the right panel, a list of actions to be performed.
  - 2.3. In the right panel, click INVENTORY SYNC to display the list of Inventories available.
  - 2.4. Select Custom Script and click SELECT. This links the START node with a blue line (*always perform*) to the node for the Dynamic Inventory, Custom Script, in the Workflow VISUALIZER window.
  - 2.5. Move your mouse over the new node and click on the green + button to add an action after the Inventory Sync of Custom Script. This displays a list of actions to be performed in the right panel.
  - 2.6. In the right panel, click PROJECT SYNC to display the list of Projects available.
  - 2.7. Select My Full-Stack Project and click SELECT. In the Workflow VISUALIZER window, this links the previous node to the node for the Project, My Full-Stack Project, with a green line, indicating that this progression will only be performed if the Inventory Sync step is successful.
  - 2.8. Move your mouse over to the new node and click on the green + button to add an action after the Project Sync of My Full-Stack Project. This displays a list of actions to be performed in the right panel.
  - 2.9. In the right panel, make sure you are in the JOBS section and select the Set up Databases Job Template.
  - 2.10. In the RUN section below, select **On Success** and click SELECT. In the Workflow VISUALIZER window, this links the node for My Full-Stack Project to the node

for the Set up Databases Job Template with a green line, indicating that this progression is only performed if the Project Sync step is successful.

- 2.11. Move your mouse over the new node and click on the green + button to add an action after the Set up Databases Job Template.
  - 2.12. In the right panel, make sure you are in the JOBS section and select the Set up Web servers Job Template. In the RUN section below, select **On Success** and click SELECT.
  - 2.13. Move your mouse over the new node and click on the green + button to add an action after the Set up Web servers Job Template.
  - 2.14. In the right panel, make sure you are in the JOBS section and select the Set up Load Balancer Job Template. In the RUN section below, select On Success and click SELECT.
  - 2.15. Click SAVE to save the Workflow Job Template.
- 3.** Grant Devops Team **Admin** role on the Workflow Job Template:
- 3.1. Click PERMISSIONS to manage the Workflow Job Template's permissions.
  - 3.2. Click the + button on the right to add permissions.
  - 3.3. Click TEAMS to display the list of available Teams.
  - 3.4. In the first section, select the box next to Devops Team. This causes the Team to display in the second section underneath the first one.
  - 3.5. In the second section below, select the **Admin** role from the drop-down list.
  - 3.6. Click SAVE to make the role assignment. This redirects you to the list of permissions for the Workflow Job Template, Full Stack Deployment, which now shows that all members of the Devops Team are assigned the **Admin** role on the Workflow Job Template.
- 4.** Add a Survey to the Full Stack Deployment Workflow Job Template:
- 4.1. Click the DETAILS button to return to the Template details screen.
  - 4.2. Click the ADD SURVEY button to add a Survey.
  - 4.3. On the next screen, fill in the details as follows:

| FIELD                | VALUE                                                             |
|----------------------|-------------------------------------------------------------------|
| PROMPT               | What environment are you deploying?                               |
| DESCRIPTION          | The environment will be displayed at the bottom of the index page |
| ANSWER VARIABLE NAME | environment_name                                                  |
| ANSWER TYPE          | Text                                                              |
| MINIMUM LENGTH       | 1                                                                 |
| MAXIMUM LENGTH       | 40                                                                |

| FIELD          | VALUE       |
|----------------|-------------|
| DEFAULT ANSWER | Development |
| REQUIRED       | Enabled     |

- 4.4. Click **+ADD** to add the Survey Prompt to the Survey. This displays a preview of your Survey on the right.

**IMPORTANT**

Before saving, make sure that the ON/OFF switch is set to **ON** at the top of the Survey editor window.

- 4.5. Click **SAVE** to add the Survey to the Job Template.
5. To create an **email** Notification Template called **Notify on Job Success and Failure**:
- 5.1. Click **Notifications** in the left quick navigation bar, to manage Notification Templates.
  - 5.2. Click **+** to add a Notification Template.
  - 5.3. On the next screen, fill in the details as follows:

| FIELD          | VALUE                                          |
|----------------|------------------------------------------------|
| NAME           | Notify on Job Success and Failure              |
| DESCRIPTION    | Sends an email to notify the status of the Job |
| ORGANIZATION   | Default                                        |
| TYPE           | Email                                          |
| HOST           | localhost                                      |
| RECIPIENT LIST | student@lab.example.com                        |
| SENDER EMAIL   | system@lab.example.com                         |
| PORT           | 25                                             |

- 5.4. Leave all the other fields untouched and click **SAVE** to save the Notification Template. You are then redirected to the list of Notification Templates.

6. To activate both the **SUCCESS** and **FAILURE** Notifications for the Full Stack Deployment Workflow Job Template:
  - 6.1. Click **Templates** in the left quick navigation bar.
  - 6.2. Click the link for the **Full Stack Deployment** Workflow Job Template.
  - 6.3. Click **NOTIFICATIONS** to manage notifications for the **Full Stack Deployment** Workflow Job Template.
  - 6.4. On the same line as the Notification Template, **Notify on Job Success and Failure**, set both ON/OFF switches for **SUCCESS** and **FAILURE** to ON.

## Evaluation

Use the Ansible Tower interface to confirm to your satisfaction that everything is configured as specified.



### NOTE

The next activity in this chapter will instruct you to launch the Workflow Job Template and otherwise test your work in this chapter.

## ► LAB

# TESTING THE PREPARED ENVIRONMENT

In this review, you will test your work in this Comprehensive Review chapter.

## OUTCOMES

You should be able to:

- Launch the Full Stack Deployment workflow Job Template.
- Verify that the workflow sent an email notification.
- Verify that the end results of your work are correct.

## BEFORE YOU BEGIN

All previous exercise in this comprehensive review should be completed before evaluating your work with this section.

## INSTRUCTIONS

Manually test your work on this comprehensive review as follows:

- As user `robert` in the Ansible Tower interface, launch the Full Stack Deployment Workflow Job Template
- As the Linux user `student` on `tower.lab.example.com`, verify that the Full Stack Deployment Workflow Job Template sent you an email when it completed.
- Verify that the web servers have been installed and configured on `servera.lab.example.com`, `serverd.lab.example.com`, `servere.lab.example.com`, and `serverf.lab.example.com`
- Verify that the load balancer has been installed and configured on `serverb.lab.example.com`.

If it is working correctly, it should direct sequential HTTP requests sent to `http://serverb.lab.example.com` to different web servers. Every refresh of this page redirects the connection to a different web server.

If the configuration is correct, it should redirect connections to the following web servers: `servera`, `serverd`, `servere` and `serverf`.

## Evaluation

As the student user on `workstation`, run the `lab review-review` script with the `grade` argument, to confirm success on this exercises. Correct any reported failures and rerun the script until successful.

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

This concludes the comprehensive review.

## ► SOLUTION

# TESTING THE PREPARED ENVIRONMENT

In this review, you will test your work in this Comprehensive Review chapter.

## OUTCOMES

You should be able to:

- Launch the `Full Stack Deployment` workflow Job Template.
- Verify that the workflow sent an email notification.
- Verify that the end results of your work are correct.

## BEFORE YOU BEGIN

All previous exercise in this comprehensive review should be completed before evaluating your work with this section.

## INSTRUCTIONS

Manually test your work on this comprehensive review as follows:

- As user `robert` in the Ansible Tower interface, launch the `Full Stack Deployment Workflow` Job Template
- As the Linux user `student` on `tower.lab.example.com`, verify that the `Full Stack Deployment Workflow` Job Template sent you an email when it completed.
- Verify that the web servers have been installed and configured on `servera.lab.example.com`, `serverd.lab.example.com`, `servere.lab.example.com`, and `serverf.lab.example.com`
- Verify that the load balancer has been installed and configured on `serverb.lab.example.com`.

If it is working correctly, it should direct sequential HTTP requests sent to `http://serverb.lab.example.com` to different web servers. Every refresh of this page redirects the connection to a different web server.

If the configuration is correct, it should redirect connections to the following web servers: `servera`, `serverd`, `servere` and `serverf`.

1. To Launch the `Full Stack Deployment` Workflow as user `robert`:
  - 1.1. Log in to the Ansible Tower web interface as user `robert` with `redhat123` as the password.
  - 1.2. Click `Templates` in the left navigation bar.
  - 1.3. On the same line as the Workflow Job Template, `Full Stack Deployment`, click the rocket icon on the right to launch the Workflow. This opens the Survey you just created and ask for your input.

- 1.4. Click NEXT followed by LAUNCH to launch the Workflow. This redirects you to a detailed status page of the running Workflow.
- 1.5. Observe the running Jobs of the Workflow. You can click on the DETAILS link of a running or completed Job to see a more detailed live output of the Job.
2. To verify that the Workflow Full Stack Deployment triggered an email notification after completion:

- 2.1. Open a terminal and connect to the tower VM.

```
[student@workstation ~]$ ssh tower
Last login: Thu Apr 20 11:33:22 2018 from workstation.lab.example.com
[student@tower ~]$
```

- 2.2. View incoming messages to the local mailbox file of the student user using the **tail** command. You should see this type of successful Workflow Job completion notification email arrive:

```
[student@tower ~]$ tail -f /var/mail/student
...output omitted...
Message-ID: <20181203131145.5064.41602@tower.lab.example.com>
```

Workflow job summary:

```
- node #6 spawns job #15, "Dynamic Inventory - Custom Script", which finished with
status successful.
- node #7 spawns job #16, "My Full-Stack Project", which finished with status
successful.
- node #8 spawns job #17, "Set up Databases", which finished with status
successful.
- node #9 spawns job #19, "Set up Web servers", which finished with status
successful.
- node #10 spawns job #21, "Set up Load Balancer", which finished with status
successful.
...output omitted...
```

3. At this point, the web servers and the balancer are installed, configured, and functioning. Verify that the results are correct:

- 3.1. Open a web browser and go to `http://servera.lab.example.com`, `http://serverd.lab.example.com`, `http://servere.lab.example.com`, and `http://serverf.lab.example.com` in separate tabs. You should see this line at the bottom of each page:

```
...output omitted...
Deployment Version: Development
```

- 3.2. Open a web browser and go to `http://serverb.lab.example.com`. Every time you refresh the page, the load balancer redirects your request to one of the web servers verified in the previous step.

## Evaluation

As the student user on `workstation`, run the `lab review-review` script with the `grade` argument, to confirm success on this exercises. Correct any reported failures and rerun the script until successful.

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



## APPENDIX A

# ANSIBLE LIGHTBULB LICENSING

# ANSIBLE LIGHTBULB LICENSE

---

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

Copyright 2017 Red Hat, Inc.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.