



Advanced Automation: Red Hat Ansible Best Practices



Red Hat Ansible Engine 2.8 DO447
Advanced Automation: Red Hat Ansible Best Practices
Edition 2 20200818
Publication date 20190626

Authors: Artur Glogowski, Jordi Sola Alaball, Razique Mahroua,
Adolfo Vazquez Rodriguez, Eduardo Ramirez, Herve Quatremain,
Daniel Kolepp, Ricardo da Costa
Editor: Steven Bonneville, Nicole Muller, David Sacco, Jeff Tyson

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 send
email to training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

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

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

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

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

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

All other trademarks are the property of their respective owners.

Document Conventions	ix
Introduction	xi
Advanced Automation: Red Hat Ansible Best Practices	xi
Orientation to the Classroom Environment	xii
Internationalization	xvi
1. Developing with Recommended Practices	1
Implementing Recommended Practices	2
Guided Exercise: Implementing Recommended Practices	8
Managing Ansible Project Materials Using Git	11
Guided Exercise: Managing Ansible Project Materials Using Git	21
Lab: Developing with Recommended Practices	26
Summary	36
2. Managing Inventories	37
Writing YAML Inventory Files	38
Guided Exercise: Writing YAML Inventory Files	45
Managing Inventory Variables	48
Guided Exercise: Managing Inventory Variables	57
Lab: Managing Inventories	61
Summary	69
3. Managing Task Execution	71
Controlling Privilege Escalation	72
Guided Exercise: Controlling Privilege Escalation	78
Controlling Task Execution	83
Guided Exercise: Controlling Task Execution	91
Running Selected Tasks	96
Guided Exercise: Running Selected Tasks	101
Optimizing Execution for Speed	104
Guided Exercise: Optimizing Execution for Speed	116
Lab: Managing Task Execution	120
Summary	131
4. Transforming Data with Filters and Plugins	133
Processing Variables Using Filters	134
Guided Exercise: Processing Variables using Filters	144
Templating External Data using Lookups	150
Guided Exercise: Templating External Data using Lookups	155
Implementing Advanced Loops	161
Guided Exercise: Implementing Advanced Loops	167
Working with Network Addresses Using Filters	175
Guided Exercise: Working with Network Addresses Using Filters	180
Lab: Transforming Data with Filters and Plug-ins	183
Summary	191
5. Coordinating Rolling Updates	193
Delegating Tasks and Facts	194
Guided Exercise: Delegating Tasks and Facts	196
Managing Rolling Updates	200
Guided Exercise: Managing Rolling Updates	206
Lab: Coordinating Rolling Updates	215
Summary	221
6. Installing and Accessing Ansible Tower	223
Explaining the Red Hat Ansible Tower Architecture	224
Quiz: Explaining the Red Hat Ansible Tower Architecture	228

Installing Red Hat Ansible Tower	230
Guided Exercise: Installing Red Hat Ansible Tower	234
Navigating Red Hat Ansible Tower	236
Guided Exercise: Accessing Red Hat Ansible Tower	246
Quiz: Installing and Accessing Red Hat Ansible Tower	249
Summary	251
7. Managing Access with Users and Teams	253
Creating and Managing Ansible Tower Users	254
Guided Exercise: Creating and Managing Ansible Tower Users	260
Managing Users Efficiently with Teams	263
Guided Exercise: Managing Users Efficiently with Teams	267
Lab: Managing Access with Users and Teams	271
Summary	275
8. Managing Inventories and Credentials	277
Creating a Static Inventory	278
Guided Exercise: Creating a Static Inventory	288
Creating Machine Credentials for Access to Inventory Hosts	292
Guided Exercise: Creating Machine Credentials for Access to Inventory Hosts	299
Lab: Creating and Managing Inventories and Credentials	302
Summary	308
9. Managing Projects and Launching Ansible Jobs	309
Creating a Project for Ansible Playbooks	310
Guided Exercise: Creating a Project for Ansible Playbooks	319
Creating Job Templates and Launching Jobs	322
Guided Exercise: Creating Job Templates and Launching Jobs	330
Lab: Managing Projects and Launching Ansible Jobs	336
Summary	344
10. Constructing Advanced Job Workflows	345
Improving Performance with Fact Caching	346
Guided Exercise: Improving Performance with Fact Caching	349
Creating Job Template Surveys to Set Variables for Jobs	354
Guided Exercise: Creating Job Template Surveys to Set Variables for Jobs	360
Creating Workflow Job Templates and Launching Workflow Jobs	364
Guided Exercise: Creating Workflow Job Templates and Launching Workflow Jobs	371
Scheduling Jobs and Configuring Notifications	374
Guided Exercise: Scheduling Jobs and Configuring Notifications	379
Lab: Constructing Advanced Job Workflows	383
Summary	391
11. Communicating with APIs using Ansible	393
Launching Jobs with the Ansible Tower API	394
Guided Exercise: Launching Jobs with the Ansible Tower API	410
Interacting with APIs using Ansible Playbooks	416
Guided Exercise: Interacting with APIs using Ansible Playbooks	420
Lab: Communicating with APIs using Ansible	429
Summary	435
12. Managing Advanced Inventories	437
Importing External Static Inventories	438
Guided Exercise: Importing External Static Inventories	442
Creating and Updating Dynamic Inventories	446
Guided Exercise: Creating and Updating Dynamic Inventories	454
Filtering Hosts with Smart Inventories	456
Guided Exercise: Filtering Hosts with Smart Inventories	460

Lab: Managing Advanced Inventories	463
Summary	471
13. Creating a Simple CI/CD Pipeline with Ansible Tower	473
Creating a Simple CI/CD Pipeline with Ansible Tower	474
Guided Exercise: Integrating a GitLab CI/CD Pipeline with Ansible Tower	481
Summary	492
14. Performing Maintenance and Routine Administration of Ansible Tower	493
Performing Basic Troubleshooting of Ansible Tower	494
Guided Exercise: Performing Basic Troubleshooting of Ansible Tower	500
Configuring TLS/SSL for Ansible Tower	507
Guided Exercise: Configuring TLS/SSL for Ansible Tower	510
Backing Up and Restoring Ansible Tower	512
Guided Exercise: Backing Up and Restoring Ansible Tower	516
Quiz: Performing Maintenance and Routine Administration of Ansible Tower	519
Summary	521
15. Comprehensive Review	523
Comprehensive Review	524
Lab: Refactoring Inventories and Projects	528
Lab: Privilege Escalation, Lookups, and Rolling Updates	535
Lab: Restoring Ansible Tower from Backup	544
Lab: Adding Users and Teams	547
Lab: Creating a Custom Dynamic Inventory	552
Lab: Configuring Job Templates	558
Lab: Configuring Workflow Job Templates, Surveys, and Notifications	568
Lab: Testing the Prepared Environment	577

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

Advanced Automation: Red Hat Ansible Best Practices

Advanced Automation: Red Hat Ansible Best Practices (DO447) is designed for experienced Red Hat Ansible Automation users who want to take their Ansible skills to the next level, enabling scalable design and operation of Ansible automation in the enterprise.

Course Objectives

- Follow recommended practices to write and manage Ansible automation
- Leverage advanced features of Ansible to perform complex tasks
- Deploy and use Red Hat Ansible Tower to manage existing Ansible projects, playbooks, and roles at scale
- Effectively manage playbooks and inventories as part of a DevOps workflow

Audience

- System administrators, DevOps engineers, and network administrators interested in central management of their Ansible projects and the execution of Ansible Playbooks in a large-scale environment.

Prerequisites

- Red Hat Certified System Administrator (EX200 / RHCSA) certification or equivalent Red Hat Enterprise Linux knowledge and experience.
- Red Hat Certified Specialist in Ansible Automation (EX407) certification, Red Hat Certified Engineer (RHCE) certification on Red Hat Enterprise Linux 8 (EX294), or equivalent Ansible experience.

Orientation to the Classroom Environment

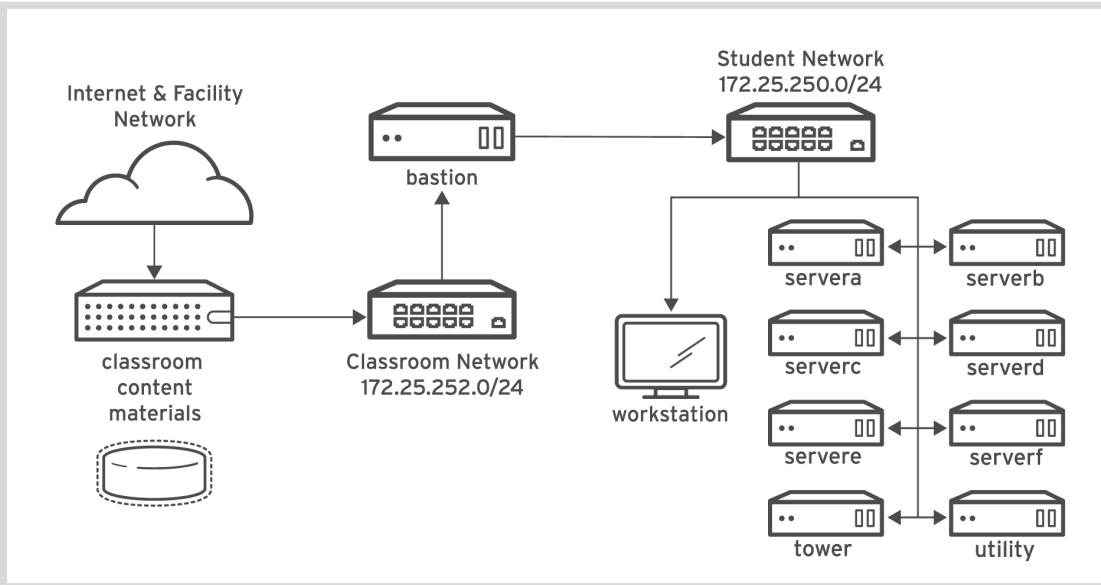


Figure 0.1: Classroom environment

In this course, the main computer system used for hands-on learning activities is **workstation**. Eight other machines are also used by students for these activities: **servera**, **serverb**, **serverc**, **serverd**, **servere**, **serverf**, **tower**, and **utility**. 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**. There is also a special **devops** user account, which has the password **redhat**.

Classroom Machines

Machine name	IP addresses	Role
content.example.com, materials.example.com, classroom.lab.example.com	172.25.254.254, 172.25.252.254	Classroom utility server
bastion.lab.example.com	172.25.250.254	Gateway system to connect student private network to classroom server (must always be running)
workstation.lab.example.com	172.25.250.9	Graphical workstation used for system administration
servera.lab.example.com	172.25.250.10	Managed server "A"

Machine name	IP addresses	Role
serverb.lab.example.com	172.25.250.11	Managed server "B"
serverc.lab.example.com	172.25.250.12	Managed server "C"
serverd.lab.example.com	172.25.250.13	Managed server "D"
servere.lab.example.com	172.25.250.14	Managed server "E"
serverf.lab.example.com	172.25.250.15	Managed server "F"
tower.lab.example.com	172.25.250.7	Ansible Tower server
utility.lab.example.com	172.25.250.8	Student server for supporting services

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

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

Controlling Your Systems

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

Controlling the Virtual Machines

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

Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots into the same state as when it was shut down (the disk will have been preserved).

Virtual Machine State	Description
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

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

Classroom/Machine Actions

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

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

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

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



Warning

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

The Autostop Timer

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

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

Internationalization

Per-user Language Selection

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

Language Settings

In the GNOME desktop environment, the user might 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.

You can start this application in two ways. You can run the command **gnome-control-center region** from a terminal window, or on the top bar, from the system menu in the right corner, select the settings button (which has a crossed screwdriver and wrench for an icon) from the bottom left of the menu.

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 you log in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications such as **gnome-terminal** that are started inside it. However, by default they do not apply to that account if accessed through an **ssh** login from a remote system or a text-based login on a virtual console (such as **tty5**).



Note

You can make your shell environment use the same **LANG** setting as your graphical environment, even when you log in through a text-based virtual console or over **ssh**. One way to do this is to place code similar to the following in your **~/.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=$(grep 'Language=' /var/lib/AccountsService/users/${USER} \
    | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, and other languages with a non-Latin character set might not display properly on text-based virtual 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 25 17:55:01 CET 2019
```

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 or later 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 you know the character's Unicode code point. Type **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.

To set the system-wide default language, run the command **localectl set-locale** **LANG=locale**, where *locale* is the appropriate value for the **LANG** environment variable 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 by clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the graphical login screen will also adjust the system-wide default language setting stored in the **/etc/locale.conf** configuration file.



Important

Text-based virtual consoles such as **tty4** are more limited in the fonts they can display than terminals in a virtual console running a graphical environment, or pseudoterminals for **ssh** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a text-based virtual console. For this reason, you should consider using English or another language with a Latin character set for the system-wide default.

Likewise, text-based virtual 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 text-based virtual consoles and the graphical environment. See the **localectl(1)** and **vconsole.conf(5)** man pages for more information.

Language Packs

Special RPM packages called *langpacks* install language packages that add support for specific languages. These langpacks use dependencies to automatically install additional RPM packages containing localizations, dictionaries, and translations for other software packages on your system.

To list the langpacks that are installed and that may be installed, use **yum list langpacks-***:

```
[root@host ~]# yum list langpacks-*  
Updating Subscription Management repositories.  
Updating Subscription Management repositories.  
Installed Packages  
langpacks-en.noarch      1.0-12.el8      @AppStream  
Available Packages  
langpacks-af.noarch       1.0-12.el8      rhel-8-for-x86_64-appstream-rpms  
langpacks-am.noarch       1.0-12.el8      rhel-8-for-x86_64-appstream-rpms  
langpacks-ar.noarch       1.0-12.el8      rhel-8-for-x86_64-appstream-rpms  
langpacks-as.noarch       1.0-12.el8      rhel-8-for-x86_64-appstream-rpms  
langpacks-ast.noarch      1.0-12.el8      rhel-8-for-x86_64-appstream-rpms  
...output omitted...
```

To add language support, install the appropriate langpacks package. For example, the following command adds support for French:

```
[root@host ~]# yum install langpacks-fr
```

Introduction

Use **yum repoquery --whatsonplements** to determine what RPM packages may be installed by a langpack:

```
[root@host ~]# yum repoquery --whatsonplements langpacks-fr
Updating Subscription Management repositories.
Updating Subscription Management repositories.
Last metadata expiration check: 0:01:33 ago on Wed 06 Feb 2019 10:47:24 AM CST.
glibc-langpack-fr-0:2.28-18.el8.x86_64
gnome-getting-started-docs-fr-0:3.28.2-1.el8.noarch
 hunspell-fr-0:6.2-1.el8.noarch
 hyphen-fr-0:3.0-1.el8.noarch
 libreoffice-langpack-fr-1:6.0.6.1-9.el8.x86_64
 man-pages-fr-0:3.70-16.el8.noarch
 mythes-fr-0:2.3-10.el8.noarch
```

**Important**

Langpacks packages use RPM *weak dependencies* in order to install supplementary packages only when the core package that needs it is also installed.

For example, when installing *langpacks-fr* as shown in the preceding examples, the *mythes-fr* package will only be installed if the *mythes* thesaurus is also installed on the system.

If *mythes* is subsequently installed on that system, the *mythes-fr* package will also automatically be installed due to the weak dependency from the already installed *langpacks-fr* package.

**References**

locale(7), **localectl(1)**, **locale.conf(5)**, **vconsole.conf(5)**, **unicode(7)**, and **utf-8(7)** 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

**Note**

This table might not reflect all langpacks available on your system. Use **yum info langpacks-SUFFIX** to get more information about any particular langpacks package.

Language Codes

Language	Langpacks Suffix	\$LANG value
English (US)	en	en_US.utf8

Language	Langpacks Suffix	\$LANG value
Assamese	as	as_IN.utf8
Bengali	bn	bn_IN.utf8
Chinese (Simplified)	zh_CN	zh_CN.utf8
Chinese (Traditional)	zh_TW	zh_TW.utf8
French	fr	fr_FR.utf8
German	de	de_DE.utf8
Gujarati	gu	gu_IN.utf8
Hindi	hi	hi_IN.utf8
Italian	it	it_IT.utf8
Japanese	ja	ja_JP.utf8
Kannada	kn	kn_IN.utf8
Korean	ko	ko_KR.utf8
Malayalam	ml	ml_IN.utf8
Marathi	mr	mr_IN.utf8
Odia	or	or_IN.utf8
Portuguese (Brazilian)	pt_BR	pt_BR.utf8
Punjabi	pa	pa_IN.utf8
Russian	ru	ru_RU.utf8
Spanish	es	es_ES.utf8
Tamil	ta	ta_IN.utf8
Telugu	te	te_IN.utf8

Chapter 1

Developing with Recommended Practices

Goal

Demonstrate and implement recommended practices for effective and efficient Ansible automation.

Objectives

- Demonstrate and describe common recommended practices for developing and maintaining effective Ansible automation solutions.
- Create and manage Ansible Playbooks in a Git repository using recommended practices.

Sections

- Implementing Recommended Practices (and Guided Exercise)
- Managing Ansible Project Materials Using Git (and Guided Exercise)

Lab

- Developing with Recommended Practices

Implementing Recommended Practices

Objectives

After completing this section, you should be able to demonstrate and describe commonly recommended practices for developing and maintaining effective Ansible automation solutions.

Describing Ansible Effective Usage

This course assumes that you have some experience using Ansible to automate Linux system administration tasks. Ansible is easy to learn and easy to start using. However, as you work with more advanced features and larger, more complex projects, it becomes more difficult to manage and maintain Ansible Playbooks, or to use them effectively.

This course introduces a number of advanced features of Red Hat Ansible Automation Platform. How to use Red Hat Ansible Engine effectively and efficiently is also discussed. Using Ansible effectively is not just about features or tools, but about practices and organization.

To paraphrase Ansible developer Jeff Geerling, the three key practices to effectively use Ansible are to

- Keep Things Simple
- Stay Organized
- Test Often

Keeping Things Simple

One of Ansible's strengths is its simplicity. If you keep your playbooks simple, they will be easier to use, modify, and understand.

Keeping Your Playbooks Readable

Keep your playbooks well commented and easy to read. Use vertical white space and comments liberally. Always give plays and tasks meaningful names that make clear what the play or task is doing. These practices help document the playbook and makes it easier to troubleshoot a failed playbook run.

YAML is not a programming language. It is good at expressing a list of tasks or items, or a set of key-value pairs. If you are struggling to express some complex control structure or conditional in your Ansible Playbook, then consider approaching the problem differently. There are clever things you can do with templates and Jinja2 filters to process data in a variable, which might be a better approach to your problem.

Use native YAML syntax, not the "folded" syntax. For example, the following example is not a recommended format:

```

- name: Postfix is installed and updated
  yum: name=postfix state=latest
  notify: restart_postfix

- name: Postfix is running
  service: name=postfix state=started

```

The following syntax is easier for most people to read:

```

- name: Postfix is installed and updated
  yum:
    name: postfix
    state: latest
  notify: update_postfix

- name: Postfix is running
  service:
    name: postfix
    state: started

```

Use Existing Modules

When writing a new playbook, start with a basic playbook and, if possible, a static inventory. Use **debug** modules as stubs as you build your design. Once your playbook functions as expected, break up your playbook into smaller, logical components using imports and includes.

Use special-purpose modules included with Ansible when you can, instead of **command**, **shell**, **raw**, or other similar modules. While there are circumstances where you need to use these general purpose modules, it will be a lot easier to make your playbook idempotent and simple to maintain if you use the modules designed for a specific task.

Many modules have a default **state** or other variable that controls what they do. For example, the **yum** module currently assumes that the package you name should be **present** in most cases. However, you should explicitly specify what **state** you want. Doing so makes it easier to read the playbook, and protects you against changes in the module's default behavior in later versions of Ansible.

Adhering to a Standard Style

You should consider having a standard "style" that you, and your coworkers, follow when writing Ansible projects. How many spaces do you indent? How do you want vertical white space to be used? How should tasks, plays, roles, and variables be named? What should be commented on and how? There is more than one reasonable way to do this, but having a consistent standard can help improve maintainability and readability.

Staying Organized

Organization of your Ansible projects and how you run playbooks can help with maintainability, troubleshooting, and auditing.

Following Conventions for Naming Variables

Variable naming can be particularly important because Ansible has a fairly flat namespace. Use descriptive variables, such as **apache_tls_port** rather than a less descriptive variable such as

In roles, it is a good practice to prefix role variables with the role name. For example, if your role is named **myapp** then your variable names could start with **myapp_** to help namespace them from variables in other roles and the playbook.

Variable names should clarify contents. A name like **apache_max_keepalive** clearly explains the meaning of the associated value (or values). Prefix roles and group variables with the name of the role or group to which the variable belongs. **apache_port_number** is more error-resistant than **port_number**.

Standardizing the Project Structure

Use a consistent pattern when structuring the files of your Ansible project on a file system. There are a number of useful patterns, but the following is a good example:

```
.
├── dbservers.yml
└── inventories/
    ├── prod/
    │   ├── group_vars/
    │   ├── host_vars/
    │   └── inventory/
    ├── stage/
    │   ├── group_vars/
    │   ├── host_vars/
    │   └── inventory/
    └── roles/
        └── std_server/
└── site.yml
└── storage.yml
└── webservers.yml
```

The file **site.yml** is the master playbook, which includes or imports playbooks that perform specific tasks: **dbservers.yml**, **storage.yml**, and **webservers.yml**. The roles are located in subdirectories in the **roles** directory, such as **std_server**. There are two static inventories in the **inventories/prod** and **inventories/stage** directories with separate inventory variable files, so that you can select different sets of servers by changing the inventory you use.

One of the benefits of the playbook structure is that you can divide up your large playbook into smaller files to make it more readable, and those smaller sub-playbooks can contain plays for a specific purpose that you can run independently.

Using Dynamic Inventories

Use dynamic inventories whenever possible. Dynamic inventories allow central management of your hosts and groups from one central source of truth, and ensure that the inventory is automatically updated. Dynamic inventories are especially powerful in conjunction with cloud providers, containers, and virtual machine management systems. Those systems might already have inventory information in a form Ansible can consume.

If you are unable to use dynamic inventories, then other tools can help you dynamically construct groups or other information. For example, you can use the **group_by** module to dynamically generate group membership based on a fact. That group membership is valid for the rest of the playbook.

```

- name: Generate dynamic groups
  hosts: all
  tasks:
    - name: Generate dynamic groups based on architecture
      group_by:
        key: arch_ "{{ ansible_facts['architecture'] }}"
    - name: Configure x86_64 systems
      hosts: arch_x86_64
      tasks:
        - name: First task for x86_64 configuration
          ...output omitted...

```

Taking Advantage of Groups

Hosts can be a member of multiple groups. Consider dividing your hosts into different categories based on different characteristics:

- Geographical: Differentiate hosts from different regions, countries, continents, or data centers.
- Environmental: Differentiate hosts dedicated to different stages of the software life cycle, including development, staging, testing, or production.
- Sites or services: Group hosts that offer or link to a subset of functions, like a specific website, an application, or a subset of features.



Important

Remember that hosts will inherit variables from all groups of which they are members. If two groups have different settings for the same variable, and a host is a member of both, then the value that is used is the last one loaded.

If there are differences in settings between two different groups that might be used at the same time, then take special care to determine how those variables should be set.

Using Roles for Reusable Content

Roles help you keep your playbooks simple and allow you to save work by reusing common code across projects. If you are writing your own roles, then keep them focused on a particular purpose or function similar to playbooks. Make roles generic and configurable through variables, so that you do not need to edit them when you use them with a different set of playbooks.

Use the **ansible-galaxy** command to initialize your role's directory hierarchy and provide initial template files. This will also make it easier for you to share your role on the Ansible Galaxy website, if you choose to do so.

The roles provided by the *redhat-system-roles* package in Red Hat Enterprise Linux are officially supported (although some roles might be in Tech Preview). Review the roles provided to determine if they are useful for you.

You can also look at the roles provided by the community through Ansible Galaxy. Be aware that these roles have varying levels of quality, so choose the ones you use carefully.

Keep your roles in the **roles** subdirectory of your project. Use the **ansible-galaxy** command to automatically get roles from a separate Git repository, even if the roles are not from Ansible Galaxy but are stored in your own repository.

Running Playbooks Centrally

To control access to your systems and audit Ansible activity, consider using a dedicated control node from which all Ansible Playbooks are run.

System administrators should still have their own accounts on the system, as well as credentials to connect to managed hosts and escalate privileges, if needed. When a system administrator leaves, their SSH key can be removed from managed hosts' **authorized_keys** file and their **sudo** command privileges revoked, without impacting other administrators.

Consider using Red Hat Ansible Tower as this central host. Red Hat Ansible Tower is included with a new Red Hat Ansible Automation subscription, and provides features that make it easier to control access to credentials, control playbook execution, simplify automation for users who are not comfortable with the Linux command line, as well as audit and track playbook runs. Later in this course, you will learn about using Red Hat Ansible Tower. However, even if you do not use Red Hat Ansible Tower, using a central control node can be beneficial.

Testing Often

Test your playbooks and your tasks frequently during the development process, when the tasks run, and once the playbooks are in use.

Testing the Results of Tasks

Should you need to confirm that a task succeeded, verify the result of the task rather than trusting the return code of the module. There is more than one way to do verify a task, depending on the module involved.

```
- Start web server
  service:
    name: httpd
    status: started

- name: Check web site from web server
  uri:
    url: http://{{ ansible_fqdn }}
    return_content: yes
  register: example_webpage
  failed_when: example_webpage.status != 200
```

Using Block/Rescue to Recover or Rollback

The **block** directive is useful for grouping tasks; when used in conjunction with the **rescue** directive, it is helpful when recovering from errors or failures.

```
- block:
  - name: Check web site from web server
    uri:
      url: http://{{ ansible_fqdn }}
      return_content: yes
    register: example_webpage
```

```

    failed_when: example_webpage.status != 200
rescue:
  - name: Restart web server
    service:
      name: httpd
      status: restarted

```

Develop Playbooks with the Latest Ansible Version

Even if you are not using the latest version of Ansible in production, you should routinely test your playbooks against the latest version of Ansible. This test will help you avoid issues as Ansible modules and features evolve.

If your playbooks print warnings or deprecation messages when they run, then you should pay attention to them and make adjustments. Generally, if a feature in Ansible is being deprecated or is changing, the project provides deprecation notices four minor releases before the feature is removed or changed.

To prepare for changes and future updates, read the Porting Guides at https://docs.ansible.com/ansible/latest/porting_guides/porting_guides.html.

Using Test Tools

A number of commands and tools are available to help you test your playbooks. Use the **ansible-playbook --syntax-check** command to check the syntax of your playbook without running it.

Use the **ansible-playbook --check** command to run your playbook against the actual managed hosts in "check mode," to see what changes the playbook would make. This check does not guarantee perfect accuracy, since the playbook might need to actually run some tasks before subsequent tasks in the playbook will work correctly. You may have some tasks that are marked with the **check_mode: no** directive. Those tasks will run even in check mode.



Note

There are other useful tools that are not currently shipped in Red Hat Enterprise Linux 8; these tools are included in Fedora, or can be obtained from upstream sources. For example:

The **ansible-lint** tool parses your playbook and looks for possible issues. Not all the issues it reports will necessarily break your playbook, but reported issues may indicate the presence of an error.

The **yamllint** tool parses a YAML file and attempts to identify issues with the YAML syntax. This tool does not have direct knowledge of Ansible, but it can catch potential YAML syntax problems.



References

Ansible Documentation: Best Practices

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

► Guided Exercise

Implementing Recommended Practices

In this exercise, you will apply recommended practices to an already existing Ansible project.

Outcomes

You should be able to identify points of improvement on Ansible projects and implement recommended practices.

Before You Begin

Log in as the **student** user on **workstation** and run the **lab development-practices start** command.

```
[student@workstation ~]$ lab development-practices start
```

This script creates an Ansible Project in **~/D0447/labs/development-practices**.

Steps

- ▶ 1. Get familiar with the project and its current status. Start by changing to the project directory.

```
[student@workstation ~]$ cd ~/D0447/labs/development-practices
```

- ▶ 2. To avoid conflicts between the variables used by the **haproxy** role, which you maintain, and your playbooks, you must put the role's variables in their own namespace. You have decided that all variables internal to the role should start with the string **haproxy_**.

Navigate through the folder structure and review the defined roles in the **roles** directory. Give role variables a name that references the role to which the variable belongs. Review the **roles/haproxy/defaults/main.yml** file and then add the **haproxy_** prefix to all variable names. File contents should display as follows:

```
# Log-level for HAProxy logs
haproxy_log_level: info

# Port exposed to clients
haproxy_port: 80

# Name for the default backend
haproxy_backend_name: app

# Port backend is exposed to
haproxy_backend_port: 80
...output omitted...
# The default is no defined backend servers.
haproxy_appservers: []
```

```
...output omitted...
# Socket used to communicate with haproxy service. DO NOT CHANGE
haproxy_socket: /var/run/haproxy.sock
```

The **roles/haproxy/templates/haproxy.cfg.j2** template file references the variables defined in the **roles/haproxy/defaults/main.yml** file. Update the variable names in this template accordingly.

```
...output omitted...
global
    #Send events/messages to rsyslog server.
    log          127.0.0.1:514 local0 {{ haproxy_log_level }}
...output omitted...
    # turn on stats unix socket
    # required for the ansible haproxy module.
    stats socket {{ haproxy_socket }} level admin
...output omitted...

frontend main
    mode http
    bind *:{{ haproxy_port }}
    default_backend {{ haproxy_backend_name }}
...output omitted...

backend {{ haproxy_backend_name }}
    balance roundrobin
{% for server in haproxy_appservers %}
    server {{ server.name }} {{ server.ip }}:{{ haproxy_backend_port }}
{% endfor %}
```

- 3. The **haproxy** role uses the **haproxy_appservers** variable to configure a pool of back end application servers. Your playbook must define this variable in order to use the role correctly because the default value of the **haproxy_appservers** variable is an empty list.

In your project directory, the **appservers.yml** file has the correct settings for the **haproxy_appservers** variable. Correct the variable name and configure it as a group variable for the **lb_servers** group.

- 3.1. Create the **group_vars** folder in the project directory. Copy the **appservers.yml** file from the project directory to the **group_vars/lb_servers.yml** file.

```
[student@workstation development-practices]$ mkdir group_vars
[student@workstation development-practices]$ cp -v \
> appservers.yml group_vars/lb_servers.yml
'appprovers.yml' -> 'group_vars/lb_servers.yml'
```

- 3.2. Edit the **group_vars/lb_servers.yml** file and rename the **appservers** variable to **haproxy_appservers**. When you finish editing, the file displays as follows:

```
haproxy_appservers:
  - name: serverb.lab.example.com
    ip: "172.25.250.11"
  - name: serverc.lab.example.com
    ip: "172.25.250.12"
```

- 4. Configure web applications hosted in Europe to provide a different message than other locations by overriding a role default with a group variable.

4.1. Add a new inventory group named **region_eu** to the end of the **inventory** file.

```
[region_eu]  
serverc.lab.example.com
```

- 4.2. Create a groups variable file named **group_vars/region_eu.yml**. Make sure the name of the file matches the inventory group created in the previous step. This file should only contain a single line that defines a **webapp_message** variable:

```
webapp_message: "Hello from Europe. This is"
```

Note that this variable applies only to hosts in the **region_eu** group. All other hosts use a default value for the **webapp_message** variable, which is defined in the **roles/webapp/defaults/main.yml** file.

- 5. Test your changes to the Ansible project by running your **site.yml** playbook:

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

Validate that **serverb.lab.example.com** responds with the default content, while **serverc.lab.example.com** responds with content defined in the **region_eu** group:

```
[student@workstation development-practices]$ curl servera; curl servera  
This is serverb.lab.example.com. (version v1.0)  
Hello from Europe. This is serverc.lab.example.com. (version v1.0)
```

The order of the output that you see might differ, since **servera.lab.example.com** redirects HTTP requests to **serverb.lab.example.com** and **serverc.lab.example.com** in a round-robin manner.

- 6. Run the **clean.yml** Ansible Playbook to revert all changes made by the previous steps.

```
[student@workstation development-practices]$ ansible-playbook clean.yml
```

Finish

On **workstation**, run the **lab development-practices finish** script to complete this lab.

```
[student@workstation ~]$ lab development-practices finish
```

This concludes the guided exercise.

Managing Ansible Project Materials Using Git

Objectives

After completing this section, you 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*. Instead of managing your infrastructure manually, you define and build your systems by running your automation code. Red Hat Ansible Automation is a key tool that can help you implement this approach.

If Ansible projects are the code which is used to define the infrastructure, then a *version control system* such as Git should be used to track and control changes to the code.

Version control also allows you to implement a life cycle for the different stages of your infrastructure code, such as development, QA, and production. You can commit your changes to a branch and test those changes in noncritical development and QA environments. Once you are confident in the changes, you can merge them to the main production code and apply the changes to your production infrastructure.

Introducing Git

Git is a *distributed version control system (DVCS)* that allows developers to manage changes to files in a project collaboratively. 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, and when those changes were made.
- 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*. Cloning a project creates a complete copy of the original remote repository as a *local repository*. This is a local copy of the entire history of the files in the version control system, and not just the latest snapshot of the project files.

The developer makes edits in a *working tree*, which is 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 *push* changes to the remote repository. Alternatively, if the local repository is accessible from the network, the owner of the remote repository can *pull* the changes from the developer's local 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, which fetches them from the remote repository and merges them into the local repository.

To use Git effectively, a user must be aware of the three possible states of a file in the working tree: *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 1.1: 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.

Describing 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 this command 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.

```
[user@demo ~]$ git config --global user.name 'Peter Shadowman'
[user@demo ~]$ git config --global user.email peter@example.com
```

If **bash** is the user's shell, another useful, optional setting 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 are not 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 a 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**:

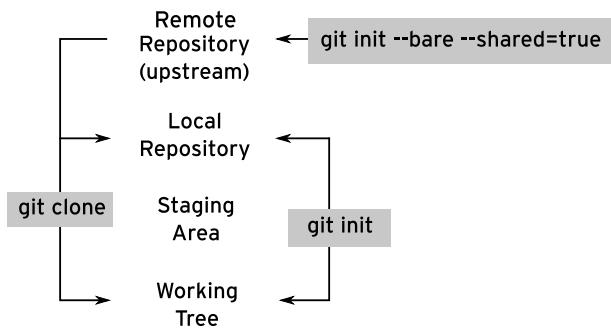
```
[user@demo ~]$ git clone git@git.lab.example.com:project.git
```



Note

Another way to start the Git workflow is to create a new, private project with the **git init** command. When a project is started in this way, no remote repository is created.

Use **git init --bare** to create a *bare repository* on a server. Bare repositories does not have a local working tree. Therefore, the bare repository cannot be used to apply file changes. Git servers usually contain bare repositories, as the working tree is not needed in the server. When developers create a local copy of the repository, they usually need a working tree to make local changes, so they create a non-bare clone. The server must also be set up to allow users to clone, pull from, and push to the repository using the HTTPS or SSH protocol.

**Figure 1.2: 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 to 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.



Note

Meaningful and concise commit messages are the key to maintaining a clear history for the Ansible Project. There are many approaches to a good commit message, but most of these approaches agree on the following three points:

- First line should be a short (usually less than 50 characters) summary of the reason for the commit.
- A blank line follows, and then the rest of the message must explain all the details and reasons for the commit.
- If available, add references to an issue or feature tracker related entries. These references expand the commit message with additional text, related people or history.

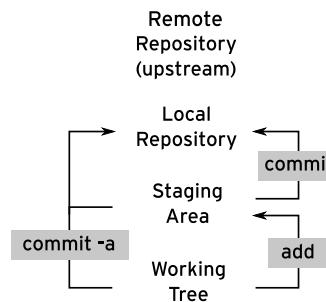


Figure 1.3: Git subcommands that add/update local repository content

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.

```
[user@demo ~]$ git config --global push.default simple
```

The **git pull** command fetches commits from the remote repository and adds them to the local repository. It also merges changes into the files in 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 your local repository, then to use **git merge** to merge the changes in the tracking branch to your current branch.

Git branches are discussed later in this section.

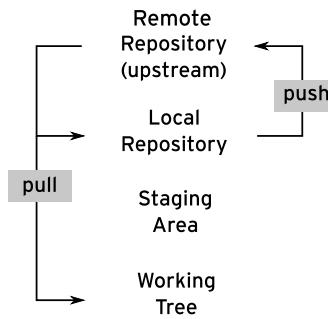


Figure 1.4: Git subcommands that interact with a remote repository

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 does not need to be entered with the command, but 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 the working tree.
<code>git add file</code>	Stage a new or changed 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 that have been staged for the next commit.
<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.
<code>git revert commit_ref</code>	Create a new commit, undoing the changes in the commit referenced. The commit hash that identifies the commit can be used, although there are other ways to reference a commit.



Important

This is 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 include:

- 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.

Git Branches and References

Changes in a Git repository bundle into *commits*. A commit contains all the information needed by Git to create and handle the whole history for the repository, including:

- A unique *ID* for the commit, in the format of a 40 hexadecimal character string. This ID is the SHA-1 hash of the contents of the commit.
- The list of repository files changed, and the exact changes to each of them. Changes may be line additions or subtractions, renaming, or deletion.
- The ID of the parent commit. That is, the ID for the commit defining the status of the repository before applying current commit changes.
- The author and the creator (or *committer*) for the commit.
- Lastly, but very important, a commit also includes a list of *references*. A reference is like a named pointer to the commit. Most common references are *tags* and *branches*.

The **git commit** command generates a new commit with all changes added to the stage with the **git add** command. You can see a Git repository as a commit graph, and branches as references to commits.

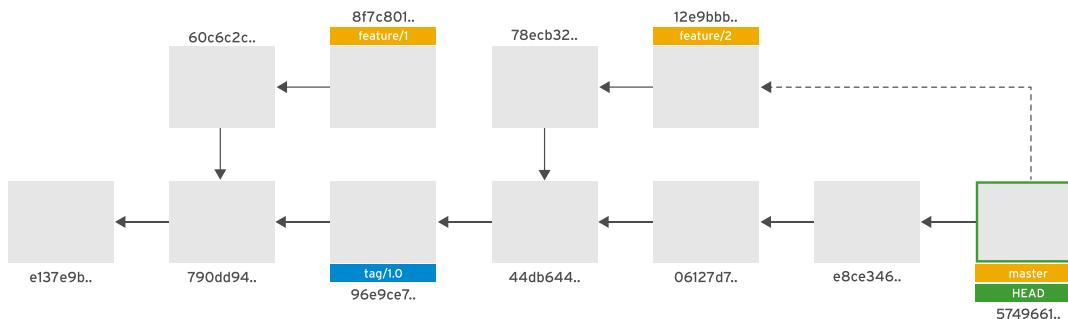


Figure 1.5: Sample Git Repository

The preceding figure depicts a sample Git repository containing 11 commits. The most recent commits are to the right, the arrows point to the older commits. It has three branches (**master**, **feature/1**, and **feature/2**) and a single tag (**tag/1.0**).

The **HEAD** reference is the current commit in the local working tree. If you make a change in your working tree, stage it with the **git add** command, and commit it with the **git commit** command, a new commit will be created with most recent commit as its parent, and **HEAD** will move to point at the new commit.

Creating Branches

Different branches in Git allow different work streams to evolve in parallel on the same Git repository. Commits for each work stream append only to that branch.

Use the **git branch** command to create a new branch from the current **HEAD** commit. This command creates a reference for the new branch, but it does not set the current **HEAD** to this branch. Use the **git checkout** command to move the **HEAD** to the appropriate branch.

In the preceding example, the most recent commit for the branch **master** (and **HEAD** at that time) was commit **5749661**, which occurred at some point in the past. A user ran the **git branch feature/1** command, creating a branch, **feature/1**. Then, the user ran **git checkout feature/1** to indicate that future commits should be made to that branch. Finally, the user staged and committed two commits to the **feature/1** branch, commits **60c6c2c** and **8f7c801**. After running these commands, the **HEAD** was at commit **8f7c801**, and the new commits were added to the **feature/1** branch.

Next, the user wanted to add commits to the **master** branch. After moving **HEAD** to the latest commit on that branch, by running the **git checkout master** command, two new commits were made: **96e9ce7** and **44bd644**. The **HEAD** pointed to **44bd644**, and commits were added to the **master** branch.

Merging Branches

When work is complete on a branch, the branch can be *merged* with the original branch. This allows work in parallel on new features and bug fixes, while the main branch is free of incomplete or untested work.

In the preceding example, it was determined that all changes from **feature/2** should be merged into the **master** branch. That is, someone wanted the most recent commit for **master** to be the result of all the commits from both branches.

This outcome was accomplished by first running the **git checkout master** command to make sure **HEAD** was on the most recent commit on the **master** branch, **e8ce346**. Then, the command **git merge feature/2** was run. This command created a new commit, **5749661**, that included all the commits from **feature/2** and all the commits from **master**. **HEAD** was moved to the latest commit.

Sometimes, changes on more than one branch can not merge automatically because each branch makes changes to the same parts of the same files. This situation is called a *merge conflict* and needs to be manually resolved by editing the affected files. Merge strategies and conflict resolution are out of scope for this course.

Creating Branches from Old Commits

Use the **git checkout** command to move **HEAD** to any commit. For example, the command **git checkout 790dd94** moves **HEAD** to that commit. Then, a new branch starting with that commit can be created using the **git branch** and **git checkout** commands. For example, one might choose to branch from a specific commit to ignore other recent changes to a branch.

You can create a branch and switch to it in a single step, by running the **git checkout** command with the **-b** option:

```
[user@demo ~]$ git checkout -b feature/2
Switched to a new branch 'feature/2'
```

**Note**

If there is a commit that you might need at some point in the future, then you can use **git tag** to put a memorable label on it (like **tag/1.0** in the example) and then run the **git checkout** command on that tag to switch to the desired commit.

Pushing Branches to Remote Repositories

Initially, any branches that you create only exist in your local repository. If you want to share them with users of the original remote repository that you cloned, then you must push them to the repository as you would push a commit.

The most common way to do this is to use the **git push** command with the **--set-upstream** (or **-u**) option to create a branch on the remote repository that will be tracked by your current local branch.

For example, to push your new branch, **feature/2**, to your remote repository, run the following commands:

```
[user@demo ~]$ git checkout feature/2
Switched to branch 'feature/2'
[user@demo ~]$ git push --set-upstream origin feature/2
```

The remote **origin** repository normally refers to the Git repository that you originally cloned. This command indicates that you want to push the current branch to the repository, creating **feature/2** and configuring Git so that the **git push** command on this branch pushes commits to the remote repository. Running the **git pull** command on this branch will pull and merge any commits from the remote repository into your local copy of the branch.

Structuring Ansible Projects in Git

Each Ansible project should have its own Git repository.

The 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/
  webserver/      # directory for roles
    tasks/          # a particular role
      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...

```

Roles take some thought to manage well. If your roles are managed as part of the playbook, then keeping them with the playbook can make sense. However, most roles are meant to be shared by multiple projects, and if each project has its own copy of each role, then they could diverge from each other and some benefit of using roles will be lost.

A role should have its own Git repository. Some people try to include role repositories in their project repositories by using an advanced feature of Git called *submodules*. Because submodules can be tricky to manage successfully, this is not a recommended approach.

A better approach is to set up your project with an empty **roles/** directory, and use **ansible-galaxy** to populate this directory with the latest version of roles from their Git repositories before you run the playbooks.

Ensure that no subdirectories get committed to **roles/** by putting a **README** file in the directory explaining what you are doing and committing a **.gitignore** file to the top of your project's Git repository to tell Git to ignore directories in **roles/**:

```
roles/**/*
```



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 User Guide: Best Practices

http://docs.ansible.com/ansible/playbooks_best_practices.html

How to Write a Git Commit Message

<https://chris.beams.io/posts/git-commit/>

► 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 those changes to the original repository.

Outcomes

You should be able to use basic **git** commands to modify files stored in an existing Git repository.

Before You Begin

Log in as the **student** user on **workstation** and run the **lab development-git start** command. This setup script installs the package for Git and creates the Git repository needed for the exercise.

```
[student@workstation ~]$ lab development-git start
```

- 1. Perform basic configuration of Git by setting the following user name, email address, and default push method using the **git config** command:

```
[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 the **git config** command. The output should resemble the following:

```
[student@workstation ~]$ git config --global -l  
user.name=Daniel George  
user.email=daniel@lab.example.com  
credential.helper=store
```

- 3. Create a new directory named **git-repos** for your Git repositories, and then change to that directory.

```
[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 HTTP server.

```
[student@workstation git-repos]$ git clone \
> http://git@git.lab.example.com:8081/git/my_webservers_DEV.git
Cloning into 'my_webservers_DEV'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), 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 provided by the server.

```
[student@workstation my_webservers_DEV]$ tree
.
├── apache-setup.yml
└── templates
    ├── httpd.conf.j2
    └── index.html.j2

1 directory, 3 files
```

- 5. Create a new branch named **development**, and then switch to that branch.

```
[student@workstation my_webservers_DEV]$ git branch development
[student@workstation my_webservers_DEV]$ git checkout development
Switched to branch 'development'
```

- 6. Edit the **index.html.j2** template so that it displays **HELLO WORLD** at the bottom of the page, and then verify your modification.

- 6.1. Using a text editor, open the **templates/index.html.j2** template file for editing. Append the string **HELLO WORLD
** to the end of the file. After completing these modifications, the **index.html.j2** file should display as follows:

```
{{ 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>
```

- 6.2. Save the changes made to the file and then exit the text editor.

- 6.3. 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 development
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")
```

- 7. Add the template to the staging area and check the status of your modifications.

- 7.1. Use the **git add** command to add the template to the staging area.

```
[student@workstation my_webservers_DEV]$ git add templates/index.html.j2
```

- 7.2. Use the **git status** command to check the status of your modifications. Git shows that you have modifications in the staging area that are ready to be committed.

```
[student@workstation my_webservers_DEV]$ git status
On branch development
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   templates/index.html.j2
```

- 8. Commit your changes and check the status of your modifications.

- 8.1. Use the **git commit** command with the commit message **My first commit**:

```
[student@workstation my_webservers_DEV]$ git commit -m "My first commit"
[development 918ceb7] My first commit
 1 file changed, 1 insertion(+)
```

- 8.2. Use the **git status** command to check the status of your modifications. Git shows that there are no more modifications to be staged or committed on the local repository.

```
[student@workstation my_webservers_DEV]$ git status
On branch development
nothing to commit, working tree clean
```

- 9. Push the changes to the remote repository using the **git push** command.

```
[student@workstation my_webservers_DEV]$ git push \
> --set-upstream origin development
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
```

```
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 424 bytes | 424.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote:
remote: To create a merge request for development, visit:
remote:   http://git.lab.example.com:8081/git/my_webservers_DEV/merge_requests/
new?merge_request%5Bsource_branch%5D=development
remote:
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
 * [new branch]      development -> development
Branch 'development' set up to track remote branch 'development' from 'origin'.
```

- ▶ 10. Change back to **master** branch to verify that the changes have been pushed to the remote repository.

```
[student@workstation my_webservers_DEV]$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

- ▶ 11. Verify that **HELLO WORLD** does not appear in the template.

```
[student@workstation my_webservers_DEV]$ grep "HELLO WORLD" \
> templates/index.html.j2
```

- ▶ 12. Use the **git merge** command to merge the latest changes from the **development** branch.

```
[student@workstation my_webservers_DEV]$ git merge development
Updating 13a7afc..86e297e
Fast-forward
  templates/index.html.j2 | 1 +
  1 file changed, 1 insertion(+)
```

- ▶ 13. 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>
```

- ▶ 14. Push the new changes in the **master** branch to the remote repository.

```
[student@workstation my_webservers_DEV]$ git push
Total 0 (delta 0), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
  13a7afc..86e297e  master -> master
```

- ▶ 15. Revert the changes so the **master** branch returns to the initial status.

- 15.1. Use the **git revert** command to create a new commit, undoing the latest commit (referred as **HEAD**).

```
[student@workstation my_webservers_DEV]$ git revert --no-edit HEAD
[master dad1df1] Revert "My first commit"
  Date: Wed Apr 24 19:48:37 2019 +0000
  1 file changed, 1 deletion(-)
```

- 15.2. Review the updated file, which has returned to the previous status.

```
[student@workstation my_webservers_DEV]$ grep "HELLO WORLD" \
> templates/index.html.j2
[student@workstation my_webservers_DEV]$
```

- 15.3. Push the new commit, reverting the changes to the remote repository by using the **git push** command.

```
[student@workstation my_webservers_DEV]$ git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 432 bytes | 432.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
 d949e6d..dd89d85  master -> master
```

Finish

On **workstation**, run the **lab development-git finish** script to clean up and finish this exercise.

```
[student@workstation ~]$ lab development-git finish
```

This concludes the guided exercise.

▶ Lab

Developing with Recommended Practices

Performance Checklist

In this lab, you will update an existing Ansible project to implement better practices and commit changes to the remote project repository.

Outcomes

You should be able to:

- Implement style guidelines for task and variable names in roles and playbooks.
- Create roles with appropriate names.
- Commit changes to a Git repository.

Before You Begin

Log in as the **student** user on **workstation** and run the **lab development-review start** command. This script downloads files required for the lab.

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

1. Clone the Git repository <http://git.lab.example.com:8081/git/development-review.git> in the **/home/student/git-repos** directory.
2. Review the **site.yml** playbook as well as any other playbooks referenced in the **site.yml** playbook. Verify that the **site.yml** playbook executes without any errors.



Note

After executing the **site.yml** playbook, **servera** distributes requests to the back-end web servers:

```
[student@workstation ~]$ curl servera
This is serverb.lab.example.com. (version v1.0)
[student@workstation ~]$ curl servera
This is serverc.lab.example.com. (version v1.0)
[student@workstation ~]$ curl servera
This is serverb.lab.example.com. (version v1.0)
[student@workstation ~]$ curl servera
This is serverc.lab.example.com. (version v1.0)
```

3. The **deploy_apache.yml** playbook uses the **my_role** role to deploy a web server. This role does not have a suitable name. Rename this role to **apache**, and update all project references to the new role name. When you have updated all project references to the new role name, the **deploy_apache.yml** playbook will execute without errors.

Commit these changes to your local repository.

**Note**

Use the `git rm` command to remove the `my_role` files from the repository.

4. The output of the `deploy_webapp.yml` playbook does not contain good summaries of the playbook's actions.

Edit the `deploy_webapp.yml` playbook to add an appropriate name to the only play in the playbook.

The output of each task for the `webapp` role indicates the Ansible module for the task. Add a descriptive task name to the only task in the `in webapp` role. Execute the `deploy_webapp.yml` playbook to test your changes.

Commit these changes to your local repository.

5. The `webapp` role contains two variables, `message` and `version`.

Edit the variable names so that each variable is prefixed with `webapp_`. Update any reference to either of these variables in project files with the appropriate new variable name. When you have made all of the necessary updates, the `deploy_webapp.yml` playbook will execute without errors.

Commit these changes to your local repository.

6. Verify that the `site.yml` playbook executes with no errors, and then push all of the committed changes to the remote repository.

Evaluation

Grade your work by running the `lab development-review grade` command from your `workstation` machine. Correct any reported failures in your local Git repository, and then commit and push the changes. After changes are pushed to the remote repository, rerun the script. Repeat this process until you receive a passing score for all criteria.

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

Finish

From `workstation`, run the `lab development-review finish` command to complete this lab.

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

This concludes the lab.

► Solution

Developing with Recommended Practices

Performance Checklist

In this lab, you will update an existing Ansible project to implement better practices and commit changes to the remote project repository.

Outcomes

You should be able to:

- Implement style guidelines for task and variable names in roles and playbooks.
- Create roles with appropriate names.
- Commit changes to a Git repository.

Before You Begin

Log in as the **student** user on **workstation** and run the **lab development-review start** command. This script downloads files required for the lab.

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

1. Clone the Git repository `http://git.lab.example.com:8081/git/development-review.git` in the **/home/student/git-repos** directory.

From a terminal, create the directory **/home/student/git-repos** if it does not already exist. Then, change to this directory and clone the repository:

```
[student@workstation ~]$ mkdir -p git-repos; cd git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/development-review.git
Cloning into 'development-review'...
remote: Enumerating objects: 107, done.
remote: Counting objects: 100% (107/107), done.
remote: Compressing objects: 100% (88/88), done.
remote: Total 107 (delta 31), reused 0 (delta 0)
Receiving objects: 100% (107/107), 12.32 KiB | 3.08 MiB/s, done.
Resolving deltas: 100% (31/31), done.
[student@workstation git-repos]$ cd development-review
```

2. Review the **site.yml** playbook as well as any other playbooks referenced in the **site.yml** playbook. Verify that the **site.yml** playbook executes without any errors.

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

PLAY [Ensure HAProxy is deployed] ****
...output omitted...
```

```
PLAY [Ensure Apache is deployed] ****
...
PLAY [web_servers] ****
...
PLAY RECAP ****
servera.lab.example.com    : ok=6    ...output omitted...
serverb.lab.example.com    : ok=6    ...output omitted...
serverc.lab.example.com    : ok=6    ...output omitted...
```

**Note**

After executing the `site.yml` playbook, `servera` distributes requests to the back-end web servers:

```
[student@workstation ~]$ curl servera
This is serverb.lab.example.com. (version v1.0)
[student@workstation ~]$ curl servera
This is serverc.lab.example.com. (version v1.0)
[student@workstation ~]$ curl servera
This is serverb.lab.example.com. (version v1.0)
[student@workstation ~]$ curl servera
This is serverc.lab.example.com. (version v1.0)
```

3. The `deploy_apache.yml` playbook uses the `my_role` role to deploy a web server. This role does not have a suitable name. Rename this role to `apache`, and update all project references to the new role name. When you have updated all project references to the new role name, the `deploy_apache.yml` playbook will execute without errors.

Commit these changes to your local repository.

**Note**

Use the `git rm` command to remove the `my_role` files from the repository.

- 3.1. Rename the `roles/my_role` subdirectory to `roles/apache`.

```
[student@workstation development-review]$ cd roles
[student@workstation roles]$ ls
firewall haproxy my_role webapp
[student@workstation roles]$ mv -v my_role apache
renamed 'my_role' -> 'apache'
[student@workstation roles]$ cd ..
[student@workstation development-review]$
```

- 3.2. Edit the name of the role in the `deploy_apache.yml` playbook from `my_role` to `apache`. Save the file. The content of the `deploy_apache.yml` file displays as follows:

```

- name: Ensure Apache is deployed
  hosts: web_servers
  force_handlers: True
  gather_facts: no

  roles:
    # The "apache" role has a dependency
    # on the "firewall" role. The
    # "firewall" role requires a
    # "firewall_rules" variable be defined.
    - role: apache
      firewall_rules:

        # Allow http requests from the
        # internal zone.
        - zone: internal
          service: http

        # Add the load balancer IP to
        # the internal zone.
        - zone: internal
          source: 172.25.250.10

```

3.3. Execute the `deploy_apache.yml` playbook to test your changes.

```

[student@workstation development-review]$ ansible-playbook deploy_apache.yml
...output omitted...

PLAY RECAP ****
serverb.lab.example.com      : ok=4      ...output omitted...
serverc.lab.example.com      : ok=4      ...output omitted...

```

3.4. Commit these changes to your local repository.

```

[student@workstation development-review]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   deploy_apache.yml
    deleted:   roles/my_role/meta/main.yml
    deleted:   roles/my_role/tasks/main.yml
    deleted:   roles/my_role/vars/main.yml

Untracked files:
  (use "git add <file>..." to include in what will be committed)

```

```
roles/apache/
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Use the **git rm** command to remove the **my_role** role from the repository:

```
[student@workstation development-review]$ git rm roles/my_role/*  
rm 'roles/my_role/meta/main.yml'  
rm 'roles/my_role/tasks/main.yml'  
rm 'roles/my_role/vars/main.yml'
```

Use the **git add** command to add the files for the new **apache** role:

```
[student@workstation development-review]$ git add roles/apache
```

Stage the changes to the **deploy_apache.yml** file:

```
[student@workstation development-review]$ git add deploy_apache.yml
```

3.5. Verify that correct set of files are staged to be committed:

```
[student@workstation development-review]$ git status  
On branch master  
Your branch is up to date with 'origin/master'.  
  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
modified:   deploy_apache.yml  
renamed:    roles/my_role/meta/main.yml -> roles/apache/meta/main.yml  
renamed:    roles/my_role/tasks/main.yml -> roles/apache/tasks/main.yml  
renamed:    roles/my_role/vars/main.yml -> roles/apache/vars/main.yml
```

3.6. Commit the staged changes:

```
[student@workstation development-review]$ git commit \  
> -m "Renamed my_role role to apache."  
[master 69b6cc5] Renamed my_role role to apache.  
 4 files changed, 2 insertions(+), 2 deletions(-)  
 rename roles/{my_role => apache}/meta/main.yml (100%)  
 rename roles/{my_role => apache}/tasks/main.yml (100%)  
 rename roles/{my_role => apache}/vars/main.yml (100%)
```

3.7. Push the changes to the remote repository:

```
[student@workstation development-review]$ git push  
...output omitted...  
To http://git.lab.example.com:8081/git/development-review.git  
 8c918b1..6db348e master -> master
```

4. The output of the **deploy_webapp.yml** playbook does not contain good summaries of the playbook's actions.

Edit the **deploy_webapp.yml** playbook to add an appropriate name to the only play in the playbook.

The output of each task for the **webapp** role indicates the Ansible module for the task. Add a descriptive task name to the only task in the **webapp** role. Execute the **deploy_webapp.yml** playbook to test your changes.

Commit these changes to your local repository.

- 4.1. Edit the **deploy_webapp.yml** file, and add a **name** attribute to the play. You can use any value for the name, but it should describe the intent of the play:

```
- name: Ensure the web application is deployed
hosts: web_servers
gather_facts: no
vars:
  version: v1.0
  message: "This is {{ inventory_hostname }}."
roles:
  - role: webapp
```

- 4.2. Edit the **roles/webapp/tasks/main.yml** file, and add a **name** attribute to the task. You can use any value for the name, but the name should describe the intent of the task:

```
---
# tasks file for webapp

- name: Ensure placeholder content is deployed
copy:
  content: "{{ message }} (version {{ version }})\n"
  dest: /var/www/html/index.html
```

- 4.3. With these changes, verify that the **deploy_webapp.yml** playbook executes without errors:

```
[student@workstation development-review]$ ansible-playbook deploy_webapp.yml
...output omitted...

PLAY RECAP ****
serverb.lab.example.com      : ok=1    changed=0    unreachable=0    ...
serverc.lab.example.com      : ok=1    changed=0    unreachable=0    ...
```

- 4.4. Commit the changes:

```
[student@workstation development-review]$ git status
...output omitted...

modified:   deploy_webapp.yml
modified:   roles/webapp/tasks/main.yml

no changes added to commit (use "git add" and/or "git commit -a")
[student@workstation development-review]$ git add deploy_webapp.yml
```

```
[student@workstation development-review]$ git add roles/webapp/*
[student@workstation development-review]$ git commit \
> -m "Added names to webapp playbook and role."
[master eb9ddee9] Added names to webapp playbook and role.
 2 files changed, 4 insertions(+), 2 deletions(-)
[student@workstation development-review]$ git push
...output omitted...
To http://git.lab.example.com:8081/git/development-review.git
 6db348e..a888c73 master -> master
```

5. The **webapp** role contains two variables, **message** and **version**.

Edit the variable names so that each variable is prefixed with **webapp_**. Update any reference to either of these variables in project files with the appropriate new variable name. When you have made all of the necessary updates, the **deploy_webapp.yml** playbook will execute without errors.

Commit these changes to your local repository.

- 5.1. Edit the **roles/webapp/tasks/main.yml** file. Add the **webapp_** prefix to all role variable references. After you save your changes, the file displays as follows:

```
---
# tasks file for webapp

- name: Ensure placeholder content is deployed
  copy:
    content: "{{ webapp_message }} (version {{ webapp_version }})\n"
    dest: /var/www/html/index.html
```

- 5.2. Change the variable names in the **roles/webapp/defaults/main.yml** file. After you save your changes, the file contains:

```
webapp_version: v1.0
webapp_message: "This is {{ inventory_hostname }}."
```

- 5.3. The **deploy_webapp.yml** playbook uses the **webapp** role. Update the name of the variables in this playbook. After you save your changes, the **deploy_webapp.yml** file displays as follows:

```
- name: Ensure web application is deployed
  hosts: web_servers
  gather_facts: no
  vars:
    webapp_version: v1.0
    webapp_message: "This is {{ inventory_hostname }}."
  roles:
    - role: webapp
```

- 5.4. Ensure that the **deploy_webapp.yml** playbook executes without errors.

```
[student@workstation development-review]$ ansible-playbook deploy_webapp.yml  
...output omitted...  
  
PLAY RECAP *****  
serverb.lab.example.com      : ok=4      ...output omitted...  
serverc.lab.example.com      : ok=4      ...output omitted...
```

5.5. Commit the changes:

```
[student@workstation development-review]$ git status  
...output omitted...  
  
modified:   deploy_webapp.yml  
modified:   roles/webapp/defaults/main.yml  
modified:   roles/webapp/tasks/main.yml  
  
no changes added to commit (use "git add" and/or "git commit -a")  
[student@workstation development-review]$ git add deploy_webapp.yml  
[student@workstation development-review]$ git add roles/webapp/*  
[student@workstation development-review]$ git commit \  
> -m "Updated webapp role variable names."  
[master 3938c0d] Updated webapp role variable names.  
 3 files changed, 5 insertions(+), 5 deletions(-)  
[student@workstation development-review]$ git push  
...output omitted...  
To http://git.lab.example.com:8081/git/development-review.git  
  a888c73..8e71ee5  master -> master
```

6. Verify that the **site.yml** playbook executes with no errors, and then push all of the committed changes to the remote repository.

6.1. Execute the **site.yml** playbook.

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

If there are errors, make any necessary edits to correct the errors. Commit and push these changes back to the repository.

When you have made all the required changes and the **site.yml** playbook executes without errors, proceed to the next step.

Evaluation

Grade your work by running the **lab development-review grade** command from your **workstation** machine. Correct any reported failures in your local Git repository, and then commit and push the changes. After changes are pushed to the remote repository, rerun the script. Repeat this process until you receive a passing score for all criteria.

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

Finish

From **workstation**, run the **lab development-review finish** command to complete this lab.

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

This concludes the lab.

Summary

In this chapter, you learned:

- Recommended practices are tools and techniques to create maintainable, comprehensible, and efficient Ansible projects.
- Three points to guide effective use of Red Hat Ansible Automation are to keep things simple, stay organized, and test often.
- Infrastructure as Code means using your Ansible Playbooks as code that documents and defines your infrastructure, and ensures that it is deployed and configured correctly.
- Use a version control system like Git to track and control changes to your automation code.

Chapter 2

Managing Inventories

Goal

Manage inventories using advanced features of Ansible.

Objectives

- Write static inventory files in YAML format instead of the older INI-like format.
- Structure host and group variables using multiple files per host or group, and use special variables to override the host, port, or remote user Ansible uses for a specific host.

Sections

- Writing YAML Inventory Files (and Guided Exercise)
- Managing Inventory Variables (and Guided Exercise)

Lab

Managing Inventories

Writing YAML Inventory Files

Objectives

After completing this section, you should be able to write static inventory files in YAML format instead of the older INI-like format.

Inventory Plugins

Starting with Ansible 2.4, the different formats of inventory files that Ansible supports are implemented as plugins. Plugins are pieces of code that enhance Ansible's functionality. By implementing inventory support through plugins, Ansible can support new formats and methods of generating inventory data simply by providing a new plugin. Traditional INI-style static inventory files and dynamic inventory scripts are each implemented with a plugin.

Most inventory plugins are disabled by default. You can enable specific plugins in your `ansible.cfg` configuration file, in the `enable_plugins` directive in the `inventory` section:

```
[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml
```

The list in the preceding example is the default if you do not specify an `enable_plugins` directive. When Ansible parses inventory sources, it will try to use each plugin in the order in which it appears in the `enable_plugins` directive.

The `script` plugin provides support for standard dynamic inventory scripts. The `ini` plugin provides support for standard INI-format static inventory files. The other plugins get inventory information from files in other formats or other sources. Details are available at <https://docs.ansible.com/ansible/latest/plugins/inventory.html>.

Some inventory plugins that are shipped with Ansible provide standardized replacements for dynamic inventory scripts. For example, `openstack` can get information about VMs in a Red Hat OpenStack Platform environment, or `aws_ec2` can get information about cloud instances from Amazon Web Services EC2.



Important

When should you use dynamic inventory scripts, and when should you use inventory plugins?

One advantage of using an included inventory plugin, if one is available for your use case, is that it is maintained along with the core Ansible code. You can write your own inventory plugin, but it must be implemented in Python like most plugins. Details are available at https://docs.ansible.com/ansible/latest/dev_guide/developing_inventory.html.

You can also continue to use your own dynamic inventory scripts, which the plugin system will continue to support. These scripts only need to be executable files, so you can write them in whichever programming language you prefer.

YAML Static Inventory Files

You can use the **yaml** inventory plugin to write static inventory files in a YAML-based syntax instead of the INI-based syntax. This plugin is enabled by default. The YAML inventory plugin was created because it is easy for users to read, easy for software to parse, and allows you to use YAML for playbooks, variable files, and inventories.

To review, here is an example of a INI static inventory file:

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

[web_servers]
serverb.lab.example.com
serverc.lab.example.com

[backend_server_pool]
server[b:f].lab.example.com
```

This static inventory has three groups:

- The **lb_servers** group includes the host **servera.lab.example.com**.
- The **web_servers** group includes the hosts **serverb.lab.example.com** and **serverc.lab.example.com**.
- The **backend_server_pool** group uses an alphabetical range to include five hosts, **serverb**, **serverc**, **serverd**, **servere**, and **serverf** in **lab.example.com**.

And here is the same inventory in YAML format:

```
lb_servers:
  hosts:
    servera.lab.example.com:
web_servers:
  hosts:
    serverb.lab.example.com:
    serverc.lab.example.com:
backend_server_pool:
  hosts:
    server[b:f].lab.example.com:
```

YAML inventories use blocks to organize related configuration items. Each block begins with the name of a group followed by a colon (:). Everything indented below the group name belongs to that group.

If indented below a group name, then the keyword **hosts** starts a block of host names. All server names indented below **hosts** belong to this group. These servers themselves form their own groups, so they must end in a colon (:).

You can also use the keyword **children** in a group block. This keyword starts a list of groups that are members of this group. Those member groups can have their own **hosts** and **children** blocks.

One advantage that the YAML syntax has over the INI syntax is that it organizes both the lists of servers and the list of nested groups in the same place in the static inventory file.

 **Important**

The **all** group implicitly exists at the top level and includes the rest of the inventory as its children. You can explicitly list it in your YAML inventory file, but it is not required:

```
all:  
  children:  
    lb_servers:  
      hosts:  
        servera.lab.example.com:  
    web_servers:  
      hosts:  
        serverb.lab.example.com:
```

Some INI-based static inventories include hosts that are not a member of any group.

```
notinagroup.lab.example.com  
  
[mailserver]  
mail.lab.example.com
```

Ansible automatically puts any of these hosts in the special group **ungrouped**. You can accomplish the same thing in a YAML-based static inventory by explicitly assigning hosts to **ungrouped**:

```
all:  
  children:  
    ungrouped:  
      notinagroup.lab.example.com:  
    mailserver:  
      mail.lab.example.com:
```

Setting Inventory Variables

You can set inventory variables directly in a YAML-based inventory file, just like you can in an INI-based inventory file.

**Note**

In many cases, it is the best practice to avoid storing variables in the static inventory file itself.

Many experienced Ansible developers prefer to use the static inventory file to simply store information about which hosts are managed and to what groups they belong. The variables and their values are stored in the **host_vars** or **group_vars** files for the inventory.

However, there might be circumstances where you want to keep variables like **ansible_port** or **ansible_connection** in the same file as the inventory itself, thus keeping this information in one place. If you set variables in too many places, it can be harder to remember where a particular variable is being set.

In a group block, you can use the **vars** keyword to set group variables directly in a YAML inventory file. For example, in an INI-based static inventory file, you could set the **smtp_relay** variable to the value **smtp.lab.example.com** for all hosts in group **monitoring** as follows:

```
[monitoring]
watcher.lab.example.com

[monitoring:vars]
smtp_relay: smtp.lab.example.com
```

In a YAML-based static inventory file, you would set it like this:

```
monitoring:
  hosts:
    watcher.lab.example.com:
      vars:
        smtp_relay: smtp.lab.example.com
```

You can set host variables as items indented under the host in a YAML inventory file. For example, in an INI-based static inventory file, you could set the **ansible_connection** variable to **local** for the host **localhost** as follows:

```
[workstations]
workstation.lab.example.com
localhost ansible_connection=local
host.lab.example.com
```

In a YAML-based static inventory file, you would set it like this:

```
workstations:
  hosts:
    workstation.lab.example.com:
      localhost:
        ansible_connection: local
    host.lab.example.com:
```

Converting from INI to YAML

You can use the **ansible-inventory** command to help convert an INI-based inventory into YAML format. However, this is not the tool's intended purpose. This tool is meant to display the entire configured inventory as Ansible sees it, and results can differ from those reported by the original inventory file. The **ansible-inventory** command parses and tests the format of the inventory files, but it does not attempt to validate whether a hostname in the inventory actually exists.



Note

Remember that a name in an inventory file is not necessarily a valid hostname, but is used by the playbook to refer to a specific managed host. That managed host's name could have an **ansible_host** host variable set that points to the real name or IP address that Ansible should use when connecting to the managed host, and the name in the inventory file can be a simplified alias for the purposes of the playbook.

For example, assume that you have an initial INI-formatted static inventory, named **origin_inventory**, as follows:

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

[web_servers]
serverb.lab.example.com
serverc.lab.example.com

[web_servers:vars]
alternate_server=serverd.lab.example.com

[backend_server_pool]
server[b:f].lab.example.com
```

You could use the following **ansible-inventory** to produce YAML output. In this example, the command has been wrapped across two lines. If typing the command as on a single line, ignore the \ at the end of the first line and the > prompt at the start of the second line.

```
[user@demo ~]$ ansible-inventory --yaml -i origin_inventory --list \
> --output destination_inventory.yml
all:
  children:
    backend_server_pool:
      hosts:
        serverb.lab.example.com:
          alternate_server: serverd.lab.example.com
        serverc.lab.example.com:
          alternate_server: serverd.lab.example.com
        serverd.lab.example.com: {}
        servere.lab.example.com: {}
        serverf.lab.example.com: {}
    lb_servers:
      hosts:
```

```

servera.lab.example.com: {}
ungrouped: {}
web_servers:
  hosts:
    serverb.lab.example.com: {}
    serverc.lab.example.com: {}

```

If you look closely at this output, you may notice something strange. The **alternate_server** group variable for **web_servers** is being reported as host variables for the two members of that group. In addition, since the group **backend_server_pool** is the first appearance of those two hosts, the value for that variable is being reported there instead of with the definition of those hosts in group **web_servers**.

The converted inventories will result in the same output. Remember that even though the group variable is being set based on membership in **web_servers**, it remains set for hosts in that group even when a play selects that host based on membership in a different group.



Important

If a host has one of its variables set based on membership in a group, and if that variable is not overridden, then it will be valid even if the host is selected for a play based on membership in a different group.

The problem with this behavior is that this is not how the original inventory worked, which may cause confusion. To fix this discrepancy, you must edit the resulting YAML. In this case, you will edit the **destination_inventory.yml** to copy the **alternate_server** variables from one of the servers, b or c, delete both definitions, and then create a **vars** section under the **web_servers**. In this new section, copy the definition of the variable **alternate_server** from the servers. You must also change the servers that were originally defined by a range in **backend_server_pool** to use a range again instead of being listed individually.

The result should look like this:

```

all:
  children:
    backend_server_pool:
      hosts:
        server[b:f].lab.example.com:
    lb_servers:
      hosts:
        servera.lab.example.com: {}
  ungrouped: {}
  web_servers:
    hosts:
      serverb.lab.example.com: {}
      serverc.lab.example.com: {}
  vars:
    alternate_server: serverd.lab.example.com

```

When converting very big inventory files, using the **ansible-inventory** command can save a lot of time, but you must use it with caution. It works better if your original static inventory file does not directly declare inventory variables, but gets them from external files in **host_vars** and **group_vars**.

**Note**

Some group or host lines in the YAML output of **ansible-inventory** end with `{}`. That indicates that the group does not have any members or group variables, or that the host has no host variables. These braces do not need to be included if you are writing in YAML format by hand.

YAML Troubleshooting Tips

If you have trouble with static inventory files in YAML format, there are a few things that you should keep in mind about YAML syntax.

Protect a Colon Followed by a Space

Remember that in an unquoted string, a colon followed by a space will cause an error. YAML will interpret this as starting a new element in a dictionary.

Here are several examples of this and how to address them:

```
title: Ansible: Best Practices # the second colon produces an error
fine: Not:a:problem # No space after the colon means no special treatment
simple: 'Quoting the value with the : character solves the problem'
double: "Double quotes also work with the : but permit escaped characters \n"
```

Protect a Variable that Starts a Value

Ansible does variable replacement with `{{ variable }}`, but anything beginning with `{` in YAML is interpreted as the start of a dictionary. Thus, you must enclose the variable placeholder with double quotes: `foo: "{{ variable }}" rest of the value`.

In general, when using any of the reserved characters `[] {} > | * & ! % # ^ @ ,`, you should use double quotes `" "` around the value.

Know the Difference Between a String and a Boolean or Float

Booleans and floating point numbers used as a value for a variable should not be quoted. Values that are quoted are treated as a string.

For example, a Boolean and a string:

```
active: yes # Boolean value
default_answer: "yes" # string containing yes
```

A floating point value and a string:

```
temperature: 36.5 # Floating point value
version: "2.0" # String containing a dot
```

**References****Ansible YAML Syntax**

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

► Guided Exercise

Writing YAML Inventory Files

In this exercise, you will convert a Red Hat Ansible inventory file from the INI format to the YAML format.

Outcomes

You should be able to replace an Ansible inventory file in the INI format with its YAML equivalent.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

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

- 1. An inventory file in INI format has been provided for you in the **/home/student/D0447/labs/inventory-yaml** directory. Change to that directory, copy the file, and use a text editor to manually convert it to YAML format.

- 1.1. Change to the **/home/student/D0447/labs/inventory-yaml** directory.

```
[student@workstation ~]$ cd /home/student/D0447/labs/inventory-yaml
```

- 1.2. Copy the existing static inventory file to **inventory.yml**.

```
[student@workstation inventory-yaml]$ cp inventory inventory.yml
```

- 2. Modify the new inventory file **inventory.yml** to adapt to the YAML syntax

- 2.1. Convert the groups to YAML format.

- The name of each group should not be surrounded by square brackets ([]), but should end with a colon (:).
- Beneath the group name, add a line indented by two spaces relative to the group name, **hosts:**, to start the block of hosts that are members of the group.
- Next, the hosts in the group should each have their own line, indented by four spaces relative to the group name, and each hostname should end in a colon.

The exact number of spaces does not matter as long as indentation is consistent and the hostnames are indented more than **hosts:**, which is indented more than the group name.

For example, the following group:

```
[active_web_servers]
server[b:c].lab.example.com
```

Would become:

```
active_web_servers:
  hosts:
    server[b:c].lab.example.com:
```

Repeat the operation for the groups **all_servers**, **inactive_web_servers**, and **region_eu**.

- 2.2. Convert the definition of the child groups of **web_servers**. The child groups are listed in the original file under the **[web_servers:children]** section.
 - Beneath the group name, add a line indented by two spaces relative to the group name, **children:**, to start the block of groups that are members of this group.
 - Next, the child groups that are part of this group should each have their own line, indented by four spaces relative to the group name, and each group name should end in a colon.

For example, the following section:

```
[web_servers:children]
active_web_servers
inactive_web_servers
```

Would become:

```
web_servers:
  children:
    active_web_servers:
    inactive_web_servers:
```

- 2.3. Finally, convert the **all_servers:children** group to YAML by putting the child groups in the **children** section.

```
[all_servers:children]
web_servers
```

Becomes:

```
all_servers:
  hosts:
    servera.lab.example.com:
  children:
    web_servers:
```

- 2.4. The **inventory.yml** file should look like this after the conversion:

```
active_web_servers:  
  hosts:  
    server[b:c].lab.example.com:  
  
inactive_web_servers:  
  hosts:  
    server[d:f].lab.example.com:  
  
region_eu:  
  hosts:  
    serverc.lab.example.com:  
    serverf.lab.example.com:  
  
web_servers:  
  children:  
    active_web_servers:  
    inactive_web_servers:  
  
all_servers:  
  hosts:  
    servera.lab.example.com:  
  children:  
    web_servers:
```

- 3. To test the inventory, use the **ansible** command to ping all the servers:

```
[student@workstation inventory-yaml]$ ansible -i inventory.yml all_servers -m ping  
...output omitted...  
serverf.lab.example.com | SUCCESS => {  
  "ansible_facts": {  
    "discovered_interpreter_python": "/usr/libexec/platform-python"  
  },  
  "changed": false,  
  "ping": "pong"  
}
```

- 4. Return to the user's home.

```
[student@workstation inventory-yaml]$ cd
```

Finish

On **workstation**, run the **lab inventory-yaml finish** script to complete this lab.

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

This concludes the exercise.

Managing Inventory Variables

Objectives

After completing this section, you should be able to:

- Structure host and group variables using multiple files per host or group
- Override the host, port, or remote user Ansible uses for a specific host with special variables.

Describing Variables Basic Principles

Variables allow you to write your tasks, roles, and playbooks to be reusable and flexible. They allow you to specify differences in configuration between different systems. Set variables in many places, including:

- In a role's **defaults** and **vars** directory.
- In the inventory file, either as a host variable or a group variable.
- In a variable file under the **group_vars** or **host_vars** subdirectories of the playbook or inventory.
- In a play, role, or task.

As you define and manage variables in a project, plan to follow these principles:

Keep It Simple

Even though Ansible variables can be defined in many ways, try to define variables using only a couple of different methods and in only a few places.

Don't Repeat Yourself

If a set of systems have a common configuration, organize them into a group and set inventory variables for them in a file in the **group_vars** directory. This way, you do not have to define the same settings on a host-by-host basis, and when you have to modify a variable for that group of systems you can do it by updating the variable file.

Organize Variables in Small, Readable Files

If you have a large project with many host groups and variables, split the variable definitions into multiple files. To make it easier to find particular variables, group related variables into a file and give the file a meaningful name.

Remember that you can use subdirectories instead of files in **host_vars** and **group_vars**. So, for example, you could have a directory **group_vars/webserver** for group **webserver**, and that directory could contain a file named **firewall.yml** which contains only variables related to firewall configuration. That directory could also contain other group variable files related to other parts of the servers' configuration.

Introducing Variable Merging and Precedence

When you define the same variable in multiple ways, Ansible uses precedence rules to pick a value for the variable. After processing all the variable definitions, Ansible generates a set of merged variables for each host at the start of each task.

When Ansible merges variables, if there are two definitions for the same variable in different places, it will use the value from the place that has the highest precedence.

A list of the precedence order is available at https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html in the subsection "Variable precedence: Where should I put a variable?". The following discussion will break this down in a bit more detail, from lowest to the highest precedence.

Command-line Options

Options that can be passed to **ansible-playbook** on the command line, other than **-e** for extra variables, have the lowest precedence. For example, you can set the remote user with the **-u** option to override the configuration file, but it can be overridden by setting **ansible_user** at a higher precedence.

Role Defaults

Default values set by a role in files in **rolename/defaults/** have a very low priority so that they are easy to override. These are used so that the role's variable is defined with some reasonable value but can be configured to be something else, so role defaults are often used for user configuration of a role.

Host and Group Variables

You can set host-specific and group-specific variables in a number of places. You can set them relative to the location of your inventory. You can set them relative to the location of your playbook. Finally, you can set them by collecting host facts (or by reading facts from a cache).

The exact precedence list for these variables is, from lowest to highest:

- Group variables set directly in an inventory file or by a dynamic inventory script.
- Group variables for **all** set in the inventory's **group_vars/all** file or subdirectory.
- Group variables for **all** set in the playbook's **group_vars/all** file or subdirectory.
- Group variables for other groups set in the inventory's **group_vars** subdirectory.
- Group variables for other groups set in the playbook's **group_vars** subdirectory.
- Host variables set directly in an inventory file or by a dynamic inventory script.
- Host variables set in the inventory's **host_vars** subdirectory.
- Host variables set in the playbook's **host_vars** subdirectory.
- Host facts and cached facts.

The biggest source of confusion here is the distinction between the **group_vars** and **host_vars** subdirectories relative to the inventory and the ones relative to the playbook. One reason for this is that if you are using a static inventory file in the same directory as the playbook you are using, there is no difference.

If you have **group_vars** and **host_vars** subdirectories in the same directory as your playbook, those group and host variables will be automatically included.

If you are using a flat inventory file in some directory other than the one the playbook is in, then the **group_vars** and **host_vars** directories in the inventory file's directory will also be automatically included. The variables included from the playbook's directory will, however, override

them if there is a conflict. For example, if you are using `/etc/ansible/hosts` as your inventory, then `/etc/ansible/group_vars` will be used as another group variable directory.

If you are using an inventory directory that contains multiple inventory files, then the `group_vars` and `host_vars` subdirectories of your inventory directory will be included.

As an example, consider the following tree structure:

```
.
├── ansible.cfg
├── group_vars/
│   └── all
└── inventory/
    ├── group_vars/
    │   └── all
    ├── phoenix-dc
    └── singapore-dc
└── playbook.yml
```

The `playbook.yml` file is the playbook. The `ansible.cfg` file configures the `inventory` directory as the inventory, and `phoenix-dc` and `singapore-dc` are two INI-based static inventory files.

The file `inventory/group_vars/all` loads group variables for group `all` based on the configuration of the inventory.

The file `group_vars/all` loads group variables for group `all` based on the location of the playbook. In the case of a conflict, these settings override `inventory/group_vars/all`.



Note

The value of this distinction is that you can set default values for variables that are bundled up with the inventory files in some location that is shared by all your playbooks. But you can still override the settings for those inventory variables in individual playbook directories.

While you can set host or group variables directly in inventory files (like `phoenix-dc` in the preceding example), it is generally not a good practice. It is a lot easier to find all the variable settings that apply to a host or group if they are grouped in a file or files that only contain settings for that host or group. If you have to also examine the inventory file or files, that can be a lot more time-consuming and error prone, especially for a large inventory.

Play Variables

The next category of variables is those set in the playbook as part of a play, task, role parameter, or which are included or imported. These have higher precedence than host or group variables, role defaults, and command-line options other than `-e`.

The precedence list for these variables is, from lowest to highest:

- Set by the `vars` section of a play.
- Set by prompting the user with a `vars_prompt` section in a play.
- Set from a list of external files by the `vars_files` section of a play.

- Set by a file in a role's **rolename/vars/** subdirectory.
- Set for the current **block** with a **vars** section of that block.
- Set for the current task with a **vars** section of that task.
- Loaded dynamically with the **include_vars** module.
- Set for a specific host by using either the **set_fact** module or by using **register** to record the result of task execution on a host.
- Parameters set for a role in the playbook when loaded by the **role** section of a play or by using the **include_role** module.
- Set by a **vars** section on tasks included with the **include_tasks** module.

It is important that you notice that the normal **vars** section on a play has the lowest precedence in this category. There are many ways to override those settings if necessary. Variables set here do override host-specific and group-specific settings in general.

Using **vars_prompt** is probably not the best practice. It requires interaction while running **ansible-playbook** and is not compatible with Red Hat Ansible Tower.

The **vars_files** directive can be very useful for organizing large lists of variables that are not host or group specific into separate files by function. It can also help you separate sensitive variables into a separate file encrypted by Ansible Vault from variables that are not sensitive and do not need to be encrypted.

You can set variables that apply only to a specific block or task. These values override the play variables and inventory variables. You should probably use these rarely, since they can make the playbook more complex.

```
- name: Task with a local variable definition
  vars:
    taskvariable: task
  debug:
    var: taskvariable
```

Notice that variables loaded by **include_vars** have a high precedence, and can override variables set for roles and specific blocks and tasks. In many cases, you might want to use **vars_files** instead if you do not want to override those values with your external variable files.

The **set_fact** module and the **register** directive both set host-specific information, either a fact or the results of task execution on that host. Note that **set_fact** sets a very high precedence value for that variable for the rest of the playbook run, but if you cache that fact it will be stored in the fact cache at normal fact precedence (lower than play variables).

Extra Variables

Extra variables set by using the **-e** option of the **ansible-playbook** command always have the highest precedence. This is useful so that you can override the global setting for a variable in your playbook from the command line without editing any of the Ansible project's files.

Separating Variables from Inventory

The inventory sources define the hosts and host groups used by Ansible, whether they are static files or a dynamic inventory script. If you are managing your inventory as static files, you can define variables in the inventory file or files, in the same files in which you define your host and host group

lists. However, this is not the best practice. As your environment grows in both size and variety, the inventory file becomes large and difficult to read.

In addition, you will probably want to migrate to dynamic inventory sources rather than static inventory files to make it easier to manage your inventory. But you might still want to have the ability to manage inventory variables statically, separately from or in addition to the output from the dynamic inventory script.

A better approach is to move variable definitions from the inventory file into separate variable files, one for each host group. Each variable file is named after a host group, and contains variable definitions for the host group:

```
[user@demo project]$ tree -F group_vars
group_vars/
├── db_servers.yml
├── lb_servers.yml
└── web_servers.yml
```

This structure makes it easier to locate configuration variables for any of the host groups: **db_servers**, **lb_servers**, or **web_servers**. As long as each host group contains a few variable definitions, the above organizational structure is sufficient. But as playbook complexity increases, even these files can become long and difficult to understand.

An even better approach for large, diverse environments is to create subdirectories for each host group under the **group_vars** directory. Ansible parses any YAML in these subdirectories, and associates the variables with a host group based on the parent directory:

```
[user@demo project2]$ tree -F group_vars
group_vars/
├── db_servers/
│   ├── 3.yml
│   ├── a.yml
│   └── myvars.yml
├── lb_servers/
│   ├── 2.yml
│   ├── b.yml
│   └── something.yml
└── web_servers/
    └── nothing.yml
```

In preceding example, variables in the **myvars.yml** file are associated with the **db_servers** host group, because the file is contained in the **group_vars/db_servers** subdirectory. The file names in this example however, make it difficult to know where to find a particular variable.

If you choose to use this organizational structure for variables, group variables with a common theme into the same file and use a file name to indicate that common theme. When a playbook uses roles, a common convention is to create variable files named after each role in playbook.

A project organized according to this convention might look like this:

```
[user@demo project3]$ tree -F group_vars
group_vars/
├── all/
│   └── common.yml
```

```

└── db_servers/
    ├── mysql.yml
    └── firewall.yml
└── lb_servers/
    ├── firewall.yml
    ├── haproxy.yml
    └── ssl.yml
└── web_servers/
    ├── firewall.yml
    ├── webapp.yml
    └── apache.yml

```

This organizational structure for a project allows you to quickly see the types of variables that are defined for each host group.

All variables present in files in directories under the **group_vars** directory are merged together with the rest of the variables. Separating variables into files grouped by functionality makes the whole playbook project easier to understand and maintain.

Special Inventory Variables

There are a number of variables that you can use to change how Ansible will connect to a host listed in the inventory. Some of these are most useful as host-specific variables, but others might be relevant to all hosts in a group or in the inventory.

ansible_connection

The connection plugin used to access the managed host. By default, **ssh** is used for all hosts except **localhost**, which uses **local**.

ansible_host

The actual IP address or fully-qualified domain name to use when connecting to the managed host, instead of using the name from the inventory file (**inventory_hostname**). By default, this variable has the same value as the inventory host name.

ansible_port

The port Ansible uses to connect to the managed host. For the (default) SSH connection plugin, the value defaults to **22**.

ansible_user

Ansible connects to the managed host as this user. Ansible's default behavior is to connect to the managed host using the same user name as the user running the Ansible Playbook on the control node.

ansible_become_user

Once Ansible has connected to the managed host, it will switch to this user using **ansible_become_method** (which is **sudo** by default). You might need to provide authentication credentials in some way.

ansible_python_interpreter

The path to the Python executable that Ansible should use on the managed host. On Ansible 2.8 and later, this defaults to **auto**, which will automatically select a Python interpreter on the host running the playbook depending on what operating system it is running, so it is less necessary to use this setting compared to older versions of Ansible.

Human Readable Inventory Host Names

When Ansible executes a task on the remote host, the output contains the inventory host name. Because you can specify alternate connection properties using the special inventory variables above, inventory host names can be assigned arbitrary names. When you assign inventory hosts with meaningful names, you are better able to understand playbook output and diagnose playbook errors.

Consider a playbook that uses the following YAML inventory file:

```
web_servers:
  hosts:
    server100.example.com:
    server101.example.com:
    server102.example.com:
lb_servers:
  hosts:
    server103.example.com:
```

With the above inventory file, any Ansible output references these names. Consider the following hypothetical output from a playbook that uses this inventory file:

```
[user@demo project]$ ansible-playbook site.yml
...output omitted...
PLAY RECAP ****
server100.example.com : ok=4    changed=0    unreachable=0    failed=0    ...
server101.example.com : ok=4    changed=0    unreachable=0    failed=0    ...
server102.example.com : ok=4    changed=0    unreachable=0    failed=0    ...
server103.example.com : ok=4    changed=0    unreachable=0    failed=1    ...
```

In the above output, you can not easily tell that the failed host is a load balancer.

Alternatively, use an inventory file that contains descriptive names and defines the necessary special inventory variables:

```
web_servers:
  hosts:
    webserver_1:
      ansible_host: server100.example.com
    webserver_2:
      ansible_host: server101.example.com
    webserver_3:
      ansible_host: server102.example.com
lb_servers:
  hosts:
    loadbalancer:
      ansible_host: server103.example.com
```

Now the output from the playbook provides a descriptive name in the **PLAY RECAP**:

```
[user@demo project]$ ansible-playbook site.yml
...output omitted...
PLAY RECAP ****
loadbalancer      : ok=4    changed=0    unreachable=0    failed=1    ...
webserver_1       : ok=4    changed=0    unreachable=0    failed=0    ...
webserver_2       : ok=4    changed=0    unreachable=0    failed=0    ...
webserver_3       : ok=4    changed=0    unreachable=0    failed=0    ...
```

There are a number of situations where you might want to use an arbitrary host name in your playbook that is mapped to a real IP address or host name with **ansible_host**. Some of them include:

- You might want Ansible to connect to that host using a specific IP address that is different from the one that resolves in DNS. For example, there might be a particular management address that is not public, or the machine might have multiple addresses in DNS but one is on the same network as the control node.
- You might be provisioning cloud systems that have arbitrary names, but you want to refer to those systems in your playbook with names that make sense based on the roles they play. If you are using a dynamic inventory, it might be that your dynamic inventory source assigns host variables automatically based on each system's intended role.
- You might be referring to the machine by a short name in the playbook, but you need to refer to it by a long fully-qualified domain name in the inventory to properly connect to it.

Identifying the Current Host Using Variables

When a play is running, there are a number of variables and facts that can be used to identify the name of the current managed host executing a task:

inventory_hostname

The name of the managed host currently being processed, taken from the inventory.

ansible_host

The actual IP address or host name that was used to connect to the managed host, as previously discussed.

ansible_facts['hostname']

The short (unqualified) host name collected as a fact from the managed host.

ansible_facts['fqdn']

The fully-qualified domain name (FQDN) collected as a fact from the managed host.

One final variable that can be useful is **ansible_play_hosts**, which is a list of all the hosts that have not yet failed during the current play (and therefore will be used for the tasks remaining in the play).



References

Variable Precedence - Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

Working with Inventory - Ansible Documentation

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

Special Variables - Ansible Documentation

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

► Guided Exercise

Managing Inventory Variables

In this exercise, you will set up directories containing multiple host variable files for some of your managed hosts, and override the name used in the inventory file with a different name or IP address for at least one of your hosts.

Outcomes

You should be able to split the location of host variable files in multiple directories to improve maintainability.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

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

- ▶ 1. Clone the Git repository `http://git.lab.example.com:8081/git/inventory-variables.git` in the **/home/student/git-repos** directory, and change to the cloned project directory.
 - 1.1. From a terminal, create the directory **/home/student/git-repos** if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd /home/student/git-repos
```

- 1.3. Clone the Git repository containing the Ansible project, which also includes the inventory files to update.

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/inventory-variables.git
```

- ▶ 2. Create the directory to hold the variables for the host groups **lb_servers** and **web_servers**.
 - 2.1. Create the **group_vars** directory.

```
[student@workstation git-repos]$ cd inventory-variables
[student@workstation inventory-variables]$ mkdir group_vars
```

- 2.2. Create the **lb_servers** directory to hold the variables for the hosts in the **lb_servers** group.

```
[student@workstation inventory-variables]$ mkdir group_vars/lb_servers
```

- 2.3. Create the **web_servers** directory to hold the variables for the **web_servers** group.

```
[student@workstation inventory-variables]$ mkdir group_vars/web_servers
```

- 3. Review the **deploy_haproxy.yml** playbook. Edit the file as follows:

- Move the configuration of the **firewall_rules** variable to a new group variable file, **group_vars/lb_servers/firewall.yml**.
 - Move the configuration of the **haproxy_appservers** variable to a new group variable file, **group_vars/lb_servers/deploy_haproxy.yml**.
- 3.1. Use a text editor to create a group variable file named **group_vars/lb_servers/firewall.yml**. Edit the file to contain the definition of the **firewall_rules** variable from the **deploy_haproxy.yml** playbook.

```
[student@workstation inventory-variables]$ vi group_vars/lb_servers/firewall.yml
```

The contents of the file should be:

```
firewall_rules:
# Allow 80/tcp connections
- port: 80/tcp
```

- 3.2. Use a text editor to create a group variable file named **group_vars/lb_server/haproxy.yml**. Edit the file to contain the definition of the **haproxy_appservers** variable from the **deploy_haproxy.yml** playbook.

```
[student@workstation inventory-variables]$ vi group_vars/lb_servers/haproxy.yml
```

The contents of the file should be:

```
haproxy_appservers:
- name: serverb.lab.example.com
  ip: 172.25.250.11
  backend_port: 80
- name: serverc.lab.example.com
  ip: 172.25.250.12
  backend_port: 80
```

- 3.3. Edit the **deploy_haproxy.yml** playbook to remove both variable definitions from the file.

```
[student@workstation inventory-variables]$ vi deploy_haproxy.yml
```

The final state of the **deploy_haproxy.yml** playbook should be:

```

- name: Ensure HAProxy is deployed
  hosts: lb_servers
  force_handlers: True

  roles:
    # The "haproxy" role has a dependency on the "firewall" role.
    # The "firewall" role requires a "firewall_rules" variable be defined.
    - role: haproxy

```

- ▶ 4. Review the `deploy_apache.yml` playbook. Edit the file to move the definition of the `firewall_rules` variable to the new group variable file, `group_vars/web_servers/firewall.yml`.
- 4.1. Use a text editor to create a group variable file named `group_vars/web_servers/firewall.yml`. Edit the file to contain the definition of the `firewall_rules` variable from the `deploy_apache.yml` playbook.

```
[student@workstation inventory-variables]$ vi group_vars/web_servers/firewall.yml
```

The contents of the file should be:

```

firewall_rules:
# Allow http requests from any internal zone source.
- zone: internal
  service: http
  source: "172.25.250.10"

```

- 4.2. Edit the `deploy_apache.yml` playbook to remove the `firewall_rules` variable under the `apache` role from the file.

```
[student@workstation inventory-variables]$ vi deploy_apache.yml
```

The final state of the file should be:

```

- name: Ensure Apache is deployed
  hosts: web_servers
  force_handlers: True

  roles:
    # The "apache" role has a dependency on the "firewall" role.
    # The "firewall" role requires a "firewall_rules" variable be defined.
    - role: apache

```

- ▶ 5. The playbook deploys the inventory host `load_balancer` as the load balancer and the hosts in group `web_servers` as the backend web servers.
- Edit the `inventory.yml` static inventory file so that Ansible connects to `servera.lab.example.com` when the host `load_balancer` is referred to in a playbook. The inventory hosts `serverb.lab.example.com` and `serverc.lab.example.com` should be in group `web_servers`.

- 5.1. Edit the `inventory.yml` file with a text editor.

Use the **ansible_host** variable to cause Ansible to connect to the **load_balancer** host using **servera.lab.example.com**.

```
[student@workstation inventory-variables]$ vi inventory.yml
```

The contents of the file should be:

```
lb_servers:  
  hosts:  
    load_balancer:  
      ansible_host: servera.lab.example.com  
  
web_servers:  
  hosts:  
    server[b:c].lab.example.com:
```

► 6. Use **ansible-playbook** to run the **site.yml** master playbook.

```
[student@workstation inventory-variables]$ ansible-playbook site.yml  
...output omitted...  
TASK [haproxy : Ensure haproxy configuration is set] ****  
changed: [load_balancer]  
...output omitted...  
TASK [apache : Install http] ****  
ok: [serverc.lab.example.com]  
ok: [serverb.lab.example.com]  
...output omitted...
```

Notice that Ansible uses the **load_balancer** inventory name in the output, instead of **servera.lab.example.com**. The playbook is not using **load_balancer** as a group name and **servera.lab.example.com** as a group member, but **load_balancer** is a managed host name in the inventory that is connected to using the DNS host name **servera.lab.example.com**.

► 7. Commit and push the changes to the Git repository.

```
[student@workstation inventory-variables]$ git add .  
[student@workstation inventory-variables]$ git commit -m "Use group_vars"  
[student@workstation inventory-variables]$ git push
```

Finish

On **workstation**, run the **lab inventory-variables finish** script to complete this lab.

```
[student@workstation inventory-variables]$ lab inventory-variables finish
```

This concludes the guided exercise.

▶ Lab

Managing Inventories

Performance Checklist

In this lab, you will convert an INI-style inventory file to YAML format and configure group variable files for two new host groups.

Outcomes

You should be able to:

- Organize playbook variables in a directory structure.
- Write an inventory file in YAML format.
- Assign arbitrary host names for remote hosts in an inventory.

Before You Begin

Log in as the **student** user on **workstation** and run **lab inventory-review start**. This script initializes the remote Git repository that you need for this lab.

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

The Git repository contains a **site.yml** playbook that configures a front end load balancer and a pool of backend web servers. The back server pool is partitioned into two groups, **A** and **B**. Each pool partition deploys an independent version of the web application.

1. Clone the Git repository <http://git.lab.example.com:8081/git/inventory-review.git> in the **/home/student/git-repos** directory. Review the project playbooks and the **inventory** file and then execute the **site.yml** playbook. Make several web requests to **servera** to verify that the same version of the web application is deployed to all back end web servers.
2. Create a variable file for the **a_web_servers** host group to set the **webapp_version** variable to the value **v1.1a**. Ensure the variable file is saved in the appropriate directory with the name **webapp.yml**.
Create a similar variable file for the **b_web_servers** host group, but set the value of the **webapp_version** variable to **v1.1b**.
3. Execute the **deploy_webapp.yml** playbook. Use **curl** or a web browser to confirm that requests for web pages sent to **servera** produce two different versions of the web application.
Commit these changes to your local repository.
4. Create a new YAML-formatted inventory file named **inventory.yml**. Ensure that this file contains the same inventory data as the INI-formatted **inventory** file. To test the YAML inventory file you created, execute the **site.yml** file using this inventory file.
When complete, the project contains both an **inventory** and **inventory.yml** file.

5. Use **ansible_host** variables to configure the real host names that are used for connections when Ansible manages these hosts. Modify the **inventory.yml** file so that the managed hosts are referred to in the inventory using the following naming convention:
- Managed hosts in the **lb_servers** host group have an inventory name matching **loadbalancer_N**, where N is assigned sequentially to servers starting at 1.
 - The **web_servers** host group has two child groups, **a_web_servers** and **b_web_servers**.
 - Hosts in the **a_web_servers** host group follow a similar numbered naming scheme of the form: **backend_aN**.
 - Hosts in the **b_web_servers** host group also follow a similar numbered naming scheme of the form **backend_bN**.
6. Verify that the **site.yml** playbook executes without errors when you use the **inventory.yml** inventory file. Also verify that the playbook output contains inventory host names that follow the stated naming conventions.

After the playbook executes without error, commit the new **inventory.yml** file to the local repository. Verify that all project changes are committed, and push all local commits to the remote repository.

Evaluation

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

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

Finish

From **workstation**, run the **lab inventory-review finish** command to complete this lab.

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

This concludes the lab.



References

For more information about A/B testing and deployments, see the *A/B Testing* article from Wikipedia:

A/B Testing

https://en.wikipedia.org/wiki/A/B_testing

► Solution

Managing Inventories

Performance Checklist

In this lab, you will convert an INI-style inventory file to YAML format and configure group variable files for two new host groups.

Outcomes

You should be able to:

- Organize playbook variables in a directory structure.
- Write an inventory file in YAML format.
- Assign arbitrary host names for remote hosts in an inventory.

Before You Begin

Log in as the **student** user on **workstation** and run **lab inventory-review start**. This script initializes the remote Git repository that you need for this lab.

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

The Git repository contains a **site.yml** playbook that configures a front end load balancer and a pool of backend web servers. The back server pool is partitioned into two groups, **A** and **B**. Each pool partition deploys an independent version of the web application.

1. Clone the Git repository <http://git.lab.example.com:8081/git/inventory-review.git> in the **/home/student/git-repos** directory. Review the project playbooks and the **inventory** file and then execute the **site.yml** playbook. Make several web requests to **servera** to verify that the same version of the web application is deployed to all back end web servers.
 - 1.1. From a terminal, create the directory **/home/student/git-repos** if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. From a terminal, create the directory **/home/student/git-repos** if it does not already exist. Then, change to this directory and clone the repository:

```
[student@workstation ~]$ cd git-repos/
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/inventory-review.git
Cloning into 'inventory-review'...
remote: Enumerating objects: 56, done.
remote: Counting objects: 100% (56/56), done.
remote: Compressing objects: 100% (42/42), done.
```

```
remote: Total 56 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (56/56), done.
[student@workstation git-repos]$ cd inventory-review
```

1.3. Review the **inventory** file.

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

[web_servers]

[web_servers:children]
a_web_servers
b_web_servers

# Group "A" of Web Servers
[a_web_servers]
serverb.lab.example.com

# Group "B" of Web Servers
[b_web_servers]
serverc.lab.example.com
```

In this project, the **web_servers** host group is composed of two other groups: **a_web_servers** and **b_web_servers**.

1.4. Execute the **site.yml** playbook

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

PLAY [Ensure HAProxy is deployed] ****
...output omitted...

PLAY RECAP ****
servera.lab.example.com    : ok=7      changed=6      unreachable=0      ...
serverb.lab.example.com    : ok=8      changed=6      unreachable=0      ...
serverc.lab.example.com    : ok=8      changed=6      unreachable=0      ...
```

1.5. Verify that the same version of the web application is deployed to all back end web servers.

```
[student@workstation inventory-review]$ curl servera
Hello from serverb.lab.example.com. (version v1.1)
[student@workstation inventory-review]$ curl servera
Hello from serverc.lab.example.com. (version v1.1)
```

2. Create a variable file for the **a_web_servers** host group to set the **webapp_version** variable to the value **v1.1a**. Ensure the variable file is saved in the appropriate directory with the name **webapp.yml**.

Create a similar variable file for the **b_web_servers** host group, but set the value of the **webapp_version** variable to **v1.1b**.

- 2.1. Create directories to hold group variable files for the **a_web_servers** and **b_web_servers** host groups.

```
[student@workstation inventory-review]$ mkdir -pv group_vars/{a,b}_web_servers
mkdir: created directory 'group_vars/a_web_servers'
mkdir: created directory 'group_vars/b_web_servers'
```

- 2.2. Create the **webapp.yml** variable file for the **a_web_servers** group. Use it to set the value of the **webapp_version** variable to **v1.1a**.

```
[student@workstation inventory-review]$ echo "webapp_version: v1.1a" > \
> group_vars/a_web_servers/webapp.yml
```

- 2.3. Create the **webapp.yml** variable file for the **b_web_servers** group. Use it to set the value of the **webapp_version** variable to **v1.1b**.

```
[student@workstation inventory-review]$ echo "webapp_version: v1.1b" > \
> group_vars/b_web_servers/webapp.yml
```

3. Execute the **deploy_webapp.yml** playbook. Use **curl** or a web browser to confirm that requests for web pages sent to **servera** produce two different versions of the web application.

Commit these changes to your local repository.

- 3.1. Execute the **deploy_webapp.yml** playbook:

```
[student@workstation inventory-review]$ ansible-playbook deploy_webapp.yml

PLAY [Ensure Web App is deployed] ****
...output omitted...

PLAY RECAP ****
serverb.lab.example.com  : ok=2    changed=1    unreachable=0    ...
serverc.lab.example.com  : ok=2    changed=1    unreachable=0    ...
```

- 3.2. Verify that requests to **servera** produce two different versions of the web application.

```
[student@workstation inventory-review]$ curl servera
Hello from serverb.lab.example.com. (version v1.1a)
[student@workstation inventory-review]$ curl servera
Hello from serverc.lab.example.com. (version v1.1b)
```

- 3.3. Commit these changes to your local repository.

```
[student@workstation inventory-review]$ git status
On branch master
Your branch is up to date with 'origin/master'.
```

```

Untracked files:
(use "git add <file>..." to include in what will be committed)

group_vars/a_web_servers/
group_vars/b_web_servers/

nothing added to commit but untracked files present (use "git add" to track)

[student@workstation inventory-review]$ git add \
> group_vars/{a,b}_web_servers
[student@workstation inventory-review]$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

new file:   group_vars/a_web_servers/webapp.yml
new file:   group_vars/b_web_servers/webapp.yml
[student@workstation inventory-review]$ git commit \
> -m "Created variable files for the A and B groups."

```

4. Create a new YAML-formatted inventory file named **inventory.yml**. Ensure that this file contains the same inventory data as the INI-formatted **inventory** file. To test the YAML inventory file you created, execute the **site.yml** file using this inventory file.

When complete, the project contains both an **inventory** and **inventory.yml** file.

- 4.1. Create a new YAML-formatted inventory file, **inventory.yml**, that contains the same data as the INI-formatted inventory file:

```

lb_servers:
  hosts:
    servera.lab.example.com:
web_servers:
  children:
    a_web_servers:
      hosts:
        serverb.lab.example.com:
    b_web_servers:
      hosts:
        serverc.lab.example.com:

```

- 4.2. Test the new inventory file.

```

[student@workstation inventory-review]$ ansible-playbook site.yml -i \
> inventory.yml

PLAY [Ensure HAProxy is deployed] ****
...output omitted...
PLAY RECAP ****

```

```
servera.lab.example.com : ok=5    changed=0    unreachable=0    ...
serverb.lab.example.com : ok=7    changed=0    unreachable=0    ...
serverc.lab.example.com : ok=7    changed=0    unreachable=0    ...
```

5. Use **ansible_host** variables to configure the real host names that are used for connections when Ansible manages these hosts. Modify the **inventory.yml** file so that the managed hosts are referred to in the inventory using the following naming convention:

- Managed hosts in the **lb_servers** host group have an inventory name matching **loadbalancer_N**, where N is assigned sequentially to servers starting at 1.
- The **web_servers** host group has two child groups, **a_web_servers** and **b_web_servers**.
- Hosts in the **a_web_servers** host group follow a similar numbered naming scheme of the form: **backend_aN**.
- Hosts in the **b_web_servers** host group also follow a similar numbered naming scheme of the form **backend_bN**.

- 5.1. Modify the **inventory.yml** so that each server follows its naming convention:

```
lb_servers:
  hosts:
    loadbalancer_1:
      ansible_host: servera.lab.example.com

web_servers:
  children:
    a_web_servers:
      hosts:
        backend_a1:
          ansible_host: serverb.lab.example.com
    b_web_servers:
      hosts:
        backend_b1:
          ansible_host: serverc.lab.example.com
```

6. Verify that the **site.yml** playbook executes without errors when you use the **inventory.yml** inventory file. Also verify that the playbook output contains inventory host names that follow the stated naming conventions.

After the playbook executes without error, commit the new **inventory.yml** file to the local repository. Verify that all project changes are committed, and push all local commits to the remote repository.

- 6.1. Verify that the **site.yml** playbook executes without errors, and that playbook output contains inventory host names that follow the naming conventions.

```
[student@workstation inventory-review]$ ansible-playbook site.yml -i \
> inventory.yml

PLAY [Ensure HAProxy is deployed] ****
...output omitted...
```

```
PLAY RECAP ****
backend_a1           : ok=7      changed=1    unreachable=0      ...
backend_b1           : ok=7      changed=1    unreachable=0      ...
loadbalancer_1       : ok=5      changed=0    unreachable=0      ...
```

6.2. Commit the new **inventory.yml** file.

```
[student@workstation inventory-review]$ git add inventory.yml
[student@workstation inventory-review]$ git commit -m "Added YAML inventory"
```

6.3. Verify that all project changes are committed, and push all local commits to the remote repository.

```
[student@workstation inventory-review]$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
[student@workstation inventory-review]$ git push
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (10/10), 904 bytes | 301.00 KiB/s, done.
Total 10 (delta 2), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/inventory-review.git
  eb2c922..0564e9a  master -> master
```

Evaluation

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

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

Finish

From **workstation**, run the **lab inventory-review finish** command to complete this lab.

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

This concludes the lab.

Summary

In this chapter, you learned:

- How to write static YAML inventory files.
- How to convert a INI inventory file to a YAML inventory file.
- How to use the **group_vars** directory to improve playbook project maintainability.
- How to use special inventory variables to control connection to hosts, and make playbook output more readable.

Chapter 3

Managing Task Execution

Goal

Control and optimize the execution of tasks by Ansible Playbooks.

Objectives

- Control automatic privilege escalation at the play, role, task, or block level.
- Configure tasks that can run before roles or after normal handlers, and notify multiple handlers at once.
- Label tasks with tags, and run only tasks labeled with specific tags or start playbook execution at a specific task.
- Optimize your playbook to run more efficiently, and use callback plugins to profile and analyze which tasks consume the most time.

Sections

- Controlling Privilege Escalation (and Guided Exercise)
- Controlling Task Execution (and Guided Exercise)
- Running Selected Tasks (and Guided Exercise)
- Optimizing Execution for Speed (and Guided Exercise)

Lab

Managing Task Execution

Controlling Privilege Escalation

Objectives

After completing this section, you should be able to:

- Control automatic privilege escalation.
- Narrow privilege escalation to the relevant context.

Privilege Escalation Strategies

Ansible Playbooks can achieve privilege escalation at many levels. Depending on the level on which you intend to control the privilege escalation, Ansible uses directives or connection variables. For plays, roles, blocks, and tasks you need to use the privilege escalation directives: **become**, **become_user**, **become_method**, and **become_flags**.

Privilege Escalation by Configuration

If you have the **become** Boolean set to **yes** (or **true**) in the **privilege_escalation** section of your Ansible configuration file, then all plays in your playbook will use privilege escalation by default. Those plays will use the current **become_method** to switch to the current **become_user** when running on the managed hosts.

You can also override the configuration file and specify privilege escalation settings when you launch the playbook using command-line options. The following table compares configuration directives and command-line options:

Privilege Escalation Directives and Options

Configuration Directive	Command-line Option
become	--become or -b
become_method	--become-method=BECOME_METHOD
become_user	--become-user=BECOME_USER
become_password	--ask-become-pass or -K



Important

If the Ansible configuration file specifies **become: false**, but the command-line includes the **-b** option, then Ansible will ignore the configuration file and uses privilege escalation by default.

Privilege Escalation in Plays

When you write a playbook, you might need privilege escalation for some plays but not for all plays. You can explicitly specify whether privilege escalation should be used by each play.

If the play does not specify whether to use privilege escalation, the default setting from the configuration file or the command-line is used.

Here is an example of a playbook that contains three plays. The first play uses **become: true** to use privilege escalation no matter what the configuration file or command-line options specify. The second play uses **become: false** to not use privilege escalation, even if the configuration file or command-line options specify to do so. The third play does not have a **become** directive and will use privilege escalation based on the default settings from the Ansible configuration file or the command line. The **ansible_user_id** variable displays the user name of the user on the managed host that is running the current play.

```
---
- name: Become the user "manager"
  hosts: webservers
  become: true
  tasks:
    - name: Show the user used by this play
      debug:
        var: ansible_user_id

- name: Do not use privilege escalation
  hosts: webservers
  become: false
  tasks:
    - name: Show the user used by this play
      debug:
        var: ansible_user_id

- name: Use privilege escalation based on defaults
  hosts: webservers
  tasks:
    - name: Show the user used by this play
      debug:
        var: ansible_user_id
```

To ensure that privilege escalation is used or is not used by your play, specify the setting explicitly in the playbook. Specifying the escalation method, or privileged user in configuration settings or inventory variables, may be necessary depending on the play or the hosts in question.

Privilege Escalation in Tasks

You can also turn on (or off) privilege escalation for just one task in the play. To do this, add the appropriate **become** directive to the task.

```
---
- name: Play with two tasks, one uses privilege escalation
  hosts: all
  become: false
  tasks:
    - name: This task needs privileges
      yum:
        name: perl
        state: installed
      become: true
```

```
- name: This task does not need privileges
  shell: perl -v
  register: perlcheck
  failed_when: perlcheck.rc != 0
```

In the preceding example, the play has privilege escalation turned off by default, but the first task uses privilege escalation.

Privilege Escalation in Blocks

If you have a subset of tasks in your play that require (or do not require) privilege escalation, you can set **become** on a **block**. All tasks in the **block** share the same privilege escalation setting, and this setting overrides the setting made at the play level.

The following examples show things you can do with this mechanism:

- Turn on privilege escalation for a subset of tasks in your play.
- Turn off privilege escalation for a subset of tasks in your play.
- Use with **become_user** to use privilege escalation to perform a subset of tasks as some other regular user used by your application (such as a database user) instead of **root**.

To use this option, add the necessary directives to the **block** dictionary. In the following example, the playbook contains one play that has privilege escalation turned off by default. The play consists of a block (containing two tasks) and a task. The tasks in the block will use privilege escalation, even though it is turned off by default for the play. The final task will not use privilege escalation because it is outside of the block.

```
---
- name: Deploy web services
  hosts: webservers
  become: false
  tasks:
    - block:
        - name: Ensure httpd is installed
          yum:
            name: httpd
            state: installed
        - name: Start and enable webserver
          service:
            name: httpd
            state: started
            enabled: yes
          become: true
    - name: Test website from itself, do not become
      uri:
        url: http://{{ ansible_host }}
        return_content: yes
      register: webpage
      failed_when: webpage.status != 200
```

Privilege Escalation in Roles

There are two basic ways for a role to perform privilege escalation.

- The first option is for the role itself to have privilege escalation variables set inside it, or on its tasks. The role's documentation may specify whether you have to set other variables such as **become_method** to use the role.
- You can also specify this information yourself, in the Ansible configuration or playbook.

Alternatively, you can set the privilege escalation settings on the play that calls the role, as previously discussed. Or, you can adjust these settings on individual roles, just like you can on individual tasks or blocks:

```
- name: Example play with one role
  hosts: localhost
  roles:
    - role: role-name
      become: true
```

Privilege Escalation with Connection Variables

You can also use connection variables to configure privilege escalation. These connection variables can be applied as inventory variables on groups or individual hosts.

The following table compares playbook and configuration directives with connection variable names:

Privilege Escalation Directives and Connection Variables

Configuration or Playbook Directive	Connection Variable
become	ansible_become
become_method	ansible_become_method
become_user	ansible_become_user
become_password	ansible_become_pass



Important

Connection variables override the **become** settings in the configuration file, as well as in plays, tasks, blocks, and roles.

Be cautious when using these variables to configure privilege escalation because of their high precedence compared to the configuration and playbook directives. Using the configuration or playbook directives will generally give you more flexibility and control of whether privilege escalation is used.

The following example demonstrates using privilege escalation variables in a YAML inventory for a group:

```

webservers:
  hosts:
    servera.lab.example.com:
    serverb.lab.example.com:
  vars:
    ansible_become: true

```

You can set connection variables for privilege escalation at host level. Doing this on a host per host basis is tedious, but can be helpful in case one host in a larger group needs specific connection options.

The following example demonstrates configuring privilege escalation using connection variables at host level:

```

webservers:
  hosts:
    servera.lab.example.com:
      ansible_become: true
    serverb.lab.example.com:

```

You can also set connection variables in the playbook, for example on the play itself. Doing so will override the inventory variables (through normal variable precedence), as well as the setting of any **become** directives.

```

---
- name: Example play using connection variables
  hosts: webservers
  vars:
    ansible_become: true
  tasks:
    - name: Play will use privilege escalation even if inventory says no
      yum:
        name: perl
        state: installed

```

Choosing Privilege Escalation Approaches

As you can see, there are many ways to control privilege escalation. So, how should you choose which method to use? Consider the following competing needs when choosing how to control privilege escalation:

- Keeping your playbook simple (a key principle of Ansible best practices) is the first consideration.
- Running tasks with the least privilege when possible (to avoid inadvertent compromises and damage to managed hosts through playbook errors) is a second consideration.

Many people who are new to Ansible run their playbooks with privilege escalation always turned on. This approach is simple, and in many cases the tasks that you are performing must be run as **root**. However, tasks that *do not* need to be run as **root** are nevertheless running with elevated privileges, which could increase risk.

Some users connect directly to the **root** account to avoid privilege escalation. This is generally not a good practice, because anyone running the playbook must have **root** credentials on the

managed hosts. This can also make it hard to evaluate which administrator performed which playbook run.

Therefore, a good practice is to selectively control which plays or tasks need privilege escalation. For example, if the **apache** user can start the **httpd** server, there is no need to run the task as the **root** user.

Ideally, you should configure privilege escalation in as simple a way as possible, and it should be clear to you whether it is in use for a task. For example, you could use **become: true** on a play to turn privilege escalation on for a play, and then selectively disable tasks that do not need escalation with **become: false** on the task. Alternatively, you could group tasks that need privilege escalation into one play and tasks that do not into another play, if compatible with the workflow.

Setting privilege escalation in the configuration file may be necessary if a playbook requires privilege escalation but you cannot edit it for some reason. Generally, if the playbook requires privilege escalation then it may make more sense for the playbook to indicate that requirement. The biggest challenge is when different hosts that use a playbook require different privilege escalation methods in order to function correctly. In that case, it can make sense to set inventory variables like **ansible_become_method** for those hosts or their group, while enabling whether privilege escalation is used through **become** in the playbook.



References

Understanding Privilege Escalation – Ansible Documentation

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

Blocks – Ansible Documentation

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

► Guided Exercise

Controlling Privilege Escalation

In this exercise, you will configure a playbook to escalate privileges only for specific plays, roles, tasks, or blocks that might need them to operate correctly.

Outcomes

You should be able to select the appropriate escalation method and privilege isolation.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab task-escalation start
```

- 1. Clone the Git repository <http://git.lab.example.com:8081/git/task-escalation.git> in the **/home/student/git-repos** directory.
 - 1.1. From a terminal, create the directory **/home/student/git-repos** if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- 1.3. Clone the repository:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/task-escalation.git
Cloning into 'task-escalation'...
remote: Enumerating objects: 56, done.
remote: Counting objects: 100% (56/56), done.
remote: Compressing objects: 100% (42/42), done.
remote: Total 56 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (56/56), done.
```

- 2. Remove the global privilege escalation from the playbook configuration.

- 2.1. Edit the **ansible.cfg** file.

```
[student@workstation git-repos]$ cd task-escalation
[student@workstation task-escalation]$ vi ansible.cfg
```

- 2.2. Remove the **privilege_escalation** block. After completing these steps, the file should display as follows:

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

- 3. Add the privilege escalation directives to the tasks.

- 3.1. Edit the **roles/firewall/tasks/main.yml** file:

```
[student@workstation task-escalation]$ vi roles/firewall/tasks/main.yml
```

Add the **become: true** directive to the **firewalld** task. The resulting file should display as follows:

```
---
# tasks file for firewall

- name: Ensure Firewall Sources Configuration
  firewalld:
    source: "{{ item.source if item.source is defined else omit }}"
    zone: "{{ item.zone if item.zone is defined else omit }}"
    permanent: true
    state: "{{ item.state | default('enabled') }}"
    service: "{{ item.service if item.service is defined else omit }}"
    immediate: true
    port: "{{ item.port if item.port is defined else omit }}"
    loop: "{{ firewall_rules }}"
    notify: reload firewalld
become: true
```

- 3.2. Edit the **roles/firewall/handlers/main.yml** file:

```
[student@workstation task-escalation]$ vi roles/firewall/handlers/main.yml
```

Add the **become: true** directive to the **firewalld** restart task. The resulting file should display as follows:

```
---
# handlers file for firewall

- name: reload firewalld
  service:
    name: firewalld
    state: reloaded
become: true
```

- 3.3. Edit the **roles/haproxy/tasks/main.yml** file:

```
[student@workstation task-escalation]$ vi roles/haproxy/tasks/main.yml
```

Surround the tasks in a block and add the **become: true** directive to the block. Remember to indent the tasks inside the block. The resulting file should display as follows:

```
---
# tasks file for haproxy

- block:
  - name: Ensure haproxy packages are present
    yum:
      name:
        - haproxy
        - socat
      state: present

  - name: Ensure haproxy is started and enabled
    service:
      name: haproxy
      state: started
      enabled: true

  - name: Ensure haproxy configuration is set
    template:
      src: haproxy.cfg.j2
      dest: /etc/haproxy/haproxy.cfg
      #validate: haproxy -f %s -c -q
      notify: reload haproxy
become: true
```

3.4. Edit the **roles/haproxy/handlers/main.yml** file:

```
[student@workstation task-escalation]$ vi roles/haproxy/handlers/main.yml
```

Surround the handlers with a block and add the **become: true** directive to the block. Remember to indent the handlers inside the block. The resulting file should display as follows:

```
---
# handlers file for haproxy

- block:
  - name: restart haproxy
    service:
      name: haproxy
      state: restarted

  - name: reload haproxy
    service:
      name: haproxy
      state: reloaded
become: true
```

3.5. Edit the **roles/apache/tasks/main.yml** file:

```
[student@workstation task-escalation]$ vi roles/apache/tasks/main.yml
```

Surround the tasks with a block and add the **become: true** directive to the block. Remember to indent the task inside the block. The resulting file should display as follows:

```
---
# tasks file for apache

- block:
  - name: Ensure httpd packages are installed
    yum:
      name:
        - httpd
        - php
        - git
        - php-mysqld
      state: present

  - name: Ensure SELinux allows httpd connections to a remote database
    seboolean:
      name: httpd_can_network_connect_db
      state: true
      persistent: true

  - name: Ensure httpd service is started and enabled
    service:
      name: httpd
      state: started
      enabled: true
become: true
```

3.6. Edit the **roles/webapp/tasks/main.yml** file:

```
[student@workstation task-escalation]$ vi roles/webapp/tasks/main.yml
```

Add the **become: true** directive to the **copy** task. The resulting file should display as follows:

```
---
# tasks file for webapp

- name: Ensure stub web content is deployed
  copy:
    content: "{{ webapp_message }}. (version {{ webapp_version}})\n"
    dest: /var/www/html/index.html
become: true
```

► 4. Run the playbook using the **ansible-playbook** command:

```
[student@workstation task-escalation]$ ansible-playbook site.yml
...output omitted...
PLAY RECAP ****
servera.lab.example.com    : ok=5      changed=4      unreachable=0      failed=0 ...
serverb.lab.example.com    : ok=5      changed=3      unreachable=0      failed=0 ...
serverc.lab.example.com    : ok=5      changed=3      unreachable=0      failed=0 ...
```

- ▶ 5. Test that the playbook has run correctly:

```
[student@workstation task-escalation]$ curl servera.lab.example.com
Hello from serverb.lab.example.com. (version v1.0)
```

Finish

On **workstation**, run the **lab task-escalation finish** script to complete this lab.

```
[student@workstation task-escalation]$ lab task-escalation finish
```

This concludes the guided exercise.

Controlling Task Execution

Objectives

After completing this section, you should be able to configure tasks that can run before roles or after normal handlers, and notify multiple handlers at once.

Controlling Order of Execution

In a play, Ansible always runs the tasks from roles, called by **roles**, before the tasks that you define under the **tasks** section. The following playbook contains both a **roles** and a **tasks** section.

```
---
- name: Ensure Apache is deployed
  hosts: www1.example.com
  gather_facts: no

  tasks:
    - name: Open the firewall
      firewalld:
        service: http
        permanent: yes
        state: enabled

  roles:
    - role: apache
```

When you run this playbook, Ansible executes the tasks from the role before the **Open the firewall** task, even though the **tasks** section is defined first:

```
[user@demo ~]$ ansible-playbook deploy_apache.yml

PLAY [Ensure Apache is deployed] ****
TASK [apache : Ensure httpd packages are installed] ****
changed: [www1.example.com]

TASK [apache : Ensure httpd service is started and enabled] ****
changed: [www1.example.com]

TASK [Open the firewall] ****
changed: [www1.example.com]

PLAY RECAP ****
www1.example.com      : ok=3      changed=3      unreachable=0      ...
```

For readability, it is a good practice to write the **tasks** section after the **roles** section, so the order of the playbook matches the order of execution.

**Note**

This order comes from the fact that each play listed in the playbook is a YAML dictionary of key-value pairs. So, the order of the top-level directives (**name**, **hosts**, **tasks**, **roles** and so on) is actually arbitrary, but Ansible handles them in a standardized order when it parses and runs the play.

It is a good practice to write your plays in a consistent order starting with the name of the play, but this is not actually required by Ansible. However, deviating from this convention will make it harder to read your playbook, and therefore is not recommended.

Importing or Including Roles as a Task

Recent versions of Ansible allow you to include or import roles as a task instead of using the **roles** section of the play. The advantage of this approach is that you can easily run a set of tasks, import or include a role, and then run more tasks. A potential disadvantage is that it may be less clear which roles your playbook is using without inspecting it closely.

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: Executing 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:
        name: role2

    - name: Another normal task
      debug:
        msg: 'second task'
```

With **import_role**, the **ansible-playbook** command starts by parsing and inserting the role in the play before starting the execution. Ansible detects and reports syntax errors immediately and does not start the playbook execution.

With **include_role**, however, Ansible parses and inserts the role in the play when it reaches the **include_role** task, during the play execution. If Ansible detects syntax errors in the role, then execution of the playbook is aborted.

However, the **when** directive behaves differently. With the **include_role** task, Ansible does not parse the role if the condition in the **when** directive is false.

Defining Pre and Post Tasks

You may want a play that runs some tasks, and the handlers they notify, before your roles. You may also want to run tasks in the play after your normal tasks and handlers run. There are two directives you can use instead of **tasks** to do this:

- **pre_tasks** is a **tasks** section that runs before the **roles** section.
- **post_tasks** is a **tasks** section that runs after the **tasks** section and any handlers notified by **tasks**.

The following playbook provides an example with **pre_tasks**, **roles**, **tasks**, **post_tasks**, and **handlers** sections. It is unusual for a play to contain all of these sections.

```
---
- name: Deploying New Application Version
  hosts: webservers

  pre_tasks:
    # Stop monitoring the web server to avoid sending false alarms
    # while the service is updating.
    - name: Disabling Nagios for this host
      nagios:
        action: disable_alerts
        host: "{{ inventory_hostname }}"
        services: http
        delegate_to: nagios-srv

  roles:
    - role: deploy-content

  tasks:
    - name: Restarting memcached
      service:
        name: memcached
        status: restarted
      notify: Notifying the support team

    # Confirm that the application is fully operational
    # after the update.
    - name: Validating the new deployment
      uri:
        url: "http://{{ inventory_hostname }}/healthz"
        return_content: yes
      register: result
      failed_when: "'OK' not in result.content"

  post_tasks:
    - name: Enabling Nagios for this host
      nagios:
        action: enable_alerts
        host: "{{ inventory_hostname }}"
        services: http
        delegate_to: nagios-srv

  handlers:
```

```
# Send a message to the support team through Slack
# every time the memcached service is restarted.
- name: Notifying the support team
  slack:
    token: G922VJP25/D923DW937/3Ffe373sfhRE6y52Fg3rvf5G1K
    msg: 'Memcached on {{ inventory_hostname }} restarted'
    delegate_to: localhost
    become: false
```

Reviewing the Order of Execution

Ansible runs the play sections in the following order:

- **pre_tasks**
- handlers notified in the **pre_tasks** section
- **roles**
- **tasks**
- handlers notified in the **roles** and **tasks** sections
- **post_tasks**
- handlers notified in the **post_tasks** section

The order of these sections in a play does not modify the order of execution, as given above. For example, if you write the **tasks** section before the **roles** section, Ansible still executes the roles before the tasks in the **tasks** section. For readability, however, it is a good practice to organize your play in the order of execution: **pre_tasks**, **roles**, **tasks**, and **post_tasks**. Handlers are usually at the end.

Ansible executes and flushes notified handlers at several points during a run: after the **pre_tasks** section, after the **roles** and **tasks** sections, and after the **post_tasks** section. This means that a handler can run more than once, at different times during the play execution, if notified in multiple sections.

To immediately run any handlers that have been notified by a particular task in the play, add a task that uses the **meta** module with the **flush_handlers** parameter. This allows you to define specific points during task execution when all notified handlers are run.

In the following example, the play executes the **Restart api server** handler, if notified, after deploying a new configuration file and before using the application API. Without this call to the **meta** module, the play only calls the handler after running all the tasks. If the handler has not run before the last task using the API is performed, then that task may fail because the configuration file was updated, but the application has not yet reread the new configuration.

```
---
- name: Updating the application configuration and cache
  hosts: app_servers

  tasks:
    - name: Deploying the configuration file
      template:
        src: api-server.cfg.j2
        dest: /etc/api-server.cfg
```

```

    notify: Restart api server

    - name: Running all notified handlers
      meta: flush_handlers

    - name: Asking the API server to rebuild its internal cache
      uri:
        url: "https://{{ inventory_hostname }}/rest/api/2/cache/"
        method: POST
        force_basic_auth: yes
        user: admin
        password: redhat
        body_format: json
        body:
          type: data
          delay: 0
          status_code: 201

    handlers:
      - name: Restart api server
        service:
          name: api-server
          state: restarted
          enabled: yes

```

Remember that in a play, handlers have global scope. A play can notify handlers defined in roles. One role can notify a handler defined by another role or by the play.

Ansible always runs handlers that have been notified in the order they are listed in the **handlers** section of the play, and not in the order in which they were notified.

Listening to Handlers

In addition to being notified by tasks, a handler can also "subscribe" to a specific notification, and run when that notification is triggered. This allows multiple handlers to be triggered by one notification.

By default, a handler executes when a notification string matches the handler name. However, as each handler must have a unique name, the only way to trigger multiple handlers at the same time is that each one subscribes to the same notification name.

The following example shows a task that notifies **My handlers** when it changes, which is whenever it runs, because it has **changed_when: true** set. Any handler with the name **My handlers**, or that lists **My handlers** in a **listen** directive, will be notified.

```

---
- name: Testing the listen directive
  hosts: localhost
  gather_facts: no
  become: no

  tasks:
    - debug:
        msg: Trigerring the handlers
      notify: My handlers

```

```

changed_when: true

handlers:
  # Listening to the "My handlers" event
  - name: Listening to a notification
    debug:
      msg: First handler was notified
    listen: My handlers

  # As an example, this handler is also triggered because
  # its name matches the notification, but no two handlers
  # can have the same name.
  - name: My handlers
    debug:
      msg: Second handler was notified

```

**Note**

In the preceding playbook, it would be better if both handlers had unique names (not **My handlers**) and used **listen: My handlers**, because the playbook would be easier to read.

The **listen** directive is particularly helpful when used with roles. Roles use notifications to trigger their handlers. A role can document that it will notify a certain handler when an event occurs. Other roles or a play might be able to use this notification to run additional handlers defined outside the role.

For example, a role may notify one of its handlers when a service needs restarting. In your playbook, you can define a handler that listens to that notification and performs additional tasks when Ansible restarts the service, such as sending a message to a monitoring tool, or restarting a dependent service.

For example, you can create a role that checks the validity of a given SSL certificate and notifies a handler that renews the certificate if expired. In a playbook, you can call this role to check your Apache HTTP Server certificate validity, and create a handler that will listen for that notification to restart the **httpd** service if the role renews the certificate.

Approaches to Handler Notification

To summarize, a task can notify multiple handlers in at least two ways:

- It can notify a list of handlers individually by **name**.
- It can notify one name for which multiple handlers are configured to **listen**.

Which approach is better? If the handlers are reused as a set by multiple tasks, then the easiest way to configure your playbook is to use the second approach with the **listen** directive. Then, you need only change the handlers to mark whether they are included in the set, and to ensure that they are listed in the correct order.

This approach is useful when a task needs to notify a lot of handlers. Instead of notifying each handler by name, the task only sends one notification. Ansible triggers all the handlers listening to that notification. This way, you can add or remove handlers without updating the task.

In the first approach, you must find and edit every affected task in order to add the handler to the list. You must also ensure that the handlers are listed in the correct order in the **handlers** section.



Note

It is an error to have a task send a notification with no handler matching that notification.

```
[user@demo ~]$ cat no_handler_error.yml
---
- name: Testing notification with no handler
  hosts: localhost
  gather_facts: no
  become: no

  tasks:
    - debug:
        msg: Trigerring a non existent handler
        changed_when: true
        notify: Restart service

[user@demo ~]$ ansible-playbook no_handler_error.yml
PLAY [Testing notification with no handler] ****
TASK [debug] ****
ERROR! The requested handler 'Restart service' was not found in either the
main handlers list nor in the listening handlers list
```

Controlling Order of Host Execution

Ansible determines which hosts to manage for a play based on the play's **hosts** directive. By default, Ansible 2.4 and later runs the play against hosts in the order in which they are listed in the inventory. You can change that order on a play-by-play basis by using the **order** directive.

The following playbook alphabetically sorts the hosts in the **web_servers** group before running the task:

```
---
- name: Testing host order
  hosts: web_servers
  order: sorted

  tasks:
    - name: Creating a test file in /tmp
      copy:
        content: 'This is a sample file'
        dest: /tmp/test.out
```

The **order** directive accepts the following values:

inventory

The inventory order. This is the default value.

reverse_inventory

The reverse of the inventory order.

sorted

The hosts in alphabetical order. Numbers sort before letters.

reverse_sorted

The hosts in reverse alphabetical order.

shuffle

Random order every time you run the play.

**Note**

Because Ansible normally runs each task in parallel on several hosts, the output of the **ansible-playbook** command might not reflect the expected order: the output shows the task completion order rather than the execution order.

For example, assume that the inventory is configured so that hosts are listed in alphabetical order, and plays are run on managed hosts in inventory order (the default behavior). You could still see output like this:

```
...output omitted...
TASK [Creating a test file in /tmp] ****
changed: [www2.example.com]
changed: [www3.example.com]
changed: [www1.example.com]
...output omitted...
```

Notice that the task on **www1.example.com** completed last.

**References****Roles – Ansible Documentation**

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

Handlers: Running Operations On Change – Ansible Documentation

https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#handlers-running-operations-on-change

Creating Reusable Playbooks – Ansible Documentation

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

► Guided Exercise

Controlling Task Execution

In this exercise, you will use **pre_tasks** and **post_tasks** sections to control whether tasks run before or after roles, and **listen** directives to notify multiple handlers at the same time.

Outcomes

You should be able to:

- Control the execution order of tasks.
- Trigger handlers using the **listen** directive.

Before You Begin

Log in as the **student** user on **workstation** and run the **lab task-execution start** command.

```
[student@workstation ~]$ lab task-execution start
```

This script creates an Ansible project in **/home/student/D0447/labs/task-execution/**.

- 1. From a terminal, change to the **/home/student/D0447/labs/task-execution/** directory and review the **deploy_haproxy.yml** playbook.

- 1.1. Change to the **/home/student/D0447/labs/task-execution/** directory.

```
[student@workstation ~]$ cd ~/D0447/labs/task-execution
[student@workstation task-execution]$
```

- 1.2. Review the contents of the **deploy_haproxy.yml** playbook.

```
[student@workstation task-execution]$ cat deploy_haproxy.yml
- name: Ensure HAProxy is deployed
  hosts: lb_servers
  force_handlers: True

  roles:
    # The "haproxy" role has a dependency on the "firewall" role.
    # The "firewall" role requires a "firewall_rules" variable be defined.
    - role: haproxy
```

Notice that the playbook only calls one role: **haproxy**.

- 2. Configure the playbook to add a *message of the day* (MOTD) on the HAProxy servers when a maintenance operation is in progress. This way, when a member of the support team logs in, the system notifies them of the maintenance window.

To configure this message, update the `deploy_haproxy.yml` playbook to add a task that creates a message file in the `/etc/motd.d/` directory of the load balancers. Because the task must run before the role, add it to the `pre_tasks` section. Use the `copy` module to create the maintenance file in the `/etc/motd.d/` file, with the `Maintenance in progress\n` contents.

Add a `post_tasks` section, and include a task to remove the message after the `haproxy` role completes.

- 2.1. Edit the `deploy_haproxy.yml` playbook to add the `pre_tasks` section. For your convenience, you can copy and paste that section from the `~/DO447/solutions/task-execution/deploy_haproxy.yml` file.

```
- name: Ensure HAProxy is deployed
hosts: lb_servers
force_handlers: True

pre_tasks:
- name: Setting the maintenance message
  copy:
    dest: /etc/motd.d/maintenance
    content: "Maintenance in progress\n"

roles:
# The "haproxy" role has a dependency on the "firewall" role.
# The "firewall" role requires a "firewall_rules" variable be defined.
- role: haproxy
```

- 2.2. After the `roles` section, add the `post_tasks` section and a task to remove the message file. Use the `file` module to remove the file.

```
- name: Ensure HAProxy is deployed
hosts: lb_servers
force_handlers: True

pre_tasks:
- name: Setting the maintenance message
  copy:
    dest: /etc/motd.d/maintenance
    content: "Maintenance in progress\n"

roles:
# The "haproxy" role has a dependency on the "firewall" role.
# The "firewall" role requires a "firewall_rules" variable be defined.
- role: haproxy

post_tasks:
- name: Removing the maintenance message
  file:
    path: /etc/motd.d/maintenance
    state: absent
```

You could define the `Removing the maintenance message` task under a `tasks` section because Ansible also runs the `tasks` section after the roles. However, to

mirror the **pre_tasks** section, it is better to define the task under the **post_tasks** section.

- 3. The playbook should also notify your support team every time Ansible reloads the **haproxy** service. The support team uses the <student@workstation.lab.example.com> email address for these kinds of notifications and also monitors the **/var/log/messages** syslog file on **workstation**. You must post the notification with both mechanisms.

To do so, review the handlers that the **haproxy** role defines and identify the handler that reloads the **haproxy** service. Edit the **deploy_haproxy.yml** playbook to add two handlers that listen to the HAProxy handler: the first uses the **mail** module and the second uses the **syslogger** module.

- 3.1. Review the handlers that the **haproxy** role defines.

```
[student@workstation task-execution]$ cat roles/haproxy/handlers/main.yml
---
# handlers file for haproxy

- name: restart haproxy
  service:
    name: haproxy
    state: restarted

- name: reload haproxy
  service:
    name: haproxy
    state: reloaded
```

Notice that the second handler reloads the **haproxy** service. Its name is **reload haproxy**.

- 3.2. Edit the **deploy_haproxy.yml** playbook to add the two handlers at the end. For your convenience, you can copy and paste the handlers from the **~/DO447/solutions/task-execution/deploy_haproxy.yml** file.

```

...output omitted...
post_tasks:
  - name: Removing the maintenance message
    file:
      path: /etc/motd.d/maintenance
      state: absent

handlers:
  - name: Sending an email to student
    mail:
      subject: "HAProxy reloaded on {{ inventory_hostname }}"
      to: student@workstation.lab.example.com
      delegate_to: localhost
      become: false
      listen: reload haproxy

  - name: Logging a message to syslog
    syslogger:
      msg: "HAProxy reloaded on {{ inventory_hostname }}"
      delegate_to: localhost
      become: false
      listen: reload haproxy

```

- 4. Run the Ansible Playbook to verify your work and confirm that Ansible triggers your handlers. When done, change to your home directory.

4.1. Run the **deploy_haproxy.yml** playbook.

```

[student@workstation task-execution]$ ansible-playbook deploy_haproxy.yml
...output omitted...
TASK [Setting the maintenance message] ****
changed: [servera.lab.example.com]

...output omitted...

RUNNING HANDLER [haproxy : reload haproxy] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [Sending an email to student] ****
ok: [servera.lab.example.com]

RUNNING HANDLER [Logging a message to syslog] ****
changed: [servera.lab.example.com]

TASK [Removing the maintenance message] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=11    changed=9    unreachable=0    ...

```

- 4.2. Use the **mail** command to confirm that Ansible has notified the **student** user of the **haproxy** service reload.

```
[student@workstation task-execution]$ mail  
Heirloom Mail version 12.5 7/5/10. Type ? for help.  
"/var/spool/mail/student": 1 message 1 new  
>N 1 root@workstation Thu May 2 "HAProxy reloaded on servera.lab.example.com"  
& q  
Held 1 message in /var/spool/mail/student
```

- 4.3. Confirm that the **/var/log/messages** file contains the notification. Use the **sudo** command to access the **/var/log/messages** file. Use **student** for the password.

```
[student@workstation task-execution]$ sudo grep \  
> "HAProxy reloaded" /var/log/messages  
[sudo] password for student: student  
...output omitted...  
May  2 11:59:10 workstation ansible_syslogger[16547]: HAProxy reloaded on  
servera.lab.example.com
```

- 4.4. Change to your home directory.

```
[student@workstation task-execution]$ cd  
[student@workstation ~]$
```

Finish

On **workstation**, run the **lab task-execution finish** command to complete this exercise.

```
[student@workstation ~]$ lab task-execution finish
```

This concludes the guided exercise.

Running Selected Tasks

Objectives

After completing this section, you should be able to label tasks with tags, run only tasks labeled with specific tags, or start playbook execution at a specific task.

Tagging Ansible Resources

When working with a large or complex playbook, you may want to run only a subset of its plays or tasks. Apply *tags* to specific resources that you may want to skip or run. A tag is a text label on an Ansible resource like a play or task. To tag a resource, use the **tags** keyword on the resource, followed by a list of the tags to apply.

When running a playbook with **ansible-playbook**, use the **--tags** option to filter the playbook and execute only specific tagged plays or tasks. Tags are available for the following resources:

- Tag each task. This is one of the most common ways tags are used.

```
- name: Install application
hosts: dbservers
vars:
  packages:
    - postfix
    - mariadb-server

tasks:
  - name: Ensure that packages are installed
    yum:
      name: "{{ packages }}"
      state: installed
    tags:
      - install
```

- Tag an entire play. Use the **tags** directive at the play level.

```
- name: Setup web services
hosts: webservers
tags:
  - setup

tasks:
  - name: Install http daemon
    yum:
      name: httpd
      state: present
```

- When including a task file in a playbook, the task can be tagged, thus allowing administrators to set a global tag for the tasks loaded by **include_tasks**:

```
tasks:
  - name: Include common web services tasks
    include_tasks: common.yml
    tags:
      - webproxy
      - webserver
```

- Tag a role in the **roles** section. All the tasks in the role are associated with this tag. In this example the role **databases** has a list of two tags, **production** and **staging**.

```
roles:
  - { role: databases, tags: ['production', 'staging'] }
```

- Tag a **block** of **tasks**. All the tasks in the block are associated with this tag. In this example we group all httpd related tasks under the **webserver** tag.

```
---
- name: Setup httpd service

tasks:
  - name: Notify start of process
    debug:
      msg: Beginning httpd setup

  - block:
    - name: Ensure httpd packages are installed
      yum:
        name:
          - httpd
          - php
          - git
          - php-mysqld
        state: present

    - name: Ensure SELinux allows httpd connections to a remote database
      seboolean:
        name: httpd_can_network_connect_db
        state: true
        persistent: true

    - name: Ensure httpd service is started and enabled
      service:
        name: httpd
        state: started
        enabled: true

tags:
  - webserver
```

 **Important**

When **roles** or **include_tasks** statements are tagged, the tag is not a way to exclude some tagged tasks the included files contain. Rather, tags in this context are a way to apply a global tag to all tasks.

Managing Tagged Resources

Use the **ansible-playbook** command to run tasks with a specific tag, using the **--tags** option, or skip tasks with a specific tag using the **--skip-tags** option.

The following playbook contains two tasks. The first task is tagged with the **webserver** tag. The second task does not have any associated tags.

```
---
- name: Example play using tagging
  hosts:
    - servera.lab.example.com
    - serverb.lab.example.com

  tasks:
    - name: httpd is installed
      yum:
        name: httpd
        state: latest
      tags: webserver

    - name: postfix is installed
      yum:
        name: postfix
        state: latest
```

To only run the first task, the **--tags** argument can be used:

```
[user@demo ~]$ ansible-playbook main.yml --tags webserver

PLAY [Example play using tagging] ****

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

TASK [httpd is installed] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

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

Because the **--tags** option was specified, the playbook ran only the task tagged with the **webserver** tag.

Specify multiple tags to **--tags** as a comma-separated list:

```
[user@demo ~]$ ansible-playbook main.yml --tags install,setup
```

Use the **--skip-tags** option to skip tasks with a specific tag and only run the tasks without that tag:

```
[user@demo ~]$ ansible-playbook main.yml --skip-tags webserver

PLAY [Example play using tagging] ****

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

TASK [postfix is installed] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

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

To list all tags that exist in a playbook, pass the **--list-tags** option to the **ansible-playbook** command. For example:

```
[student@demo examples]$ ansible-playbook playbook.yml --list-tags

playbook: playbook.yml

play #1 (webservers): Setup web services TAGS: [setup]
    TASK TAGS: [setup]

play #2 (webservers): Teardown web services TAGS: [teardown]
    TASK TAGS: [teardown]
```

Special Tags

Ansible has a special tag that can be assigned in a playbook: **always**. A resource tagged **always** will always run, even if it does not match the list of tags passed to **--tags**. The only exception is when it is explicitly skipped, using the **--skip-tags always** option.

A task you tag with the **never** special tag does not run, except when you run the playbook with the **--tags** option set to **never** or to one of the other tag associated with the task.

There are three additional special tags:

- The **tagged** tag will run any resource with an explicit tag.
- The **untagged** tag will run any resource that does not have an explicit tag, and exclude all tagged resources.
- The **all** tag will include all tasks in the play, whether they have a tag. This is the default behavior of Ansible.



References

Tags – Ansible Documentation

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

► Guided Exercise

Running Selected Tasks

In this exercise, you will use tags to run only specific tasks in a playbook.

Outcomes

You should be able to tag specific tasks in a playbook to run or skip them.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab task-tagging start
```

- 1. Clone the Git repository `http://git.lab.example.com:8081/git/task-tagging.git` in the **/home/student/git-repos** directory.

- 1.1. From a terminal, create the directory **/home/student/git-repos** if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- 1.3. Clone the repository:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/task-tagging.git
Cloning into 'task-tagging'...
remote: Enumerating objects: 56, done.
remote: Counting objects: 100% (56/56), done.
remote: Compressing objects: 100% (42/42), done.
remote: Total 56 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (56/56), done.
[student@workstation git-repos]$ cd task-tagging
```

- 2. Create a new playbook to test the successful deployment of the web application.

- 2.1. Create the **test_webapp.yml** file.

```
[student@workstation task-tagging]$ vi test_webapp.yml
```

- 2.2. Add a task that verifies that the content of the web application is accessible. Use the the **uri** module to retrieve the contents. The contents of the file should display as follows:

```
---
- name: Web application smoke test
  hosts: web_servers
  gather_facts: no
  tasks:
    - name: Verify content of http://localhost
      uri:
        url: http://localhost
        return_content: yes
        register: test_url
        failed_when: "'Hello from' not in test_url.content"
      tags:
        - tests
```

- 2.3. Add the new playbook to the **site.yml** file.

```
[student@workstation task-tagging]$ vi site.yml
```

Import the new playbook. The content of the file should display as follows:

```
- name: Deploy HAProxy
  import_playbook: deploy_haproxy.yml

- name: Deploy Web Server
  import_playbook: deploy_apache.yml

- name: Deploy Web App
  import_playbook: deploy_webapp.yml

- name: Test deployed Web App
  import_playbook: test_webapp.yml
```

- 3. Use the **site.yml** playbook to test the web application deployment only. After the test fails, execute the playbook to configure the environment, skipping all tests.

- 3.1. Use the **tests** tag to test the web application before it is setup.

```
[student@workstation task-tagging]$ ansible-playbook site.yml --tags tests
...output omitted...
TASK [Verify content of http://localhost] ****
fatal: [serverc.lab.example.com]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/libexec/platform-python"}, "changed": false, "content": "", "elapsed": 0, "failed_when_result": true, "msg": "Status code was -1 and not [200]: Request failed: <urlopen error [Errno 111] Connection refused>", "redirected": false, "status": -1, "url": "http://localhost"}
```

```

fatal: [serverb.lab.example.com]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/libexec/platform-python"}, "changed": false, "content": "", "elapsed": 0, "failed_when_result": true, "msg": "Status code was -1 and not [200]: Request failed: <urlopen error [Errno 111] Connection refused>", "redirected": false, "status": -1, "url": "http://localhost"}}

PLAY RECAP ****...
serverb.lab.example.com : ok=0     changed=0      unreachable=0    failed=1 ...
serverc.lab.example.com : ok=0     changed=0      unreachable=0    failed=1 ...

```

This command executed only the task tagged with the **tests** tag; because the server was not setup, the test failed.

- 3.2. Run the playbook, skipping the tests to setup the web application.

```

[student@workstation task-tagging]$ ansible-playbook site.yml --skip-tags tests
...output omitted...
PLAY RECAP ****...
servera.lab.example.com : ok=6     changed=6      unreachable=0    failed=0 ...
serverb.lab.example.com : ok=6     changed=6      unreachable=0    failed=0 ...
serverc.lab.example.com : ok=6     changed=6      unreachable=0    failed=0 ...

```

- 3.3. Use the **tests** tag again to test the web application.

```

[student@workstation task-tagging]$ ansible-playbook site.yml --tags tests
...output omitted...
TASK [Verify content of http://localhost] ****...
ok: [serverc.lab.example.com]
ok: [serverb.lab.example.com]

PLAY RECAP ****...
serverb.lab.example.com : ok=1     changed=0      unreachable=0    failed=0 ...
serverc.lab.example.com : ok=1     changed=0      unreachable=0    failed=0 ...

```

Now that the web application is setup, the test was successful.

Finish

On **workstation**, run the **lab task-tagging finish** script to complete this lab.

```
[student@workstation task-tagging]$ lab task-tagging finish
```

This concludes the exercise.

Optimizing Execution for Speed

Objectives

After completing this section, you should be able to optimize your playbook to run more efficiently, and use callback plug-ins to profile and analyze which tasks consume the most time.

Optimizing Playbook Execution

There are a number of ways in which you can optimize your Ansible Playbooks. Writing playbooks to run efficiently becomes increasingly important as the number of hosts you manage increases.

Optimizing the Infrastructure

Each release of Red Hat Ansible Engine adds enhancements and improvements. Running the latest version of Ansible may help increase the speed of your playbooks as the core components of Ansible and especially the modules provided with it are optimized over time.

An architectural optimization that you can make is to keep your control node "close" to the managed nodes from a networking perspective. Ansible relies heavily on network communication and transfer of data. High latency connections or low bandwidth between the control node and its managed hosts will degrade the execution time of playbooks.

Disabling Fact Gathering

Each play has a hidden task which runs first, using the **setup** module to collect facts from each host. Those facts provide some information about the nodes that plays can use through the **ansible_facts** variable.

Collecting the facts on each remote host takes time. If you do not use those facts in your play, skip the facts gathering task by setting the **gather_facts** directive to **False** (or **no**).

The following play disables facts gathering:

```
---
- name: Demonstrate disabling the facts gathering
  hosts: web_servers
  gather_facts: False

  tasks:
    - debug:
        msg: "gather_facts is set to False"
```

The following example uses the Linux **time** command to compare the execution time of the previous playbook when facts gathering is enabled, and when it is disabled.

```
[user@demo ~]$ time ansible-playbook speed_facts.yml
PLAY [Demonstrate activating the facts gathering] ****
```

```

TASK [Gathering Facts] ****
ok: [www1.example.com]
ok: [www2.example.com]
ok: [www3.example.com]

TASK [debug] ****
ok: [www1.example.com] => {
    "msg": "gather_facts is set to True"
}
ok: [www2.example.com] => {
    "msg": "gather_facts is set to True"
}
ok: [www3.example.com] => {
    "msg": "gather_facts is set to True"
}

PLAY RECAP ****
www1.example.com    : ok=1    changed=0    unreachable=0    ...
www2.example.com    : ok=1    changed=0    unreachable=0    ...
www3.example.com    : ok=1    changed=0    unreachable=0    ...

real 0m6.171s
user 0m2.146s
sys 0m1.118s
[user@demo ~]$ time ansible-playbook speed_nofacts.yml

PLAY [Demonstrate disabling the facts gathering]****

TASK [debug] ****
ok: [www1.example.com] => {
    "msg": "gather_facts is set to False"
}
ok: [www2.example.com] => {
    "msg": "gather_facts is set to False"
}
ok: [www3.example.com] => {
    "msg": "gather_facts is set to False"
}

PLAY RECAP ****
www1.example.com    : ok=1    changed=0    unreachable=0    ...
www2.example.com    : ok=1    changed=0    unreachable=0    ...
www3.example.com    : ok=1    changed=0    unreachable=0    ...

real 0m1.336s
user 0m1.116s
sys 0m0.246s

```



Note

To get the execution time of a playbook you can also use the Ansible **timer** callback plug-in, instead of using the **time** command. Callback plug-ins will be discussed later in this section.

Playbooks often use the `ansible_facts['hostname']`, `ansible_hostname`, `ansible_facts['nodename']`, or `ansible_nodename` variables to refer to the host currently being processed. Those variables come from the facts gathering task, but you can usually replace them with the `inventory_hostname` and `inventory_hostname_short` magic variables.

You can also choose to collect facts manually by running the `setup` module as a task on selected hosts as part of a play, and those facts will be available for subsequent plays in the playbook.

Increasing Parallelism

When Ansible is running a play, it will run the first task on every host in the current batch, then run the second task on every host in the current batch, and so on until the play completes. The `forks` parameter controls how many connections Ansible can have active at the same time. By default, this is set to **5**, which means that even if there are 100 hosts to process on the current task, Ansible will communicate with them in groups of five. Once it has communicated with all 100, then Ansible will move to the next task.

By increasing the `forks` value, Ansible runs each task simultaneously on more hosts at a time, and the playbook usually completes in less time. For example, if you set `forks` to **100**, Ansible can attempt to open connections to all 100 hosts in the previous example simultaneously. This will place more load on the control node, which still needs enough time to communicate with each of the hosts.

Specify the number of forks to use in the Ansible configuration file, or by passing a `-f` option to `ansible-playbook`.

The following example shows the `forks` parameter set to **100** under the `[defaults]` section of the `ansible.cfg` configuration file.

```
[user@demo ~]$ cat ansible.cfg
[defaults]
inventory=inventory
remote_user=devops
forks=100
```

Because the `forks` value tells Ansible how many worker processes to start, a number that is too high may hurt your control node and your network. Try first with a conservative value, **20** or **50** for example, and increase that number step by step, each time monitoring the system.



Note

The `cgroup_perf_recap` callback plug-in can help you monitor the CPU and memory usage on your control node during a playbook run. Callback plug-ins will be discussed later in this section.

Avoiding Loops with the Package Manager Modules

Some modules accept a list of items to work on and do not require the use of a loop. This approach can increase efficiency, since the module will be called once rather than multiple times.

The modules for managing operating system packages work this way. The following example uses the `yum` module to install several packages in a single transaction, which is the most efficient way to install a group of packages.

```
---
- name: Install the packages on the web servers
  hosts: web_servers
  become: True
  gather_facts: False

  tasks:
    - name: Ensure the packages are installed
      yum:
        name:
          - httpd
          - mod_ssl
          - httpd-tools
          - mariadb-server
          - mariadb
          - php
          - php-mysqlnd
        state: present
```

The preceding playbook would be equivalent to running the following command from the shell prompt:

```
[root@demo ~]# yum install httpd mod_ssl httpd-tools \
> mariadb-server mariadb php php-mysqlnd
```

The following example is *not* efficient. It uses a loop to install the packages one at a time:

```
---
- name: Install the packages on the web servers
  hosts: web_servers
  become: True
  gather_facts: False

  tasks:
    - name: Ensure the packages are installed
      yum:
        name: "{{ item }}"
        state: present
      loop:
        - httpd
        - mod_ssl
        - httpd-tools
        - mariadb-server
        - mariadb
        - php
        - php-mysqlnd
```

The second example is equivalent to running multiple **yum** commands:

```
[root@demo ~]# yum install httpd
[root@demo ~]# yum install mod_ssl
[root@demo ~]# yum install httpd-tools
[root@demo ~]# yum install mariadb-server
[root@demo ~]# yum install mariadb
[root@demo ~]# yum install php
[root@demo ~]# yum install php-mysqld
```

The second example is slower and less efficient because Ansible runs the **yum** module seven times, starting a process for the module seven times, and doing dependency resolution seven times.

Not all Ansible modules accept a list for the **name** parameter. For example, the **service** module only accepts a single value for its **name** parameter, and you need a loop to operate on multiple items:

```
- name: Starting the services on the web servers
hosts: web_servers
become: True
gather_facts: False

tasks:
- name: Ensure the services are started
  service:
    name: "{{ item }}"
    state: started
    enabled: yes
  loop:
    - httpd
    - mariadb
```

Use the **ansible-doc** command to get information about what types of values different module arguments will accept:

```
[user@demo ~]$ ansible-doc yum
...output omitted...
- name
  A package name or package specifier with version, like
  `name-1.0'.
  If a previous version is specified, the task also needs to
  turn `allow_downgrade' on. See the `allow_downgrade'
  documentation for caveats with downgrading packages.
  When using state=latest, this can be ``''' which means run
  `yum -y update'.
  You can also pass a url or a local path to a rpm file (using
  state=present). To operate on several packages this can accept
  a comma separated string of packages or (as of 2.0) a list of
  packages.
  (Aliases: pkg)[Default: (null)]
...output omitted...
[user@demo ~]$ ansible-doc service
...output omitted...
= name
  Name of the service.
```

```
type: str
...output omitted...
```

Efficiently Copy Files to Managed Hosts

The **copy** module recursively copies directories to managed hosts. When the directory is large, with a lot of files, the copy can take a long time. If you run the playbook multiple times, subsequent copies will take less time because the module only copies the files that are different.

However, it is generally more efficient to use the **synchronize** module to copy large numbers of files to managed hosts. This module uses **rsync** in the background and is faster than the **copy** module in most cases.

The following playbook uses the **synchronize** module to copy the **web_content** directory to the web servers recursively.

```
---
- name: Deploy the web content on the web servers
  hosts: web_servers
  become: True
  gather_facts: False

  tasks:
    - name: Ensure web content is updated
      synchronize:
        src: web_content/
        dest: /var/www/html
```

Using Templates

The **lineinfile** module inserts or removes lines in a file, such as configuration directives in a configuration file. The following playbook example updates the Apache HTTP Server configuration file by replacing several lines.

```
---
- name: Configure the Apache HTTP Server
  hosts: web_servers
  become: True
  gather_facts: False

  tasks:

    - name: Ensure proper configuration of the Apache HTTP Server
      lineinfile:
        dest: /etc/httpd/conf/httpd.conf
        regexp: "{{ item.regexp }}"
        line: "{{ item.line }}"
        state: present
      loop:
        - regexp: '^Listen 80$'
          line: 'Listen 8181'
        - regexp: '^ServerAdmin root@localhost'
          line: 'ServerAdmin support@example.com'
```

```
- regexp: '^DocumentRoot "/var/www/html"'
  line: 'DocumentRoot "/var/www/web"'
- regexp: '^<Directory "/var/www/html">'
  line: '<Directory "/var/www/web">'
```

When used with a loop, **lineinfile** is not very efficient (and can be error-prone). In this situation, use either the **template** or the **copy** module instead.

```
---
- name: Configure the Apache HTTP Server
  hosts: web_servers
  become: True
  gather_facts: False

  tasks:
    - name: Ensure proper configuration of the Apache HTTP Server
      template:
        src: httpd.conf.j2
        dest: /etc/httpd/conf/httpd.conf
```

The **httpd.conf.j2** template file in the previous example is the customized version of the **httpd.conf** file.

Optimizing SSH connections

Establishing an SSH connection can be a slow process. When a play has many tasks and targets a large set of nodes, the overall time Ansible spends establishing these connections can significantly increase the global execution time of your playbook.

To mitigate this issue, Ansible relies on two features that SSH provides:

ControlMaster

The **ControlMaster** SSH directive allows multiple simultaneous SSH sessions with a remote host to use a single network connection. The first SSH session establishes the connection, and the additional sessions to the same host reuse that connection, bypassing the slow initial process. SSH destroys the shared connection as soon as the last session is closed.

ControlPersist

The **ControlPersist** SSH directive keeps the connection open in the background, rather than destroying the connection after the last session. This directive allows a later SSH session to reuse the connection. **ControlPersist** indicates how long SSH should keep an idle connection opened; each new session resets this idle timer.

Ansible enables the **ControlMaster** and **ControlPersist** features with the **ssh_args** directive under the **[ssh_connection]** section of the Ansible configuration file. The default value for **ssh_args** is as follows.

```
[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

With this value, Ansible keeps SSH connections open for 60 seconds after the last session completes. You can increase the value of **ControlPersist** when you have a large set of hosts, resulting in tasks taking more than 60 seconds to complete.

With the default value of 60 seconds, SSH has already closed the connection with the first host by the time the next task starts, and must re-establish the connection. You may also increase the value of the **forks** directive for tasks to complete faster, before the idle timer expires.

If your **forks** value or the **ControlPersist** setting is large, the control node may use more simultaneous connections. Make sure that the control node is configured with enough file handles available to support many active network connections.

Enabling Pipelining

To run a task on a remote node, Ansible performs several SSH operations to copy the module and all its data to the remote node and to execute the module. To increase the performance of your playbook, you can activate the pipelining feature. With pipelining, Ansible establishes fewer SSH connections.

To activate pipelining, set the **pipelining** directive to **True** in the **[ssh_connection]** section of the Ansible configuration file.

```
[ssh_connection]
  pipelining = True
```

Ansible does not use pipelining by default because the feature requires that the **requiretty** sudo option on all the remote nodes is disabled. On Red Hat Enterprise Linux 8, that sudo option is disabled by default, but it may be active on other systems..

To disable the option, use the **visudo** command to edit the **/etc/sudoers** file on your managed nodes and disable the **requiretty** flag:

```
[root@demo ~]# visudo
...output omitted...
Defaults !requiretty
...output omitted...
```

Profiling Playbook Execution with Callback plug-ins

Callback plug-ins extend Ansible by adjusting how it responds to various events. Some of these plug-ins modify the output of the command-line tools, such as the **ansible-playbook** command, to provide additional information. For example, the **timer** plug-in shows the playbook execution time in the output of the **ansible-playbook** command.



Important

Red Hat Ansible Tower logs some information about jobs (playbook runs), which it extracts from the output of the **ansible-playbook**. Because some callback plug-ins modify this output, you should use them with caution or avoid using them with Red Hat Ansible Tower.

Ansible comes with a collection of callback plug-ins that you can enable in the **ansible.cfg** file through the **callback_whitelist** directive.

```
[user@demo ~]$ cat ansible.cfg
[defaults]
inventory=inventory
remote_user=devops
callback_whitelist=timer, profile_tasks, cgroup_perf_recap
```

To list the available plug-ins, use the **ansible-doc -t callback -l** command. To access the documentation for a specific plug-in, run the **ansible-doc -t callback plug-in-name** command.

```
[user@demo ~]$ ansible-doc -t callback -l
actionable           shows only items that need attention
aws_resource_actions summarizes all "resource:actions" completed
cgroup_memory_recap  Profiles maximum memory usage of tasks and full ...
cgroup_perf_recap    Profiles system activity of tasks and full ...
context_demo         demo callback that adds play/task context
...output omitted...
[user@demo ~]$ ansible-doc -t callback cgroup_perf_recap
> CGROUP_PERF_RECAP (/usr/lib/python3.6/site-packages/ansible/plugins/...)
```

This is an ansible callback plugin utilizes cgroups to profile system activity of ansible and individual tasks, and display a recap at the end of the playbook execution

* This module is maintained by The Ansible Community
 OPTIONS (= is mandatory):
 = control_group
 Name of cgroups control group
 ...output omitted...

Profiling the Control Node CPU and Memory

The **cgroup_perf_recap** callback plug-in profiles the control node during a playbook run. At the end of the playbook execution, it displays a global summary and a summary per task. Those summaries include the CPU and memory consumption, and the maximum number of processes launched during the playbook and tasks executions.

```
...output omitted...
PLAY RECAP ****
www1.example.com      : ok=9     changed=7     unreachable=0 ...
www2.example.com      : ok=10    changed=9     unreachable=0 ...
www3.example.com      : ok=10    changed=9     unreachable=0 ...

CGROUP PERF RECAP ****
Memory Execution Maximum: 102.26MB
cpu Execution Maximum: 146.44%
pids Execution Maximum: 17.00

memory:
Gathering Facts (5254...002c): 96.85MB
firewall : Ensure Firewall Sources Configuration (5254...0010): 102.13MB
```

```

haproxy : Ensure haproxy packages are present (5254...0014): 95.81MB
haproxy : Ensure haproxy is started and enabled (5254...0015): 98.93MB
haproxy : Ensure haproxy configuration is set (5254...0016): 98.04MB
Ensure required packages are installed (5254...001e): 80.90MB
Ensure firewall allows the services (5254...001f): 83.33MB
Ensure the services are enabled (5254...0020): 85.73MB

cpu:
Gathering Facts (5254...002c): 127.22%
firewall : Ensure Firewall Sources Configuration (5254...0010): 144.22%
haproxy : Ensure haproxy packages are present (5254...0014): 136.78%
haproxy : Ensure haproxy is started and enabled (5254...0015): 139.32%
haproxy : Ensure haproxy configuration is set (5254...0016): 146.44%
Ensure required packages are installed (5254...001e): 89.34%
Ensure firewall allows the services (5254...001f): 95.06%
Ensure the services are enabled (5254...0020): 99.86%

pids:
Gathering Facts (5254...002c): 12.00
firewall : Ensure Firewall Sources Configuration (5254...0010): 14.00
haproxy : Ensure haproxy packages are present (5254...0014): 14.00
haproxy : Ensure haproxy is started and enabled (5254...0015): 14.00
haproxy : Ensure haproxy configuration is set (5254...0016): 13.00
Ensure required packages are installed (5254...001e): 14.00
Ensure firewall allows the services (5254...001f): 17.00
Ensure the services are enabled (5254...0020): 17.00

```

You can use this plug-in to identify the tasks that are consuming your control node resources. **cgroup_perf_recap** is also useful to monitor the system activity when you are tuning the **forks** parameter in the **ansible.cfg** file. This way, you can make sure that the value you select does not saturate the system.

The **cgroup_perf_recap** callback plug-in relies on the Linux *control groups (cgroup)* feature to monitor and profile the **ansible-playbook** command. On a Linux system, you can use the control groups to limit and monitor the resources, such as memory or CPU, that a group of processes can consume. To set these limits, create a new group, set limits, and add your processes to the group.

To use the **cgroup_perf_recap** callback plug-in, you must first create a dedicated control group running the **ansible-playbook** command. The plug-in uses the monitoring features of the control group to collect the performance metrics of the **ansible-playbook** process and its children.

To create the dedicated control group, use the **cgcreate** command as the **root** user.

```
[user@demo ~]$ sudo cgcreate -a user:user -t user:user \
> -g cpuacct,memory,pids:ansible_profile
```

The **-a** and **-t** options display the user and group that can access and manage the control group. Use the user and the group of the account you are using to run the **ansible-playbook** command. The **-g** option gives the name of the new control group. The example uses **ansible_profile** for the name. Control groups you create with the **cgcreate** command do not persist across reboots.

The next step is to enable the plug-in in the `ansible.cfg` file.

```
[user@demo ~]$ cat ansible.cfg
[defaults]
inventory=inventory
remote_user=devops
callback_whitelist=cgroup_perf_recap

[callback_cgroup_perf_recap]
control_group=ansible_profile
```

The `control_group` directive under the `[callback_cgroup_perf_recap]` section gives the name of your control group.

With this setup, you can now profile your playbook. To do so, run the `ansible-playbook` command in your new control group using the `cexec` command.

```
[user@demo ~]$ cexec -g cpacct,memory,pids:ansible_profile \
> ansible-playbook deploy_webservers.yml
...output omitted...
PLAY RECAP ****
www1.example.com      : ok=9      changed=7      unreachable=0 ...
www2.example.com      : ok=10     changed=9      unreachable=0 ...
www3.example.com      : ok=10     changed=9      unreachable=0 ...

CGROUP PERF RECAP ****
Memory Execution Maximum: 102.26MB
cpu Execution Maximum: 146.44%
pids Execution Maximum: 17.00
...output omitted...
```

Timing Tasks and Roles

The `timer`, `profile_tasks`, and `profile_roles` callback plug-ins are useful to identify slow tasks and roles.

The `timer` plug-in displays the duration of the playbook execution.

The `profile_tasks` adds the start time of each task and displays at the end of the playbook execution the time spent in each task, sorted in descending order.

The `profile_roles` displays at the end the time spent in each role, sorted in descending order.

To activate these plug-ins, add or update the `callback_whitelist` directive in the `ansible.cfg` file.

```
[user@demo ~]$ cat ansible.cfg
[defaults]
inventory=inventory
remote_user=devops
callback_whitelist=timer, profile_tasks, profile_roles
```

You do not have to enable all three plug-ins: select the ones that you need.

The following example shows the output of the **ansible-playbook** command when you activate the three plug-ins.

```
[user@demo ~]$ ansible-playbook deploy_webservers.yml
...output omitted...
PLAY RECAP ****
www1.example.com    : ok=9      changed=7      unreachable=0 ...
www2.example.com    : ok=10     changed=9      unreachable=0 ...
www3.example.com    : ok=10     changed=9      unreachable=0 ...

Thursday 16 May 2019  12:35:32 +0000 (0:00:06.018)  0:01:29.772 ****
==

yum ----- 30.18s
firewall ----- 19.27s
haproxy ----- 16.58s
firewalld ----- 10.43s
service ----- 10.39s
gather_facts ----- 2.85s
-----
total ----- 89.70s
Thursday 16 May 2019  12:35:32 +0000 (0:00:06.018)  0:01:29.772 ****
==

Ensure required packages are installed ----- 30.18s
firewall : reload firewalld ----- 17.70s
Ensure firewall allows the services ----- 10.43s
Ensure the services are enabled ----- 10.39s
haproxy : reload haproxy ----- 6.02s
haproxy : Ensure haproxy packages are present ----- 4.32s
haproxy : Ensure haproxy configuration is set ----- 4.15s
Gathering Facts ----- 2.85s
haproxy : Ensure haproxy is started and enabled ----- 2.09s
firewall : Ensure Firewall Sources Configuration ----- 1.57s
Playbook run took 0 days, 0 hours, 1 minutes, 29 seconds
```



References

[ssh_config\(5\)](#) and [sudoers\(5\)](#) man pages

Ansible Configuration Settings – Ansible Documentation

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

Callback Plugins – Ansible Documentation

<https://docs.ansible.com/ansible/latest/plugins/callback.html>

► Guided Exercise

Optimizing Execution for Speed

In this exercise, you will modify a playbook to optimize it for more efficient execution, analyzing the playbook's performance with callback plug-ins.

Outcomes

You should be able to:

- Activate Ansible callback plug-ins in the configuration file.
- Profile a playbook execution time.
- Optimize a playbook.

Before You Begin

Log in as the **student** user on **workstation** and run **lab task-speed start**.

```
[student@workstation ~]$ lab task-speed start
```

This script creates an Ansible project in **/home/student/D0447/labs/task-speed/**.

- 1. From a terminal, change to the **/home/student/D0447/labs/task-speed/** directory and review the **deploy_webservers.yml** playbook.
- 1.1. Change to the **/home/student/D0447/labs/task-speed/** directory.

```
[student@workstation ~]$ cd ~/D0447/labs/task-speed
[student@workstation task-speed]$
```

- 1.2. Review the contents of the **deploy_webservers.yml** playbook.

```
[student@workstation task-speed]$ cat deploy_webservers.yml
---
- name: Deploy the web servers
  hosts: web_servers
  become: True

  tasks:
    - name: Ensure required packages are installed
      yum:
        name: "{{ item }}"
        state: present
      loop:
        - httpd
        - mod_ssl
        - httpd-tools
        - mariadb-server
```

```

- mariadb
- php
- php-mysqld

- name: Ensure the services are enabled
  service:
    name: "{{ item }}"
    state: started
    enabled: True
  loop:
    - httpd
    - mariadb

- name: Ensure the web content is installed
  copy:
    src: web_content/
    dest: /var/www/html

```

The playbook installs packages, start services, and recursively copies a local directory to the managed nodes.

- ▶ 2. Activate the **timer** and the **profile_tasks** callback plug-ins, and then run the **deploy_webservers.yml** playbook.
 - 2.1. Edit the **ansible.cfg** file, and then add the **callback_whitelist** directive with the two callback plug-ins to activate.

```
[student@workstation task-speed]$ vi ansible.cfg
[defaults]
inventory=inventory.yml
remote_user=devops
callback_whitelist=timer, profile_tasks
```

- 2.2. Run the **deploy_webservers.yml** playbook, and then take note of the total elapsed time.

```
[student@workstation task-speed]$ ansible-playbook deploy_webservers.yml
...output omitted...
PLAY RECAP ****
serverb.lab.example.com : ok=4    changed=3    unreachable=0    ...
serverc.lab.example.com : ok=4    changed=3    unreachable=0    ...

Friday 17 May 2019  07:49:55 +0000 (0:00:50.723)      0:01:23.269 ****
Ensure the web content is installed ----- 50.72s
Ensure required packages are installed ----- 26.58s
Ensure the services are enabled ----- 3.93s
Gathering Facts ----- 1.98s
Playbook run took 0 days, 0 hours, 1 minutes, 23 seconds
```

The playbook and tasks execution times are probably different on your system. However, web content deployment and package installations should be the most time-consuming tasks.

- 3. Optimize the **deploy_webservers.yml** playbook. To do so, disable facts gathering, because the playbook does not use facts. Also, remove the loop in the package installation task and replace the **copy** module with the **synchronize** module.

If you make a mistake, you can restore the original file from its backup in **deploy_webservers.yml.backup**.

- 3.1. Edit the **deploy_webservers.yml** playbook. Add the **gather_facts** directive and set it to **False**. Do not close the file yet.

```
[student@workstation task-speed]$ vi deploy_webservers.yml
---
- name: Deploy the web servers
  hosts: web_servers
  become: True
  gather_facts: False

...output omitted...
```

- 3.2. Remove the loop from the package installation task. To do so, move the list of packages under the **name** variable and remove the **loop** directive.

```
...output omitted...
tasks:
  - name: Ensure required packages are installed
    yum:
      name:
        - httpd
        - mod_ssl
        - httpd-tools
        - mariadb-server
        - mariadb
        - php
        - php-mysqlnd
      state: present

...output omitted...
```

The next task uses the **service** module to start the **httpd** and **mariadb** services. This task also uses a loop. However, in contrast to the **yum** module, the **service** module does not accept a list of services in its **name** attribute. Leave this task as it is.

- 3.3. In the last task, replace the **copy** module with the **synchronize** module, which is more efficient when deploying large directories.

```
...output omitted...
- name: Ensure the web content is installed
  synchronize:
    src: web_content/
    dest: /var/www/html
```

Save and close the file when done.

- 4. Run the **deploy_webservers.yml** optimized playbook and compare the execution time with its first execution. Before proceeding, run the **clean.yml** playbook. This playbook removes the packages and the web content from the managed hosts.
- 4.1. Run the **clean.yml** playbook.

```
[student@workstation task-speed]$ ansible-playbook clean.yml
```

- 4.2. Run the **deploy_webservers.yml** playbook.

```
[student@workstation task-speed]$ ansible-playbook deploy_webservers.yml
...output omitted...
PLAY RECAP ****
serverb.lab.example.com      : ok=3    changed=3    unreachable=0    ...
serverc.lab.example.com      : ok=3    changed=3    unreachable=0    ...

Friday 17 May 2019  08:22:48 +0000 (0:00:24.010)      0:00:42.727 ****
Ensure the web content is installed ----- 24.01s
Ensure required packages are installed ----- 13.56s
Ensure the services are enabled ----- 5.09s
Playbook run took 0 days, 0 hours, 0 minutes, 42 seconds
```

Notice that the fact gathering task is absent. The playbook and tasks execution times should also be reduced.

- 5. Ansible Tower is not compatible with the **timer** and the **profile_tasks** callback plug-ins, so disable them. When done, change to your home directory.
- 5.1. Disable the **timer** and the **profile_tasks** callback plug-ins by deleting the **callback_whitelist** line in **ansible.cfg**.

```
[student@workstation task-speed]$ cat ansible.cfg
[defaults]
inventory=inventory.yml
remote_user=devops
```

- 5.2. Change to your home directory.

```
[student@workstation task-speed]$ cd
[student@workstation ~]$
```

Finish

On **workstation**, run the **lab task-speed finish** command to complete this exercise.

```
[student@workstation ~]$ lab task-speed finish
```

This concludes the guided exercise.

▶ Lab

Managing Task Execution

Performance Checklist

In this lab, you will reorganize privilege escalation, add tags and handlers to tasks, and profile playbooks to optimize them.

Outcomes

You should be able to:

- Change privilege escalation to a more secure configuration.
- Add task hooks and handlers to alter the task behavior.
- Tag tasks to control their execution.

Before You Begin

Log in as the **student** user on **workstation** and run **lab task-review start**. This script initializes the remote Git repository that you need for this lab.

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

1. Clone the Git repository `http://git.lab.example.com:8081/git/task-review.git` in the `/home/student/git-repos` directory. Change to the cloned project directory.
2. Modify the **ansible.cfg** file so that privileges do not escalate by default for each task. For the **firewall**, **haproxy**, **apache**, and **webapp** roles, add privilege escalation to any tasks or handlers that require it. If all tasks in a role require privilege escalation, put the tasks inside a block and define privilege escalation for just the block. Do not put handler tasks inside of a block.
3. Add a **copy** task to the **deploy_haproxy.yml** playbook that executes before any other task and writes the text **Playbook site.yml ready to start** to the `/tmp/site.ready` file on the servers in the **lb_servers** group.
4. Add a handler to the **haproxy** role that writes the message **Reloaded** to the `/tmp/haproxy.status` file. Notify the handler from the task that uses the **template** module in the **haproxy** role. Use **haproxy_filehandler** as the name of the handler.
5. Add the **apache_installer** tag to the **yum** task in the **apache** role.
6. Enable the **timer** and **profile_tasks** callback plug-ins for the playbook.
7. Run the **site.yml** playbook and analyze the output to find the most time expensive task.
8. Refactor the most expensive task in the **apache** role to make it more efficient. To verify the changes, run the **site.yml** playbook with an appropriate tag so that you only execute the modified task.

Evaluation

Grade your work by running the **lab task-review grade** command from your **workstation** machine.

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

Correct any reported failures.

Some grading criteria depend on the execution of handler tasks. Prior to grading your work again, execute:

```
[student@workstation task-review]$ lab task-review finish  
...output omitted...  
[student@workstation task-review]$ ansible-playbook site.yml  
...output omitted...
```

These commands ensure that handler tasks are executed again to satisfy grading criteria.

Finish

From **workstation**, run the **lab task-review finish** command to complete this lab.

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

This concludes the lab.

► Solution

Managing Task Execution

Performance Checklist

In this lab, you will reorganize privilege escalation, add tags and handlers to tasks, and profile playbooks to optimize them.

Outcomes

You should be able to:

- Change privilege escalation to a more secure configuration.
- Add task hooks and handlers to alter the task behavior.
- Tag tasks to control their execution.

Before You Begin

Log in as the **student** user on **workstation** and run **lab task-review start**. This script initializes the remote Git repository that you need for this lab.

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

1. Clone the Git repository `http://git.lab.example.com:8081/git/task-review.git` in the **/home/student/git-repos** directory. Change to the cloned project directory.
 - 1.1. From a terminal, create the directory **/home/student/git-repos** if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- 1.3. Clone the repository, and change to the cloned project directory:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/task-review.git
Cloning into 'task-review'...
remote: Enumerating objects: 62, done.
remote: Counting objects: 100% (62/62), done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 62 (delta 7), reused 0 (delta 0)
Unpacking objects: 100% (62/62), done.
[student@workstation git-repos]$ cd task-review
```

2. Modify the **ansible.cfg** file so that privileges do not escalate by default for each task. For the **firewall**, **haproxy**, **apache**, and **webapp** roles, add privilege escalation to any tasks or handlers that require it. If all tasks in a role require privilege escalation, put the tasks inside a block and define privilege escalation for just the block. Do not put handler tasks inside of a block.

- 2.1. Edit the **ansible.cfg** file.

```
[student@workstation task-review]$ vi ansible.cfg
```

- 2.2. Remove the **become=true** entry from the **privilege_escalation** block. The file should display as follows:

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

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

- 2.3. Edit the **roles/firewall/tasks/main.yml** file:

```
[student@workstation task-review]$ vi roles/firewall/tasks/main.yml
```

Add the **become: true** directive to the **firewalld** task:

```
---
# tasks file for firewall

- name: Ensure Firewall Sources Configuration
  firewalld:
    source: "{{ item.source if item.source is defined else omit }}"
    zone: "{{ item.zone if item.zone is defined else omit }}"
    permanent: true
    state: "{{ item.state | default('enabled') }}"
    service: "{{ item.service if item.service is defined else omit }}"
    immediate: true
    port: "{{ item.port if item.port is defined else omit }}"
    loop: "{{ firewall_rules }}"
    notify: reload firewalld
    become: true
```

- 2.4. Edit the **roles/firewall/handlers/main.yml** file:

```
[student@workstation task-review]$ vi roles/firewall/handlers/main.yml
```

Add the **become** directive to the task. The file should display as follows:

```
---
# handlers file for firewall
```

```
- name: reload firewalld
  service:
    name: firewalld
    state: reloaded
  become: true
```

2.5. Edit the **roles/haproxy/tasks/main.yml** file:

```
[student@workstation task-review]$ vi roles/haproxy/tasks/main.yml
```

Nest the tasks within the **block** directive and add the **become: true** directive to the tasks block. Remember to indent the tasks now inside the block. The file should display as follows:

```
---
# tasks file for haproxy

- block:
  - name: Ensure haproxy packages are present
    yum:
      name:
        - haproxy
        - socat
      state: present

  - name: Ensure haproxy is started and enabled
    service:
      name: haproxy
      state: started
      enabled: true

  - name: Ensure haproxy configuration is set
    template:
      src: haproxy.cfg.j2
      dest: /etc/haproxy/haproxy.cfg
      notify: reload haproxy
  become: true
```

2.6. Edit the **roles/haproxy/handlers/main.yml** file:

```
[student@workstation task-review]$ vi roles/haproxy/handlers/main.yml
```

Add the **become: true** directive to each task. The file should display as follows:

```
---
# handlers file for haproxy

- name: restart haproxy
  service:
    name: haproxy
    state: restarted
  become: true
```

```
- name: reload haproxy
  service:
    name: haproxy
    state: reloaded
  become: true
```

2.7. Edit the **roles/apache/tasks/main.yml** file:

```
[student@workstation task-review]$ vi roles/apache/tasks/main.yml
```

Surround the tasks in a block and add the **become: true** directive to the block. Remember to indent the tasks now inside the block. The file should display as follows:

```
---
# tasks file for apache

- block:
  - name: Ensure httpd packages are installed
    yum:
      name: "{{ item }}"
      state: present
    loop:
      - httpd
      - php
      - git
      - php-mysqld

  - name: Ensure SELinux allows httpd connections to a remote database
    seboolean:
      name: httpd_can_network_connect_db
      state: true
      persistent: true

  - name: Ensure httpd service is started and enabled
    service:
      name: httpd
      state: started
      enabled: true

  become: true
```

2.8. Edit the **roles/webapp/tasks/main.yml** file:

```
[student@workstation task-review]$ vi roles/webapp/tasks/main.yml
```

Add the **become: true** directive to the **copy** task. The file should display as follows:

```
---
# tasks file for webapp

- name: Ensure stub web content is deployed
  copy:
    content: "{{ webapp_message }}. (version {{ webapp_version}})\n"
```

```
dest: /var/www/html/index.html
become: true
```

3. Add a **copy** task to the **deploy_haproxy.yml** playbook that executes before any other task and writes the text **Playbook site.yml ready to start** to the **/tmp/site.ready** file on the servers in the **lb_servers** group.

- 3.1. Edit the file **deploy_haproxy.yml** and add a **pre_tasks** block that writes the file with the message at the beginning of the playbook.

```
[student@workstation task-review]$ vi deploy_haproxy.yml
```

The file should display as follows:

```
- name: Ensure HAProxy is deployed
  hosts: lb_servers
  force_handlers: true
  gather_facts: false

  pre_tasks:
    - name: Setting the maintenance message
      copy:
        dest: /tmp/site.ready
        content: "Playbook site.yml ready to start"

  roles:
    # The "haproxy" role has a dependency on the "firewall" role.
    # The "firewall" role requires a "firewall_rules" variable be defined.
    - role: haproxy
```

4. Add a handler to the **haproxy** role that writes the message **Reloaded** to the **/tmp/haproxy.status** file. Notify the handler from the task that uses the **template** module in the **haproxy** role. Use **haproxy_filehandler** as the name of the handler.

- 4.1. Edit the **roles/haproxy/handlers/main.yml** file and add the handler that copies the text to the file.

```
[student@workstation task-review]$ vi roles/haproxy/handlers/main.yml
```

The file should display as follows:

```
---
# handlers file for haproxy

- name: restart haproxy
  service:
    name: haproxy
    state: restarted
  become: true

- name: reload haproxy
  service:
    name: haproxy
    state: reloaded
```

```

become: true

- name: haproxy filehandler
copy:
  dest: /tmp/haproxy.status
  content: "Reloaded"

```

4.2. Edit the file **roles/haproxy/tasks/main.yml**:

```
[student@workstation task-review]$ vi roles/haproxy/tasks/main.yml
```

Call the **haproxy filehandler** handler in the **Ensure haproxy configuration is set** task. The file should look like:

```

---
# tasks file for haproxy

- block:
  - name: Ensure haproxy packages are present
    yum:
      name:
        - haproxy
        - socat
      state: present

  - name: Ensure haproxy is started and enabled
    service:
      name: haproxy
      state: started
      enabled: true

  - name: Ensure haproxy configuration is set
    template:
      src: haproxy.cfg.j2
      dest: /etc/haproxy/haproxy.cfg
    notify:
      - reload haproxy
      - haproxy filehandler
  become: true

```

5. Add the **apache_installer** tag to the **yum** task in the **apache** role.

5.1. Edit the **roles/apache/tasks/main.yml** file and add the tag **apache_installer** to the **yum** task.

```
[student@workstation task-review]$ vi roles/apache/tasks/main.yml
```

The output should display as follows:

```

---
# tasks file for apache

- block:
  - name: Ensure httpd packages are installed

```

```

yum:
  name: "{{ item }}"
  state: present
loops:
  - httpd
  - php
  - git
  - php-mysqlnd
tags:
  - apache_installer

- name: Ensure SELinux allows httpd connections to a remote database
  seboolean:
    name: httpd_can_network_connect_db
    state: true
    persistent: true

- name: Ensure httpd service is started and enabled
  service:
    name: httpd
    state: started
    enabled: true
  become: true

```

6. Enable the **timer** and **profile_tasks** callback plug-ins for the playbook.

- 6.1. Edit the configuration file **ansible.cfg** and add the plug ins to the **callback_whitelist** directive.

```
[student@workstation task-review]$ vi ansible.cfg
```

The file should display as follows:

```

[defaults]
inventory=inventory
remote_user=devops
callback_whitelist=timer,profile_tasks

[privilegeEscalation]
become_method=sudo
become_user=root
become_ask_pass=false

```

7. Run the **site.yml** playbook and analyze the output to find the most time expensive task.

- 7.1. Run the main playbook **site.yml**.

```
[student@workstation task-review]$ ansible-playbook site.yml
...output omitted...
PLAY RECAP ****
servera.lab.example.com      : ok=7    changed=6    unreachable=0    failed=0 ...
serverb.lab.example.com      : ok=6    changed=5    unreachable=0    failed=0 ...
serverc.lab.example.com      : ok=6    changed=4    unreachable=0    failed=0 ...
```

```

Wednesday 22 May 2019 18:47:32 +0000 (0:00:01.664)      0:00:52.487 *****
=====
apache : Ensure httpd packages are installed ----- 14.24s
firewall : reload firewalld ----- 11.54s
haproxy : Ensure haproxy packages are present ----- 4.87s
firewall : reload firewalld ----- 4.75s
haproxy : reload haproxy ----- 4.61s
haproxy : haproxy filehandler ----- 4.09s
firewall : Ensure Firewall Sources Configuration ----- 3.09s
haproxy : Ensure haproxy configuration is set ----- 2.80s
haproxy : Ensure haproxy is started and enabled ----- 2.45s
apache : Ensure SELinux allows httpd connections to a remote database --- 2.32s
Setting the maintenance message ----- 2.22s
apache : Ensure httpd service is started and enabled ----- 2.11s
firewall : Ensure Firewall Sources Configuration ----- 1.27s
webapp : Ensure stub web content is deployed ----- 1.23s
Playbook run took 0 days, 0 hours, 1 minutes, 1 seconds

```

The **apache** task, **Ensure httpd packages are installed**, takes most of the playbook execution time.

8. Refactor the most expensive task in the **apache** role to make it more efficient. To verify the changes, run the **site.yml** playbook with an appropriate tag so that you only execute the modified task.

8.1. Edit the file **roles/apache/tasks/main.yml**:

```
[student@workstation task-review]$ vi roles/apache/tasks/main.yml
```

Remove the loop from the **yum** task. The file should display as follows:

```

---
# tasks file for apache

- block:
  - name: Ensure httpd packages are installed
    yum:
      name:
        - httpd
        - php
        - git
        - php-mysqld
      state: present
    tags:
      - apache_installer

  - name: Ensure SELinux allows httpd connections to a remote database
    seboolean:
      name: httpd_can_network_connect_db
      state: true
      persistent: true

  - name: Ensure httpd service is started and enabled
    service:
      name: httpd

```

```
state: started  
enabled: true  
become: true
```

8.2. Execute the **site.yml** playbook using the tag **apache_installer**.

```
[student@workstation task-review]$ ansible-playbook site.yml \  
> --tags apache_installer  
...output omitted...  
PLAY RECAP ****  
serverb.lab.example.com : ok=1    changed=1    unreachable=0    failed=0 ...  
serverc.lab.example.com : ok=1    changed=1    unreachable=0    failed=0 ...  
  
Wednesday 22 May 2019 19:40:12 +0000 (0:00:07.640)          0:00:07.741 ****  
=====  
apache : Ensure httpd packages are installed ----- 7.64s  
Playbook run took 0 days, 0 hours, 0 minutes, 7 seconds
```

Evaluation

Grade your work by running the **lab task-review grade** command from your **workstation** machine.

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

Correct any reported failures.

Some grading criteria depend on the execution of handler tasks. Prior to grading your work again, execute:

```
[student@workstation task-review]$ lab task-review finish  
...output omitted...  
[student@workstation task-review]$ ansible-playbook site.yml  
...output omitted...
```

These commands ensure that handler tasks are executed again to satisfy grading criteria.

Finish

From **workstation**, run the **lab task-review finish** command to complete this lab.

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

This concludes the lab.

Summary

In this chapter, you learned:

- Privilege escalation can be configured at the play, role, block, or task level.
- Privilege escalation uses the **become**, **become_user**, **become_method**, and **become_flags** directives.
- A play is executed in the following order: **pre_tasks**, **roles**, **tasks**, and **post_tasks**.
- Handlers are always run in the order that they are defined in the play.
- Tags are used to mark certain tasks to be skipped or executed.
- Disabling facts gathering speeds up the execution of playbooks.
- The **timer**, **profile_tasks**, and **profile_roles** callback plug-ins can be used to profile playbooks.

Chapter 4

Transforming Data with Filters and Plugins

Goal

Populate, manipulate, and manage data in variables using filters and plugins.

Objectives

- Format, parse, and define the values of variables using filters.
- Populate variables with data from external sources using lookup plugins.
- Use filters and lookup plug-ins to implement iteration over complex data structures.
- Inspect, validate, and manipulate variables containing networking information with filters.

Sections

- Processing Variables using Filters (and Guided Exercise)
- Templating External Data Using Lookups (and Guided Exercise)
- Implementing Advanced Loops (and Guided Exercise)
- Working with Network Addresses Using Filters (and Guided Exercise)

Lab

Transforming Data with Filters and Plugins

Processing Variables Using Filters

Objectives

After completing this section, students should be able to format, parse, and define the values of variables using filters.

Ansible Filters

Ansible applies variable values to playbooks and templates using Jinja2 expressions. For example, the following Jinja2 expression replaces the name of the variable enclosed in double curly braces with its value:

```
 {{ variable }}
```

Jinja2 expressions also support *filters*. Filters are used to modify or process the value from the variable that is being placed in the playbook or template. Some filters are provided by the Jinja2 language, and others are included with Red Hat Ansible Engine as plug-ins. It is also possible to create custom filters, although that is beyond the scope of this course. Filters can be incredibly useful to prepare data for use in your playbook or template.

To understand filters, you must first know more about how variable values are handled by Ansible.

Variable Types

Ansible stores run-time data in variables. The YAML structure or the content of the value defines the exact type of data. Some value types include:

- *Strings* (a sequence of characters)
- *Numbers* (a numeric value)
- *Booleans* (true/false values)
- *Dates* (ISO-8601 calendar date)
- *Null* (sets the variable to undefined the variable)
- *Lists or Arrays* (a sorted collection of values)
- *Dictionaries* (a collection of key-value pairs)

Strings

A string is a sequence of characters and is the default data type in Ansible. Strings do not need to be wrapped in quotes or double quotes. Ansible trims trailing white spaces from unquoted strings.

```
my_string: Those are the contents of the string
```

YAML format allows you to define multiline strings. Use the pipe operator (|) to retain line breaks, or the greater than operator (>) to suppress them.

```
string_with_breaks: |
  This string
  has several
```

```
line breaks

string_without_breaks: >
    This string will not
    contain any line breaks.
    Separated lines are joined
    by a space character.
```

Numbers

When the variable contents conform to a number, Ansible (or YAML, to be precise) parses the string and generates a numeric value, either an **Integer** or a **Float**.

- **Integers** contain decimal characters, and are optionally preceded by the + or - signs:

```
answer: 42
```

- If a decimal point (.) is included within an integer-like value, it is parsed as a **Float**.

```
float_answer: 42.0
```

- Scientific notation can also be used to write big **Integers** or **Floats**

```
scientific_answer: 0.42e+2
```

- Hexadecimal numbers start with **0x**, followed by hexadecimal characters only. The following example value is the hexadecimal number 2A (42 in decimal):

```
hex_answer: 0x2A
```

- If you place a number in quotes, it is treated as a **String**

```
string_not_number: "20"
```

Booleans

Boolean values contain **yes**, **no**, **y**, **n**, **on**, **off**, **true**, or **false** strings. Values are not case-sensitive, but the Jinja2 documentation recommends that you use lowercase for consistency.

Date

If the string conforms to the **ISO-8601** standard, Ansible converts the string into a **date** typed value.

```
my_date_time: 2019-05-30T21:15:32.42+02:00
my_simple_date: 2019-05-30
```

Null

Use either the **null** or tilde (~) values to declare a variable as undefined.

```
my_undefined: null
```

Lists or Arrays

A list, also known as an array, is a sorted collection of values. Lists are the basic structures for data collections and loops.

Write lists as a comma-separated sequence of values, wrapped in square brackets, or one element per line, prefixed with a dash (-). The following examples are equivalent:

```
my_list: ['Douglas', 'Marvin', 'Arthur']
my_list:
- Douglas
- Marvin
- Arthur
```

Like arrays in most programming languages, you can access specific elements of a list by using an index number starting from **0**:

```
- name: Confirm that the second list element is "Marvin"
assert:
that:
- my_list[1] == 'Marvin'
```

Dictionaries

Dictionaries, also called maps or hashes in other contexts, are structures that link string keys to values for direct access. Like lists, dictionaries can be written in a single line or across multiple lines with the colon (:) notation:

```
my_dict: { Douglas: Human, Marvin: Robot, Arthur: Human }
my_dict:
Douglas: Human
Marvin: Robot
Arthur: Human
```

Access an item in a dictionary by its key, providing it immediately after the dictionary name, and wrapped in square brackets:

```
assert:
that:
- my_dict['Marvin'] == 'Robot'
```



Note

You can also access the dictionary entry `my_dict['Marvin']` with the syntax `my_dict.Marvin`. However, this dot notation is not recommended because it can collide with reserved names for attributes and methods of Python dictionaries. For more information see https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#creating-valid-variable-names.

Processing Data with Filters

Filters allow you to process the value of a variable in order to extract information, transform it, or use it to calculate a new value. To apply a filter, follow the name of the variable with the pipe (|) character and the name of the filter to apply. Some filters might require optional arguments or options in parentheses. You can chain multiple filters in a single expression.

As an example, the following expression filters the value of the variable `myname`, ensuring that the first letter of the value is capitalized by using a standard Jinja2 filter:

```
{{ myname | capitalize }}
```

This expression can also be used to convert a variable to a different type. The following example expression ensures that the result is a string, not an integer or a float:

```
{{ mynumber | string }}
```

The next example is more complex and shows a complete task. The **assert** module tests to see if expressions are true, and fails if they are not. The Jinja2 expression at the start of the test takes the list [1, 4, 2, 2], and uses the **unique** filter to remove duplicate elements, and then uses the **sort** filter to sort them. Note that this example uses a filter to manipulate hard-coded data instead of the value of a variable.

```
- name: Test to see if the assertion is true, fail if not
  assert:
    that:
      - "{{ [ 1, 4, 2, 2 ] | unique | sort }} is eq( [ 1, 2, 4 ] )"
```

The output of the **sort** filter is compared for equality to the expected list, using the **eq** Jinja2 test. As both the result and the expected value are equal, the **assert** module succeeds.



Important

Filters do not change the value stored in the variable. The Jinja2 expression processes that value and uses the result without changing the variable itself.

Overview of Selected Filters

There are many filters available, both as standard filters from Jinja2 and as additional filters provided by Red Hat Ansible Engine. This section provides an overview of a number of useful filters, but not an exhaustive list.

Links in the **References** section contain officially available filters from Ansible and Jinja2. In particular, the Jinja2 filters documented at <http://jinja.pocoo.org/docs/2.10/templates/#builtin-filters> provide many useful utility functions.

Checking if a Variable is Defined

The first two Ansible-specific filters act on whether the input is defined. These filters are useful in roles to make sure that variables have reasonable values.

mandatory

Fails and aborts the Ansible Playbook if the variable is not defined with a value.

```
{{ my_value | mandatory }}
```

default

If the variable is not defined with a value, then this filter will set it to the value specified in parentheses. If the second parameter in the parentheses is **True**, then the filter will also set the variable to the default value, if the initial value of the variable is an empty string or the Boolean value **False**.

```
{{ my_value | default(my_default, True) }}
```

The **default** filter can also take the special value **omit**, which causes the value to remain undefined if it had no value initially. If the variable already has a value, **omit** will not change the value.

The following task to ensure the **jonfoo** user exists is an example of using the **default(omit)** filter. If the variable **supplementary_groups['jonfoo']** is already defined, then the task will make sure that the user is a member of those groups. If it is not already defined, then the **groups** parameter for the **user** module will not be set in this task.

```
- name: Ensure user jonfoo exists.
  user:
    name: jonfoo
    groups: "{{ supplementary_groups['jonfoo'] | default(omit) }}"
```

Performing Mathematical Calculations

Jinja2 provides a number of mathematical filters that can operate on a number. You can also perform some basic mathematical calculations on numbers:

Arithmetic operations

Operator	Purpose
+	Add two numbers
-	Subtract two numbers
/	Perform floating point division
//	Perform integer division
%	Get the remainder of integer division
*	Multiply two numbers
**	Raise the left number to the power of the right number

In some cases, you might first need to convert the value to an integer with the **int** filter, or to a float with the **float** filter. For example, the following Jinja2 expression adds one to the current hour number, which is collected as a fact and stored as a string, not an integer:

```
{{ ( ansible_facts['date_time']['hour'] | int ) + 1 }}
```

There are also a number of filters that can perform a mathematical operation on a number: **log**, **pow**, **root**, **abs**, and **round** are examples.

```
{{ 1764 | root }}
```

Manipulating Lists

There are many filters that you can use to analyze and manipulate lists.

If the list consists of numbers, you can use **max**, **min** or **sum** to find the largest number, the smallest number, and sum of all list items.

```
{{ [2, 4, 6, 8, 10, 12] | sum }}
```

Extracting list elements

Information about the contents of the list can be obtained, such as the **first** or **last** elements, or the **length** of the list:

```
- name: All three of these assertions are true
assert:
  that:
    - "{{ [ 2, 4, 6, 8, 10, 12 ] | length }}" is eq( 6 )
    - "{{ [ 2, 4, 6, 8, 10, 12 ] | first }}" is eq( 2 )
    - "{{ [ 2, 4, 6, 8, 10, 12 ] | last }}" is eq( 12 )
```

The **random** filter returns a random element from the list:

```
{{ ['Douglas', 'Marvin', 'Arthur'] | random }}
```

Modifying the order of list elements

A list can be reordered in several ways. The **sort** filter sorts the list in the natural order of its elements. The **reverse** filter returns a list where the order is the opposite of the original order. The **shuffle** filter returns a list with the same elements, but in a random order.

```
- name: reversing and sorting lists
assert:
  that:
    - "{{ [ 2, 4, 6, 8, 10 ] | reverse | list }}" is eq( [ 10, 8, 6, 4, 2 ] )
    - "{{ [ 4, 8, 10, 6, 2 ] | sort | list }}" is eq( [ 2, 4, 6, 8, 10 ] )"
```

Merging lists

Some times it is useful to merge several lists into a single one to simplify iteration. The **flatten** filter recursively takes any inner list in the input list value, and adds the inner values out to the outer list.

```
- name: Flatten turns nested lists on the left to list on the right
assert:
  that:
    - "{{ [ 2, [4, [6, 8]], 10 ] | flatten }}" is eq( [ 2, 4, 6, 8, 10 ] )"
```

Use **flatten** to merge lists values that come from iterating a parent list.

Operating on lists as sets

Make sure that a list has no duplicate elements with the **unique** filter. This can be useful if you are operating on a list of facts that you have collected, such as usernames or host names that might have duplicate entries.

```
- name: 'unique' leaves unique elements
assert:
that:
- "{{ [ 1, 1, 2, 2, 3, 4, 4 ] | unique | list }} is eq( [ 1, 2, 3, 4 ] )"
```

If two lists have no duplicate elements, then you can use set theory operations on them.

- The **union** filter returns a set with elements from both input sets.
- The **intersect** filter returns a set with elements common to both sets.
- The **difference** filter returns a set with elements from first set that are not present in the second set.

```
- name: 'difference' provides elements not in specified set
assert:
that:
- "{{ [ 2, 4, 6, 8, 10 ] | difference([2, 4, 6, 16]) }} is eq( [8, 10] )"
```

Use the **symmetric_difference** filter to get items to address when operating with a set.

Manipulating Dictionaries

Unlike lists, dictionaries are not ordered in any way. They are just a collection of key-value pairs. But you can use filters to construct dictionaries, and you can convert them into lists or lists into dictionaries.

Joining dictionaries

Two dictionaries can be joined by the **combine** filter. Entries from the second dictionary have higher priority than entries from the first dictionary, as seen in the following task:

```
- name: 'combine' combines two dictionaries into one
vars:
expected:
A: 1
B: 4
C: 5
assert:
that:
- "{{ {'A':1,'B':2} | combine({'B':4,'C':5}) }} is eq( expected )"
```

Reshaping dictionaries

A dictionary can be reshaped to a list of items with the **dict2items** filter, and a list of item can be reshaped back to a dictionary with the **items2dict** filter:

```
- name: converting between dictionaries and lists
vars:
characters_dict:
  Douglas: Human
  Marvin: Robot
  Arthur: Human
characters_items:
- key: Douglas
  value: Human
- key: Marvin
  value: Robot
```

```

- key: Arthur
  value: Human
assert:
that:
- "{{ characters_dict | dict2items }}" is eq( characters_items )
- "{{ characters_items | items2dict }}" is eq( characters_dict )

```

Hashing, Encoding, and Manipulating Strings

A number of filters are available to manipulate the text of a value. You can take various checksums, create password hashes, and convert text to and from Base64 encoding, as used by a number of applications.

Hashing strings and passwords

The **hash** filter returns the hash value of the input string, using the provided hashing algorithm:

```

- name: the string's SHA-1 hash
vars:
  expected: '8bae3f7d0a461488ced07b3e10ab80d018eb1d8c'
assert:
that:
- "'{{ 'Arthur' | hash('sha1') }}'" is eq( expected )

```

Use the **password_hash** filter to generate password hashes:

```
{{ 'secret_password' | password_hash('sha512') }}
```

Encoding strings

Binary data can be translated to **base64** by the **b64encode** filter, and translated back to binary format by the **b64decode** filter:

```

- name: Base64 encoding and decoding of values
assert:
that:
- "'{{ 'â€œú' | b64encode }}'" is eq( 'w6LDic0vw7TDug==' )
- "'{{ 'w6LDic0vw7TDug==' | b64decode }}'" is eq( 'â€œú' )

```

Before sending strings to the underlying shell, and to avoid parsing or code injection issues, it is a good practice to sanitize the string by using the **quote** filter:

```

- name: Put quotes around 'my_string'
shell: echo {{ my_string | quote }}

```

Formatting Text

Use the **lower**, **upper**, or **capitalize** filters to enforce the case of an input string:

```
- name: Change case of characters
assert:
that:
- "'{{ 'Marvin' | lower }}' is eq( 'marvin' )"
- "'{{ 'Marvin' | upper }}' is eq( 'MARVIN' )"
- "'{{ 'marvin' | capitalize }}' is eq( 'Marvin' )"
```

Replacing text

The **replace** filter is useful when you need to replace all occurrences of a substring inside the input string:

```
- name: Replace 'ar' with asterisks
assert:
that:
- "'{{ 'marvin, arthur' | replace('ar','**') }}' is eq( 'm**vin, **thur' )"
```

Much more complex searches and replacements are available by using regular expressions and the **regex_search** and **regex_replace** filters.

```
- name: Test results of regex search and search-and-replace
assert:
that:
- "'{{ 'marvin, arthur' | regex_search('ar\S*r') }}' is eq( 'arthur' )"
- "'{{ 'arthur up' | regex_replace('ar(\S*)r','\\1mb') }}' is eq( 'thumb
up' )"
```

Manipulating JSON Data

Many data structures used by Ansible are in JSON format. JSON and YAML notation are closely related, and Ansible data structures can be processed as JSON. Likewise, many APIs that Ansible Playbooks might interact with consume or provide information in JSON format. Because this format is widely used, JSON filters are particularly useful.

JSON queries

Use the **json_query** filter to extract information from Ansible data structures.

```
- name: Get the 'name' elements from the list of dictionaries in 'hosts'
vars:
hosts:
- name: bastion
  ip:
    - 172.25.250.254
    - 172.25.252.1
- name: classroom
  ip:
    - 172.25.252.254
assert:
that:
- "{{ hosts | json_query('[*].name') }} is eq( ['bastion','classroom'] )"
```

Parsing and encoding data structures

Transforming data structures to and from text is useful for debugging and communication. Data structures serialize to JSON or YAML format with the `to_json` and `to_yaml` filters. Use `to_nice_json` and `to_nice_yaml` filters to obtain a formatted human-readable output.

```
- name: Convert between JSON and YAML format
vars:
  hosts:
    - name: bastion
      ip:
        - 172.25.250.254
        - 172.25.252.1
  hosts_json: '[{"name": "bastion", "ip": ["172.25.250.254", "172.25.252.1"]}]'
assert:
  that:
    - "'{{ hosts | to_json }}' is eq( hosts_json )"
```

In the sections and chapters that follow, other filters are introduced that are suited to specific scenarios. Review the official Ansible and Jinja2 documentation to discover more useful filters to meet your needs.



References

Filters

https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html
(Ansible documentation)

Template Designer Documentation: Filters

<http://jinja.pocoo.org/docs/2.10/templates/#builtin-filters>
(Jinja2 documentation)

► Guided Exercise

Processing Variables using Filters

In this exercise, you will add filters to tasks in two different roles to process variables.

Outcomes

You should be able to use filters to process and manipulate variables in a playbook or role.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab data-filters start
```

This command creates a git repository at `http://git.lab.example.com:8081/git/data-filters.git` that contains a partially complete playbook project, which you finish during this guided exercise.

In this exercise, you deploy a HAProxy load balancer on **servera** to distribute the incoming web requests between **serverb** and **serverc**. On those two back-end servers, you deploy Apache HTTP Server and some initial content.

- ▶ 1. Change to the **/home/student/git-repos** directory and clone the project repository. In this exercise, you do not make changes to the **deploy_haproxy.yml** playbook or the **haproxy** role. Execute the **deploy_haproxy.yml** playbook to deploy the load balancer.
 - 1.1. Create the **/home/student/git-repos** directory if it does not exist, and then change to this directory.

```
[student@workstation ~]$ cd /home/student/git-repos
```

- 1.2. Clone the repository, and change to the project root directory:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/data-filters.git
[student@workstation git-repos]$ cd data-filters
```

- 1.3. Execute the **deploy_haproxy.yml** playbook to deploy the load balancer. Because you have not deployed the web servers, requests to **servera** result in a **503** HTTP status code.

```
[student@workstation data-filters]$ ansible-playbook deploy_haproxy.yml
```

```
PLAY [Ensure HAProxy is deployed] ****
...output omitted...
PLAY RECAP ****
loadbalancer_01 : ok=7    changed=6    unreachable=0    ...
```

```
[student@workstation data-filters]$ curl servera
<html><body><h1>503 Service Unavailable</h1>
...output omitted...
```

- 2. Carefully review the task list and variables for the **apache** role.

2.1. Review the task list file for the **apache** role, **roles/apache/tasks/main.yml**.

```
[student@workstation data-filters]$ cat roles/apache/tasks/main.yml
...output omitted...
- name: Calculate the package list
  set_fact:
    # TODO: Combine the apache_base_packages and
    # apache_optional_packages variables into one list.
    apache_package_list: "{{ apache_base_packages }}"

- name: Ensure httpd packages are installed
  yum:
    name: "{{ apache_package_list }}"
    state: present
    # TODO: omit the 'enablerepo' directive
    # below if the apache_enablerepos_list is empty;
    # otherwise use the list as the value for the
    # 'enablerepo' directive.
    #enablerepo: "{{ apache_enablerepos_list }}"
...output omitted...
```

The **apache_package_list** should be a combined list of both base and optional packages to install the **httpd** service. You edit and correct the definition of this variable in a later step.

The role's **apache_base_packages** variable is defined as a list containing one package, **httpd**. To prevent host group variables from overriding this value, the variable is defined in the **vars/main.yml** file:

```
apache_base_packages:
  - httpd
```

The **apache_optional_packages** variable is defined as an empty list in the role's **defaults/main.yml** file:

```
apache_optional_packages: []
```

The **apache_enablerepos_list** variable contains a list of YUM repository IDs. Any repository ID in this list is temporarily enabled to install any packages.

The default value is an empty list, as defined in the **roles/apache/defaults/main.yml** file:

```
apache_enablerepos_list: []
```

- 3. Correct the Jinja2 expression that defines the **apache_package_list** variable in the **apache** role's first task. Remove the **TODO** comment section and save the role's task file.

Define the **apache_optional_packages** variable for the **web_servers** host group to contain the list of values: **git**, **php**, and **php-mysqld**.

- 3.1. Edit the Jinja2 expression in the **apache** role's first task that defines the **apache_package_list** variable. Add the **union** filter to create a single list from **apache_base_packages** and **apache_optional_packages** list.

When you finish, remove the comments from this task. The first task of the **roles/apache/tasks/main.yml** file is the following:

```
- name: Determine the package list
  set_fact:
    apache_package_list: "{{ apache_base_packages |
      union(apache_optional_packages) }}"
```



Warning

The **apache_package_list** variable definition is a single line of text. Do not split Jinja2 filter expressions over multiple lines, or your playbook will fail.

Save the file.

- 3.2. Define the **apache_optional_packages** variable in a new file, **group_vars/web_servers/apache.yml**. The variable defines a list of three packages: **git**, **php**, and **php-mysqld**. This overrides the role's default value for this variable.

The content of the **group_vars/web_servers/apache.yml** file follows:

```
apache_optional_packages:
  - git
  - php
  - php-mysqld
```

Save the file.

- ▶ 4. Remove comments from the **enablerepo** directive in the **apache** role's second task. Edit the directive's Jinja2 expression to use the **default** filter to omit this directive if the variable evaluates to a Boolean value of **False**. Remove the **TODO** comment section for the second task, and save the task file. The content of the second task displays as follows:

```
- name: Ensure httpd packages are installed
  yum:
    name: "{{ apache_package_list }}"
    state: present
    enablerepo: "{{ apache_enablerepos_list | default(omit, true) }}"
```

- ▶ 5. Execute the **deploy_apache.yml** playbook with the **-v** option. Verify that optional packages are present on the web servers.

```
[student@workstation data-filters]$ ansible-playbook deploy_apache.yml -v
...output omitted...
TASK [apache : Calculate the package list] ****
```

```

ok: [webserver_01] => {"ansible_facts": {"apache_package_list": ["httpd", "git", "php", "php-mysqlnd"]}, "changed": false}
...output omitted...

TASK [apache : Ensure httpd packages are installed] ****
changed: [webserver_01] => {"changed": true, "msg": "", "rc": 0, "results": ["Installed: httpd", "Installed: git", "Installed: php", "Installed: php-mysqlnd", "..."]}
...output omitted...

```

- ▶ 6. Review the task list and variable definitions for the **webapp** role. The **webapp** role ensures that the correct web application content exists on each host. When correctly implemented, the role removes any content from the root web directory that does not belong to the web application.
- Edit the three tasks in the **webapp** role that have a **TODO** comment. Use filters to implement the functionality indicated in each comment.

6.1. Review the tasks in the **roles/webapp/tasks/main.yml** file:

```

---
# tasks file for webapp

- name: Ensure stub web content is deployed
  copy:
    content: "{{ webapp_message }} (version {{ webapp_version }})\n"
    dest: "{{ webapp_content_root }}/index.html"

- name: Find deployed webapp files
  find:
    paths: "{{ webapp_content_root }}"
    recurse: yes
  register: webapp_find_files

- name: Compute webapp file list
  set_fact:
    # TODO: Use the map filter to extract
    # the 'path' attribute of each entry
    # in the 'webapp_find_files'
    # variable 'files' list.
    webapp_deployed_files: []

- name: Compute relative webapp file list
  set_fact:
    # TODO: Use the 'map' filter, along with
    # the 'relpath' filter, to create the
    # 'webapp_rel_deployed_files' variable
    # from the 'webapp_deployed_files' variable.
    #
    # Files in the 'webapp_rel_deployed_files'
    # variable should have a path relative to
    # the 'webapp_content_root' variable.
    webapp_rel_deployed_files: []

- name: Remove Extraneous Files

```

```

file:
  path: "{{ webapp_content_root_dir }}/{{ item }}"
  state: absent
# TODO: Loop over a list of files
# that are in the 'webapp_rel_deployed_files'
# list, but not in the 'webapp_file_list' list.
# Use the difference filter.
loop: []

```

The `webapp_file_list` variable is defined in the `roles/webapp/vars/main.yml` file. This variable defines a file manifest for the web application. For this version of the web application, the only file in the application is `index.html`.

The `webapp_content_root_dir` variable defines the directory location of web application content on each web server. The default value is `/var/www/html`.

- 6.2. Replace the empty list for the `webapp_deployed_files` variable in the third task of the `webapp` role with a Jinja2 expression. Start with the `webapp_find_files['files']` variable and apply the `map` filter, followed by the `list` filter. Provide the `map` filter with an argument of `attribute='path'` to retrieve the `path` attribute from each entry in the list.

After you remove the **TODO** comments, the third task displays as follows:

```

- name: Compute the webapp file list
  set_fact:
    webapp_deployed_files: "{{ webapp_find_files['files'] | map(attribute='path') | list }}"

```



Warning

The `webapp_deployed_files` variable definition is a single line of text. Do not split Ninja2 filter expressions over multiple lines, or your playbook will fail.

- 6.3. Replace the empty list for the `webapp_rel_deployed_files` variable in the fourth task of `webapp` role with a Jinja2 expression. Start with the `webapp_deployed_files` variable and apply the `map` filter, followed by the `list` filter.

The first argument to the `map` function is the '`relpath`' string which executes the `relpath` function on each item of the `webapp_deployed_files` list. The second argument to the `map` function is the `webapp_content_root_dir` variable. This variable is passed as an argument to the `relpath` function.

After you remove the **TODO** comments, the fourth task displays as follows:

```

- name: Compute the relative webapp file list
  set_fact:
    webapp_rel_deployed_files: "{{ webapp_deployed_files | map('relpath', webapp_content_root_dir) | list }}"

```



Warning

The `webapp_rel_deployed_files` variable definition is a single line of text. Do not split Ninja2 filter expressions over multiple lines, or your playbook will fail.

- 6.4. Replace the empty loop list in the `webapp` role's fifth task with a Jinja2 expression. Start with the `webapp_rel_deployed_files` variable and apply the `difference` filter. Provide the `webapp_file_list` variable as an argument to the `difference` filter.

After you remove the `TODO` comments, the fifth task displays as follows:

```
- name: Remove Extraneous Files
  file:
    path: "{{ webapp_content_root_dir }}/{{ item }}"
    state: absent
  loop: "{{ webapp_rel_deployed_files | difference(webapp_file_list) }}"
```

Save all of the changes to the `roles/webapp/tasks/main.yml` file.

- ▶ 7. Execute the `deploy_webapp.yml` playbook with the `-v` option. Verify that playbook identifies a non-application file on `webserver_01` and removes it.

```
[student@workstation data-filters]$ ansible-playbook deploy_webapp.yml -v
...output omitted...
TASK [webapp : Compute the webapp file list] ****
ok: [webserver_01] => {"ansible_facts": {"webapp_deployed_files": ["/var/www/html/test.html", "/var/www/html/index.html"]}, "changed": false}
ok: [webserver_02] => {"ansible_facts": {"webapp_deployed_files": ["/var/www/html/index.html"]}, "changed": false}

TASK [webapp : Compute the relative webapp file list] ****
ok: [webserver_01] => {"ansible_facts": {"webapp_rel_deployed_files": ["test.html", "index.html"]}, "changed": false}
ok: [webserver_02] => {"ansible_facts": {"webapp_rel_deployed_files": ["index.html"]}, "changed": false}

TASK [webapp : Remove Extraneous Files] ****
changed: [webserver_01] => (item=test.html) => {"ansible_loop_var": "item",
  "changed": true, "item": "test.html", "path": "/var/www/html/test.html", "state": "absent"}

PLAY RECAP ****
webserver_01    : ok=5   changed=2   unreachable=0   failed=0      skipped=0   ...
webserver_02    : ok=4   changed=1   unreachable=0   failed=0      skipped=1   ...
```

The `webserver_01` host initially has two files deployed in the `/var/www/html` directory: `test.html` and `index.html`. Someone with access to the host may have installed a temporary test page on the web server.

The playbook removes any web server file from the web content root directory that is not part of the actual web application.

Finish

On `workstation`, run the `lab data-filters finish` script to complete this guided exercise.

```
[student@workstation data-filters]$ lab data-filters finish
```

This concludes the guided exercise.

Templating External Data using Lookups

Objectives

After completing this section, students should be able to populate variables with data from external sources using lookup plug-ins.

Lookup Plug-ins

A *lookup plug-in* is an Ansible extension to the Jinja2 templating language. These plug-ins enable Ansible to use data from external sources, such as files and the shell environment.

Calling Lookup Plug-ins

You can call lookup plug-ins with one of two Jinja2 template functions, **lookup** or **query**. Both methods have a syntax that is very similar to filters. Specify the name of the function, and in parentheses the name of the lookup plug-in you want to call and any arguments that plug-in needs.

For example, the following variable definition uses the **file** lookup plug-in to put the contents of the file **/etc/hosts** into the Ansible variable **myhosts**:

```
vars:  
  hosts: "{{ lookup('file', '/etc/hosts') }}"
```

You can include more than one file name to the **file** plug-in. When called with the **lookup** function, each file's contents will be separated by a comma in the templated value.

```
vars:  
  hosts: "{{ lookup('file', '/etc/hosts', '/etc/issue') }}"
```

The Jinja2 template expression above results in the following structure (line-wrapped):

```
hosts: "127.0.0.1    localhost localhost.localdomain localhost4  
localhost4.localdomain4\n::1        localhost localhost.localdomain localhost6  
localhost6.localdomain6\n\nn,\\$\\nKernel \\\\r on an \\\\m (\\\\l)"
```

In Ansible 2.5 and later, the **query** function can be used instead of **lookup** to call lookup plug-ins. The difference between the two is that instead of the values returned being comma-separated, **query** always returns a list, which can be easier to parse and work with.

The preceding example could be called like this:

```
vars:  
  hosts: "{{ query('file', '/etc/hosts', '/etc/issue') }}"
```

The **query** call will return this data structure (line-wrapped):

```

hosts:
  - "127.0.0.1  localhost localhost.localdomain localhost4
localhost4.localdomain4\n::1          localhost localhost.localdomain localhost6
localhost6.localdomain6\n\n"
  - "\\\$\\nKernel \\\\r on an \\\\m (\\\\l)"

```

Selecting Lookup Plug-ins

There are dozens of plug-ins available in the default Ansible installation. Use the command `ansible-doc -t lookup -l` to obtain the complete list of available lookup plug-ins. For details about the purpose of specific plug-ins and how to use them, run the `ansible-doc -t lookup PLUGIN_NAME` command.

The following lines introduce some of the most useful plug-ins:

Reading the Contents of Files

The **file** plug-in allows Ansible to load the contents of a local file into a variable. If you provide a relative path, the plug-in looks for files in your playbook's **files** directory.

The following example reads the contents of the user's public key file and uses the **authorized_key** module to add the authorized key to the managed host. The example uses a loop and the **+** operator to append strings in the template in order to lookup the **files/fred.key.pub** and **files/naoko.key.pub** files.

```

- name: Add authorized keys
  hosts: all
  vars:
    users:
      - fred
      - naoko
  tasks:
    - name: Add authorized keys
      authorized_key:
        user: "{{ item }}"
        key: "{{ lookup('file', item + '.key.pub') }}"
      loop: "{{ users }}"

```

A useful trick with the **file** plug-in is that if the file is in YAML or JSON format, you can parse it into properly structured data with the **from_yaml** or **from_json** filters.

```
my_yaml: "{{ lookup('file', '/path/to/my.yaml') | from_yaml }}"
```



Note

The **file** plug-in reads files that are on the *control node*, not the managed host.

Applying Data with a Template

Like the **file** plug-in, the **template** plug-in returns the contents of files. The difference is that the **template** plug-in expects the file contents to be a Jinja2 template, and evaluates that

template before applying the contents. If you pass a relative path to the template file, the plug-in will look for it in your playbook's **templates** directory.

```
{{ lookup('template', 'my.template.j2') }}
```

The previous example will process the **templates/my.template.j2** template in your playbook's directory. As an example, assume the content of the template is the following:

```
Hello {{ name }}.
```

The following task will display "**Hello class.**".

```
- name: Print "Hello class."
  vars:
    name: class
  debug:
    msg: "{{ lookup('template', 'my.template.j2') }}"
```

The **template** plug-in allows some extra parameters, like defining the start and end marking sequences. In case the output string is a YAML value, the **convert_data** option parses the string to provide structured data.



Important

Do not confuse the **template** lookup plug-in with the **template** module.

Reading an Environment Variable on the Control Node

The **env** plug-in queries environment variables from the *control node*. It is useful when the controller host is a containerized application, and configuration maps and secrets are injected into the host by a container management application, such as Kubernetes or Red Hat OpenShift.

```
{{ lookup('env', 'MY_PASSWORD') }}
```

Reading Command Output on the Control Node

The **pipe** and **lines** plug-ins both run a command on the Ansible control node and return the output. While the **pipe** plug-in returns the raw output generated by the command, the **lines** plug-in splits the output of that command into lines.

For example, assume that you have the following Jinja2 expression:

```
{{ query('pipe', 'ls files') }}
```

This expression will return the raw output of the **ls** command as a string. If you use the **lines** plug-in, then the expression is as follows:

```
{{ query('lines', 'ls files') }}
```

This command results in a list with each line of output returned by **ls** as a list item.

One interesting application of this functionality is to get the first line (or any specific line) of output from a set of commands:

```
- name: Prints the first line of some files
  debug:
    msg: "{{ item[0] }}"
  loop:
    - "{{ query('lines', 'cat files/my.file') }}"
    - "{{ query('lines', 'cat files/my.file.2') }}
```

Note that this example may not be the most efficient way to do this particular task, given the existence of the **head** command.

Getting Content from a URL

Similar to the way the **file** plug-in gets the contents of a file, the **url** plug-in gets content from a **URL**.

```
{{ lookup('url', 'https://my.site.com/my.file') }}
```

Many options are available for controlling authentication, selecting web proxies, or splitting the content returned into lines.

However, one advantage of using the **URL** plug-in is that you can use the data returned as values in your variables, possibly processing it first with a filter.

Getting Information from the Kubernetes API

The **k8s** plug-in provides full access to the Kubernetes API through the *openshift* Python module. To fetch a Kubernetes object, you must provide the object type using the **kind** option. Providing additional object details, such as the **namespace** option or **label_selector** option, helps to filter the results:

```
{{ lookup('k8s', kind='Deployment', namespace='ns', resource_name='my_res') }}
```



Note

Use **openshift** as an alias for the **k8s** plug-in when the Ansible Playbook manages Red Hat OpenShift Container Platform instances.

Be aware that the **k8s** plug-in is a *lookup* plug-in. Its primary purpose is to extract information from the Kubernetes cluster, not to update it. Use the **k8s module** to manage a Kubernetes cluster.



Note

Custom plug-ins can be made available to playbooks by dropping the appropriate Python script in a **lookup_plugins** directory next to the Ansible Playbook file. Plug-in development is beyond the scope of this course.

Handling Lookup Error

Most Ansible plug-ins are designed to abort the Ansible Playbook on a failure. However, the **lookup** function delegates execution to other plug-ins that may not need to abort the Ansible Playbook on failure. For example, the **file** plug-in may not need to abort the Ansible Playbook if the file is not found, but might need to recover by creating the missing file.

To adapt to different plug-in needs, the **lookup** plug-in accepts the **errors** parameter.

```
 {{ lookup('file', 'my.file', errors='warn') | default("Default file content") }}
```

The default value for the **errors** option is **strict**. This means that the **lookup** plug-in raises a fatal error if the underlying script fails. If the **errors** option has the value **warn**, the **lookup** plug-in logs a warning when the underlying script fails and returns an empty string (or an empty list). If the **errors** option has the value **ignore**, the **lookup** plug-in silently ignores the error and returns an empty string or list.



References

Ansible Documentation for Lookup Plugins.

<https://docs.ansible.com/ansible/latest/plugins/lookup.html>

► Guided Exercise

Templating External Data using Lookups

In this exercise, you will leverage using different **lookup** plug-ins to obtain and manipulate data from different sources.

Outcomes

You should be able to identify different sources of information in Ansible Playbooks and come up with existing **lookup** plug-ins that may retrieve and use that information.

Before You Begin

Log in as the **student** user on **workstation** and run **lab data-lookups start**. This setup script installs the needed Python requirements and gathers the lab exercise files.

```
[student@workstation ~]$ lab data-lookups start
```

- ▶ 1. This guided exercise is composed of two parts, using different directories. In the first section, you will create an Ansible Playbook populating all users defined in a plain text file to a managed host.
 - 1.1. Navigate to the **~/D0447/labs/data-lookups/users** directory, and use your preferred editor to open the **site.yml** file.

```
[student@workstation ~]$ cd ~/D0447/labs/data-lookups/users  
[student@workstation users]$ vim site.yml
```

- 1.2. Reading the contents of a file can be performed many ways, but here you will use the **lines** lookup plug-in.

```
"{{ query('lines', 'cat users.txt') }}"
```

This plug-in reads the output of **cat users.txt** as individual lines. The **query** function breaks each line into a list item.

The Ansible task needs to loop through the users in this simple list. You can use **loop** to do this:

```
loop: "{{ query('lines', 'cat users.txt') }}"
```

The file should display as follows:

```

- name: Populate users from a file
  hosts: all
  gather_facts: no
  tasks:
    - name: Create remote user
      debug:
        msg: "To be done"
      loop: "{{ query('lines','cat users.txt') }}"

```

- 1.3. When creating users, you will also create a random password. The **password lookup** plug-in generates random passwords and optionally stores the passwords in a local file.

The previous step added a loop on users, so we can use the same **item** to generate the associated file for each user.

```
"{{ lookup('password', 'credentials/' + item + ' length=9') }}"
```

Store the password into a task variable for ease of usage. The **site.yml** file should display as follows. Be careful to include the space after the first single quote in '**length=9**'.

```

- name: Populate users from file
  hosts: all
  gather_facts: no
  tasks:
    - name: Create remote user
      vars:
        password: "{{ lookup('password', 'credentials/' + item + ' length=9') }}"
      debug:
        msg: "To be done"
      loop: "{{ query('lines','cat users.txt') }}"

```

- 1.4. In the **site.yml** playbook, replace the **debug** module with the **user** module, configured to populate the users.

Note that the password just created should be hashed before being used. The **update_password: on_create** option will only set the password if the user does not yet exist. If the user already exists but the password is different, the password will not be changed.

```

- name: Populate users from file
  hosts: all
  gather_facts: no
  tasks:
    - name: Create remote user
      vars:
        password: "{{ lookup('password', 'credentials/' + item + ' length=9') }}"
      user:
        name: "{{ item }}"
        password: "{{ password | password_hash('sha512') }}"
        update_password: on_create
      loop: "{{ query('lines','cat users.txt') }}"

```

15. Execute the playbook, and verify that it creates the users:

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

PLAY [Populate users from file] ****
TASK [Create remote user] ****
changed: [serverf.lab.example.com] => (item=jonfoo)
changed: [serverf.lab.example.com] => (item=janebar)
changed: [serverf.lab.example.com] => (item=philbaz)

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

[student@workstation users]$ ssh jonfoo@serverf "cat /etc/passwd"
...output omitted...
jonfoo:x:1002:1002::/home/jonfoo:/bin/bash
janebar:x:1003:1003::/home/janebar:/bin/bash
philbaz:x:1004:1004::/home/philbaz:/bin/bash
...output omitted...
```

Note we are using the **jonfoo** user to log into the **serverf** server, demonstrating the user exists and is available.

16. Execute the **clean.yml** to remove the users from the remote host and clean up the server:

```
[student@workstation users]$ ansible-playbook clean.yml

PLAY [Remove users listed in file] ****
TASK [Remove remote users] ****
ok: [serverf.lab.example.com] => (item=jonfoo)
ok: [serverf.lab.example.com] => (item=janebar)
ok: [serverf.lab.example.com] => (item=philbaz)

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

[student@workstation users]$ ssh serverf "grep jonfoo /etc/passwd"; echo $?
1
[student@workstation users]$
```

The output indicates the pattern was not found in the **/etc/passwd** file, confirming that the user **jonfoo** is no longer available.

- 2. In the second part of this guided exercise, you will use **lookup** plug-ins to internationalize the static web application deployed in previous chapters. Each server provides a localized version of the application, based on its own configured locale. In case no locale is configured, the server uses the locale provided by the controller.

The static application contains the version of the project used for deployment, also known as the *Infrastructure-as-Code version (IaC version)*. For the sake of simplicity, use the Git commit id for the project as the IaC version of the Ansible project.

- 2.1. Navigate into the **~/DO447/labs/data-lookups/i18n** directory.

```
[student@workstation ~]$ cd ~/DO447/labs/data-lookups/i18n/
[student@workstation i18n]$
```

- 2.2. There are two files in the `~/DO447/labs/data-lookups/i18n/roles/webapp/templates` directory. Those templates confirm the contents of the application in two different languages: `en_US.UTF-8` English (the default language) and `es_ES.UTF-8` Spanish. Review those files and note the template variables used.

```
[student@workstation i18n]$ cat roles/webapp/templates/webapp.en_US.UTF-8.j2
Greetings from {{ inventory_hostname }}.
```

```
App Version: {{ webapp_version }}
```

```
IaC Version: {{ iac_version }}
```

```
[student@workstation i18n]$ cat roles/webapp/templates/webapp.es_ES.UTF-8.j2
Saludos desde {{ inventory_hostname }}.
```

```
App Version: {{ webapp_version }}
```

```
IaC Version: {{ iac_version }}
```

- 2.3. Use your preferred editor to edit the `roles/webapp/defaults/main.yml` file. This file defines three variables:

controller_lang

Holds the locale configured in the controller.

iac_version

Contains the version of the Ansible project (the commit id, as previously stated).

webapp_version

Defines the version of the web application being deployed.

To obtain the locale for the controller host, use the `env lookup` plug-in to retrieve the value of the `LANG` environment variable. It is also a good practice to provide a default value, in case the environment variable is not available:

```
controller_lang: "{{ lookup('env', 'LANG') | default('en_US.UTF-8') }}"
```

Obtain the current Ansible project commit id by using the `git rev-parse --short HEAD` command. Use the `pipe lookup` to instruct Ansible to execute that command:

```
iac_version: "{{ lookup('pipe', 'git rev-parse --short HEAD') | quote }}"
```



Note

The `quote` filter is used to sanitize the string from inappropriate characters. In this case, the `git` command does not return any harmful character, but it is a good security practice to use the `quote` filter when using input from external sources.

The `roles/webapp/defaults/main.yml` file should display as follows:

```
controller_lang: "{{ lookup('env', 'LANG') | default('en_US.UTF-8') }}"
iac_version: "{{ lookup('pipe', 'git rev-parse --short HEAD')|quote }}"
webapp_version: v1.0
```

- 2.4. Use your preferred editor to edit the **roles/webapp/tasks/main.yml** file. Note the missing **content** attribute for the **copy** module. It should contain the appropriate contents of the web application for the calculated locale.

Use the **template lookup** to load and resolve the template related to the calculated locale:

```
content: "{{ lookup('template', 'webapp.' + locale + '.j2') }}"
```

The previous code looks in the role's templates directory for a template file whose name begins with **webapp.**, followed by the value of the locale, and with the extension **.j2**. The files that will be used are **webapp.en_US.UTF-8.j2** and **webapp.es_ES.UTF-8.j2**.

The **~/DO447/labs/data-lookups/i18n/roles/webapp/tasks/main.yml** file should display as follows:

```
---
- name: Ensure stub web content is deployed
  vars:
    locale: "{{ ansible_facts.env.LANG | default(controller_lang) }}"
  copy:
    content: "{{ lookup('template', 'webapp.' + locale + '.j2') }}"
    dest: /var/www/html/index.html
```

- 2.5. Set the **LANG** variable to **en_US.UTF-8** (it should already have that value). Verify the Ansible Playbook is correct and executes successfully:

```
[student@workstation i18n]$ LANG=en_US.UTF-8
[student@workstation i18n]$ ansible-playbook site.yml

PLAY [Ensure HAProxy is deployed] ****
...output omitted...
TASK [webapp : Ensure stub web content is deployed] ****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

PLAY RECAP ****
servera : ok=10 changed=9 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
serverb : ok=8 changed=6 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
serverc : ok=8 changed=6 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Validate that each server provides the right contents based on its configured locale. By twice reaching the **servera** (the load balancer), the **round-robin** algorithm redirects first query to **serverb**, and second query to **serverc**.

```
[student@workstation i18n]$ curl servera; curl servera;
Saludos desde serverb.lab.example.com.
```

```
App Version: v1.1a
IaC Version: 57eee8e
Greetings from serverc.lab.example.com.

App Version: v1.1b
IaC Version: 57eee8e
[student@workstation i18n]$ ssh serverb "printenv LANG"
es_ES.UTF-8
[student@workstation i18n]$ ssh serverc "printenv LANG"
[student@workstation i18n]$ printenv LANG
en_US.UTF-8
```

Note how the **serverb** locale is **es_ES.UTF-8** and the server responds with Spanish localized content. The locale is unset on the **serverc** server, so Ansible uses the locale defined in the controller: **en_US.UTF-8**.

- 2.6. Execute the **clean.yml** command to clean up the environment for future exercises:

```
[student@workstation i18n]$ ansible-playbook clean.yml
```

Finish

On **workstation**, run the **lab data-lookups finish** script to clean up and finish this exercise.

```
[student@workstation ~]$ lab data-lookups finish
```

This concludes the guided exercise.

Implementing Advanced Loops

Objectives

After completing this section, you should be able to use filters and lookup plug-ins to implement iteration over complex data structures.

Loops and Lookup Plug-ins

Using loops to iterate over tasks can help simplify your Ansible Playbooks. The **loop** keyword loops over a flat list of items. When used in conjunction with lookup plug-ins, you can construct more complex data in your lists for your loops.

The **loop** keyword was introduced in Ansible 2.5. Before that, task iteration was implemented by using keywords that started with **with_** and ended with the name of a lookup plug-in. The equivalent to **loop** in this syntax is **with_list**, and is designed for iteration over a simple flat list. For simple lists, **loop** is the best syntax to use.

As an example, the following three syntaxes have the same results. The first of these is preferred:

```
- name: using loop
  debug:
    msg: "{{ item }}"
  loop: "{{ mylist }}"

- name: using with_list
  debug:
    msg: "{{ item }}"
  with_list: "{{ mylist }}"

- name: using lookup plugin
  debug:
    msg: "{{ item }}"
  loop: "{{ lookup('list', mylist) }}"
```

You can refactor a **with_*** style iteration task to use the **loop** keyword, by using an appropriate combination of lookup plug-ins and filters to match the functionality.

Using the **loop** keyword in place of **with_*** style loops has the following benefits:

- No need to memorize or find a **with_*** style keyword to suit your iteration scenario. Instead, use plug-ins and filters to adapt a **loop** keyword task to your use case.
- Focus on learning the plug-ins and filters that are available in Ansible, which have broader applicability beyond just iteration.
- You have command-line access to the lookup plug-in documentation, with the **ansible-doc -t lookup** command. This helps you discover lookup plug-ins and design custom iteration scenarios using them.



Important

The message from Ansible upstream has been evolving since Ansible 2.5. Using the **loop** keyword instead of the **with_*** loops is recommended, but there are some use cases where the old syntax might be better. According to the documentation, the **with_*** syntax is not deprecated and it should be valid for the foreseeable future. The syntax of **loop** might continue to evolve in future releases of Ansible.

Some key tips, based on the upstream guidance for Ansible 2.8:

- The **loop** keyword requires a list, and will not accept a string. If you are having problems, then remember the difference between **lookup** and **query**.
- Any use of **with_*** that is discussed in "Migrating from with_X to loop" [https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html#migrating-to-loop] can be safely converted. These mostly use filters.
- If you need to use a lookup plug-in to convert a **with_*** construct to use **loop**, then it might be clearer to keep using the **with_*** syntax.

Example Iteration Scenarios

The following examples show some ways to construct more complex loops using Jinja2 expressions, filters, lookup plug-ins, and the **with_*** syntax.

Iterating over a List of Lists

The **with_items** keyword provides a way to iterate over complex lists. As an example, consider a hypothetical play with the following task:

```
- name: Remove build files
  file:
    path: "{{ item }}"
    state: absent
  with_items:
    - "{{ app_a_tmp_files }}"
    - "{{ app_b_tmp_files }}"
    - "{{ app_c_tmp_files }}
```

The **app_a_tmp_files** variable contains a list of temporary files, as do both the **app_b_tmp_files** and **app_c_tmp_files**. The **with_items** keyword combines these three lists into a single list containing the entries from all three lists. It automatically performs one level flattening of its list.

To refactor a **with_items** task to use the **loop** keyword, use the **flatten** filter. The **flatten** filter recursively searches for embedded lists, and creates a single list from discovered values.

The **flatten** filter accepts a **levels** argument, that specifies an integer number of levels to search for embedded lists. A **levels=1** argument specifies that values are obtained by only descending into one additional list for each item in the initial list. This is the same one level flattening that **with_items** does implicitly.

To refactor a **with_items** task to use the **loop** keyword, you must also use the **flatten(levels=1)** filter:

```

- name: Remove build files
  file:
    path: "{{ item }}"
    state: absent
  loop: "{{ list_of_lists | flatten(levels=1) }}"
vars:
  list_of_lists:
    - "{{ app_a_tmp_files }}"
    - "{{ app_b_tmp_files }}"
    - "{{ app_c_tmp_files }}"

```

**Important**

Because `loop` does not perform this implicit one level flattening, it is not exactly equivalent to `with_items`. However, as long as the list passed to the loop is a simple list, both methods will behave identically. The distinction only matters if you have a list of lists.

Iterating Over Nested Lists

Data from variable files, Ansible facts, and external services are often a composition of simpler data structures, like lists and dictionaries. Consider the `users` variable defined below:

```

users:
  - name: paul
    password: "{{ paul_pass }}"
    authorized:
      - keys/paul_key1.pub
      - keys/paul_key2.pub
    mysql:
      hosts:
        - "%"
        - "127.0.0.1"
        - "::1"
        - "localhost"
    groups:
      - wheel

  - name: john
    password: "{{ john_pass }}"
    authorized:
      - keys/john_key.pub
    mysql:
      password: other-mysql-password
      hosts:
        - "utility"
    groups:
      - wheel
      - devops

```

The `users` variable is a list. Each entry in the list is a dictionary with the keys: `name`, `password`, `authorized`, `mysql`, and `groups`. The keys `name` and `password` define simple strings, while the

authorized and **groups** keys define lists. The **mysql** key references another dictionary that contains MySQL related metadata for each user.

Similar to the **flatten** filter, the **subelements** filter creates a single list from a list with nested lists. The filter processes a list of dictionaries, and each dictionary contains a key that refers to a list. To use the **subelements** filter, you must provide the name of a key on each dictionary that corresponds to a list.

To illustrate, consider again the previous **users** variable definition. The **subelements** filter enables iteration through all users and their authorized key files defined in the variable:

```
- name: Set authorized ssh key
  authorized_key:
    user: "{{ item.0.name }}"
    key: "{{ lookup('file', item.1) }}"
  loop: "{{ users | subelements('authorized') }}"
```

The **subelements** filter creates a new list from the **users** variable data. Each item in the list is itself a two element list. The first element contains a reference to each user. The second element contains a reference to a single entry from the **authorized** list for that user.

Iterating Over a Dictionary

You often encounter data that is organized as a set of key/value pairs, commonly referred to in the Ansible community as a dictionary, instead of being organized as a list. As an example, consider the following definition of a **users** variable:

```
users:
  demo1:
    name: Demo User 1
    mail: demo1@example.com
  demo2:
    name: Demo User 2
    mail: demo2@example.com
  ...output omitted...
  demo200:
    name: Demo User 200
    mail: demo200@example.com
```

Before Ansible 2.5, you would have to use the **with_dict** keyword to iterate through the key/value pairs of this dictionary. Each iteration, the **item** variable has available two attributes: **key** and **value**. The **key** attribute contains the value of one of the dictionary keys, while the **value** attribute contains the data associated with the dictionary key:

```
- name: Iterate over Users
  user:
    name: "{{ item.key }}"
    comment: "{{ item.value.name }}"
    state: present
  with_dict: "{{ users }}"
```

Alternatively, you can use the **dict2items** filter to transform a dictionary into a list, and this is probably easier to understand. The items in this list are structured the same as items produced by the **with_dict** keyword:

```
- name: Iterate over Users
  user:
    name: "{{ item.key }}"
    comment: "{{ item.value.name }}"
    state: present
  loop: "{{ users | dict2items }}"
```

Iterating Over a File Globbing Pattern

You can construct a loop that iterates over a list of files that match a provided file globbing pattern with the **fileglob** lookup plug-in.

To illustrate, consider the following play:

```
- name: Test
  hosts: localhost
  gather_facts: no
  tasks:
    - name: Test fileglob lookup plugin
      debug:
        msg: "{{ lookup('fileglob', '~/.bash*') }}"
```

Output from the **fileglob** lookup plug-in is a string of comma-separated files, indicated by the use of the double-quotes character ("") around the **msg** variable's data:

```
PLAY [Test] ****
TASK [Test fileglob lookup plugin] ****
ok: [localhost] => {
    "msg": "/home/student/.bash_logout,/home/student/.bash_profile,/home/
student/.bashrc,/home/student/.bash_history"
}

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

To force a lookup plug-in to return a list of values, instead of a string of comma-separated values, use the **query** keyword in place of the **lookup** keyword. Consider the following modification of the previous playbook example:

```
- name: Test
  hosts: localhost
  gather_facts: no
  tasks:
    - name: Test fileglob lookup plugin
      debug:
        msg: "{{ query('fileglob', '~/.bash*') }}"
```

The output of this modified playbook indicates that the **msg** keyword references a list of files, because the data is encapsulated by brackets ([...]):

```
PLAY [Test] ****
TASK [Test fileglob lookup plugin] ****
ok: [localhost] => {
  "msg": [
    "/home/student/.bash_logout",
    "/home/student/.bash_profile",
    "/home/student/.bashrc",
    "/home/student/.bash_history"
  ]
}

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

To use the data from this lookup plug-in in a loop, ensure that the processed data is returned as a list. Both tasks in the following play iterate over the files matching the `~/.bash*` globbing pattern:

```
- name: Both tasks have the same result
hosts: localhost
gather_facts: no
tasks:

- name: Iteration Option One
  debug:
    msg: "{{ item }}"
  loop: "{{ query('fileglob', '~/.bash*') }}"

- name: Iteration Option Two
  debug:
    msg: "{{ item }}"
  with_fileglob:
    - "~/.bash*"
```



References

Loops – Ansible Documentation

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

► Guided Exercise

Implementing Advanced Loops

In this exercise, you will implement advanced iteration techniques to process complex data structures in three different scenarios.

Outcomes

You should be able to use filters and lookup plugins to iterate over complex data structures.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab data-loops start
```

After the command completes, change to the **~/DO447/labs/data-loops** directory.

```
[student@workstation ~]$ cd ~/DO447/labs/data-loops
[student@workstation data-loops]$
```



Note

Scenario 1

The **scenario_1.yml** playbook contains a single play with a single task to create users on the Identity Management server. The task uses the **with_dict** keyword to iterate over entries in the **users** variable. The **users** variable is defined in the **group_vars/all/users.yml** file.

- 1. In this step, you will refactor the task in the **scenario_1.yml** playbook to use the **loop** keyword and remove the **with_dict** keyword. Use an appropriate filter to transform the **users** variable into a list that you can use with the **loop** keyword.

- 1.1. Review the **scenario_1.yml** playbook to familiarize yourself with it.

The **ipa_*** variables are defined in the **group_vars/all/ipa_vars.yml** file, except for the **ipa_admin_pass** variable. Because this value is sensitive, it is defined in the **group_vars/all/vault.yml** Ansible Vault file. Use a password of **redhat321** to decrypt the file.

The **users** variable is defined in the **group_vars/all/users.yml** file.

```
[student@workstation data-loops]$ cat scenario_1.yml
---
- name: Add Users To IDM
  hosts: localhost
  gather_facts: no
  tasks:
```

```
- name: Create Users
  ipa_user:
    name: "{{ item.key }}"
    givenname: "{{ item.value.firstname }}"
    sn: "{{ item.value.surname }}"
    displayname: "{{ item.value.firstname + ' ' + item.value.surname }}"
    sshpubkey: "{{ lookup('file', item.value.pub_key_file) }}"
    state: present
    ipa_host: "{{ ipa_server }}"
    ipa_user: "{{ ipa_admin_user }}"
    ipa_pass: "{{ ipa_admin_pass }}"
    validate_certs: "{{ ipa_validate_certs }}"
  with_dict: "{{ users }}"
```

- 1.2. Review the structure of the **users** group variable defined in **group_vars/all/users.yml** to familiarize yourself with it.

The **users** variable is a dictionary. Each key in the dictionary represents a username, and the value for each key is another dictionary containing metadata for that user. Each user metadata dictionary contains three keys: **firstname**, **surname**, and **pub_key_file**.

```
[student@workstation data-loops]$ cat group_vars/all/users.yml
users:
  johnd:
    firstname: John
    surname: Doe
    pub_key_file: pubkeys/johnd/id_rsa.pub
  janed:
    firstname: Jane
    surname: Doe
    pub_key_file: pubkeys/janed/id_rsa.pub
```

- 1.3. Instead of using the **with_dict** keyword, you can use the **loop** keyword with the **dict2items** filter for your loop. Change the **with_dict** keyword to the **loop** keyword, and add the **dict2items** filter to the end of the Jinja2 expression. Save the file.

The expected content of the **scenario_1.yml** playbook is:

```
[student@workstation data-loops]$ cat scenario_1.yml
---
- name: Add Users To IDM
  hosts: localhost
  gather_facts: no
  tasks:
    - name: Create Users
      ipa_user:
        name: "{{ item.key }}"
        givenname: "{{ item.value.firstname }}"
        sn: "{{ item.value.surname }}"
        displayname: "{{ item.value.firstname + ' ' + item.value.surname }}"
        sshpubkey: "{{ lookup('file', item.value.pub_key_file) }}"
        state: present
      ipa_host: "{{ ipa_server }}"
```

```

ipa_user: "{{ ipa_admin_user }}"
ipa_pass: "{{ ipa_admin_pass }}"
validate_certs: "{{ ipa_validate_certs }}"
loop: "{{ users | dict2items }}"

```

14. Execute the **scenario_1.yml** playbook. Because the playbook uses Ansible Vault variables, use the **--vault-id @prompt** option with the **ansible-playbook** command. Enter a value of **redhat321** for the password:

```

[student@workstation data-loops]$ ansible-playbook --vault-id @prompt \
> scenario_1.yml
Vault password (default): redhat321
PLAY [Add Users To IDM] ****

TASK [Create Users] ****
changed: [localhost] => (item={'key': 'johnd', 'value': {'firstname': 'John',
'surname': 'Doe', 'pub_key_file': 'pubkeys/johnd/id_rsa.pub'}})
changed: [localhost] => (item={'key': 'janed', 'value': {'firstname': 'Jane',
'surname': 'Doe', 'pub_key_file': 'pubkeys/janed/id_rsa.pub'}})

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

```



Note

Scenario 2.

The **scenario_2.yml** playbook configures a database server and enables appropriate user access to databases hosted on the server. A playbook task uses the **with_subelements** keyword to iterate over a list of databases for each user in the **db_user_access** variable. The **db_user_access** variable is defined in the **group_vars/all/db_vars.yml** file.

In this step, you will refactor the **scenario_2.yml** playbook to use the **loop** keyword and remove the **with_subelements** keyword. Use an appropriate filter to transform the **db_user_access** variable into a list that you can use with the **loop** keyword.

- 2.1. Review the **scenario_2.yml** playbook to familiarize yourself with it.

The **user_passwords** variable is a simple dictionary. Each key represents a username, and each key's value is the associated username's password. Because this variable contains sensitive information, it is protected in the **group_vars/all/vault.yml** Ansible Vault file.

The **db_user_access** variable is defined in the **group_vars/all/db_vars.yml** file.

```

[student@workstation data-loops]$ cat scenario_2.yml
---
- name: Enable Database Access
  hosts: db_server
  gather_facts: no
  tasks:
    - import_tasks: database_setup.yml

```

```

- name: Enable user access to database
  mysql_user:
    name: "{{ item.0.username }}"
    priv: "{{ item.1 }}.*:ALL"
    host: workstation.lab.example.com
    append_privs: yes
    password: "{{ user_passwords[item.0.username] }}"
    state: present
  with_subelements:
    - "{{ db_user_access }}"
    - db_list
...output omitted...

```

- 2.2. Review the structure of the **db_user_access** group variable defined in the file **group_vars/all/db_vars.yml** to familiarize yourself with it.

The **db_user_access** variable is a list. Each entry in the list is itself a dictionary, with keys of: **username**, **dept**, **role**, and **db_list**. The **db_list** key references a list of databases to which the given user is granted access. As an example, the **johnd** user is granted access to both the **employees** and **invoices** databases.

Each user requires access to each database in their **db_list** list.

```

[student@workstation data-loops]$ cat group_vars/all/db_vars.yml
...output omitted...
db_user_access:
  - username: johnd
    dept: sales
    role: manager
    db_list:
      - employees
      - invoices

  - username: janed
    dept: sales
    role: associate
    db_list:
      - invoices
      - inventory
...output omitted...

```

- 2.3. Change the **with_subelements** keyword to the **loop** keyword, and add the **subelements** filter to the end of the Jinja2 expression. Configure the **subelements** filter to process the **db_list** key of each entry in the **db_user_access** list. Save the file.

The content of the **scenario_2.yml** playbook is:

```

[student@workstation data-loops]$ cat scenario_2.yml
---
- name: Enable Database Access
  hosts: db_server
  gather_facts: no
  tasks:
    - import_tasks: database_setup.yml

```

```

- name: Enable user access to database
  mysql_user:
    name: "{{ item.0.username }}"
    priv: "{{ item.1 }}.*:ALL"
    host: workstation.lab.example.com
    append_privs: yes
    password: "{{ user_passwords[item.0.username] }}"
    state: present
  loop: "{{ db_user_access | subelements('db_list') }}"

- name: Smoke Test - database connectivity
  shell: "{{ lookup('template', 'db_test_command.j2') }}"
  loop: []
  delegate_to: workstation
  changed_when: no

```

- 2.4. Replace the empty list in the **loop** keyword of the smoke test task with the same Jinja2 expression from the **Enable user access to database** task. Save the file. The content of smoke test task follows:

```

- name: Smoke Test - database connectivity
  shell: "{{ lookup('template', 'db_test_command.j2') }}"
  loop: "{{ db_user_access | subelements('db_list') }}"
  delegate_to: workstation
  changed_when: no

```

- 2.5. Some of the variable references in the **templates/db_test_command.j2** file are not correct. Change the **username** and **database** variables to reference the correct attributes of each iteration item. Use the preceding task as a guide for the variable references. Save the file.

The content of the **templates/db_test_command.j2** template is:

```

echo "use {{ item.1 }}" | \
mysql -u {{ item.0.username }} \
-p{{ user_passwords[item.0.username] }} \
-h {{ inventory_hostname }}

```

This single command tests that a user can connect to the database server and use a particular database.



Note

There is no space between the **-p** option and the password value, unlike the **-u** and **-h** options. Do not modify anything other than variable references inside a Jinja2 expression.

- 2.6. Execute the **scenario_2.yml** playbook and verify that each user can access each database specified in the corresponding **db_list** list.

```
[student@workstation data-loops]$ ansible-playbook --ask-vault-pass scenario_2.yml
Vault password: redhat321
```

```

PLAY [Enable Database Access] ****
...output omitted...

TASK [Enable user access to database] ****
changed: [servere] => (item=[{'username': 'johnd', ...}, 'employees'])
changed: [servere] => (item=[{'username': 'johnd', ...}, 'invoices'])
changed: [servere] => (item=[{'username': 'janed', ...}, 'invoices'])
changed: [servere] => (item=[{'username': 'janed', ...}, 'inventory'])

TASK [Smoke Test - database connectivity] ****
ok: [servere] => (item=[{'username': 'johnd', ...}, 'employees'])
ok: [servere] => (item=[{'username': 'johnd', ...}, 'invoices'])
ok: [servere] => (item=[{'username': 'janed', ...}, 'invoices'])
ok: [servere] => (item=[{'username': 'janed', ...}, 'inventory'])

PLAY RECAP ****
servere : ok=6    changed=4    unreachable=0    failed=0    ...

```

**Note**

Scenario 3

The **scenario_3.yml** playbook creates a **developer** user account on the development web servers. The playbook also adds the SSH public key of any developer to the **authorized_keys** file for the **developer** user on the development servers.

In this step, you will refactor the second task to loop over the **public_keys_lists** variable defined in the **group_vars/all/public_keys.yml** file. Use the **map** filter, followed by the **flatten** filter, to generate a list of a simple list of SSH public key files for all developers. Update the task's **key** keyword to contain the file content of each iteration item.

- Review the **scenario_3.yml** playbook to familiarize yourself with it.

The **with_file** keyword loops through each file in the list. The **key** keyword contains the content of the file in the given iteration.

The **scenario_3.yml** playbook is tightly coupled with the list of public key files, which reduces the ability to reuse the playbook in other environments.

In a later step, you extract this same list of files from the **public_key_lists** variable.

```

[student@workstation data-loops]$ cat scenario_3.yml
---
- name: Allow Developer Access To Dev Servers
  hosts: dev_web_servers
  gather_facts: no
  tasks:
    - name: Create shared account
      user:
        name: developer
        state: present

```

```

- name: Set up multiple authorized keys
  authorized_key:
    user: developer
    state: present
    key: "{{ item }}"
  with_file:
    - pubkeys/johnd/id_rsa.pub
    - pubkeys/johnd/laptop_rsa.pub
    - pubkeys/janed/id_rsa.pub

```

- 3.2. Review the structure of the **public_key_lists** variable:

```
[student@workstation data-loops]$ cat group_vars/all/public_keys.yml
---
public_key_lists:
  - username: johnd
    public_keys:
      - pubkeys/johnd/id_rsa.pub
      - pubkeys/johnd/laptop_rsa.pub
  - username: janed
    public_keys:
      - pubkeys/janed/id_rsa.pub
```

The **public_key_lists** variable is a list. Each entry in the list is itself a dictionary, with keys of: **username** and **public_keys**. The **public_keys** key references a list of public key files for the given user.

As an example, the **johnd** user has two public key files: **pubkeys/johnd/id_rsa.pub** and **pubkeys/johnd/laptop_rsa.pub**. The filenames are relative to the root directory of the playbook project.

- 3.3. Change the **with_file** keyword to the **loop** keyword, and start a Jinja2 expression that references the **public_key_lists** variable. Add a **map** filter to the end of the expression to extract the **public_keys** attribute. This creates a list of lists - each entry in the list is a list of SSH public key file for a particular user.

To create a single list of files, add a **flatten** filter after the **map** filter.

Use the **file** lookup plugin to configure the **key** keyword with the file content of each iteration item. Save the file.

The content of the **scenario_3.yml** playbook is:

```
[student@workstation data-loops]$ cat scenario_3.yml
---
- name: Allow Developer Access To Dev Servers
  hosts: dev_web_servers
  gather_facts: no
  tasks:
    - name: Create shared account
      user:
        name: developer
        state: present
    - name: Set up multiple authorized keys
      authorized_key:
        user: developer
```

```
state: present
key: "{{ lookup('file', item) }}"
loop: "{{ public_key_lists | map(attribute='public_keys') | flatten }}"
```

- 3.4. Execute the **scenario_3.yml** playbook. Verify that the playbook process the same three public key files.

```
[student@workstation data-loops]$ ansible-playbook --ask-vault-pass scenario_3.yml
Vault password: redhat321

PLAY [Allow Developer Access To Dev Servers] ****
TASK [Create shared account] ****
changed: [servera]

TASK [Set up multiple authorized keys] ****
changed: [servera] => (item=pubkeys/johnd/id_rsa.pub)
changed: [servera] => (item=pubkeys/johnd/laptop_rsa.pub)
changed: [servera] => (item=pubkeys/janed/id_rsa.pub)

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

Finish

On **workstation**, run the **lab data-loops finish** script to complete this lab.

```
[student@workstation ~]$ lab data-loops finish
```

This concludes the guided exercise.

Working with Network Addresses Using Filters

Objectives

After completing this section, students should be able to inspect, validate, and manipulate variables containing network information with filters.

Gathering and Processing Network Information

A number of filters and lookup plugins are available to collect and process network information for Ansible automation. These filters and plugins can be useful in conjunction with fact gathering when you configure the network devices on your managed hosts.

The standard **setup** module that automatically gathers facts at the start of many plays collects a lot of network-related information from each managed host.



Note

If you only need network-related facts, you can run **setup** as an explicit task and use the **gather_subset** option to limit collection to those facts.

```
- name: Collect only network related facts
  setup:
    gather_subset:
      - '!all'
      - network
```

There are a number of particularly useful facts. The fact **ansible_facts['interfaces']** is a list of all the network interface names on the system. You can use this list to examine the facts for each of the network interfaces on the system. For example, if **enp11s0** is an interface on the system, then it has a fact named **ansible_facts['enp11s0']** that is a dictionary containing its MAC address, IPv4 and IPv6 addresses, kernel module, and so on as values.

There are some other useful facts to remember:

Selected Networking Facts

Fact name	Description
ansible_facts['dns']['nameservers']	The DNS nameservers used for name resolution by the managed host.
ansible_facts['domain']	The domain for the managed host.
ansible_facts['all_ipv4_addresses']	All the IPv4 addresses configured on the managed host.
ansible_facts['all_ipv6_addresses']	All the IPv6 addresses configured on the managed host.

Fact name	Description
<code>ansible_facts['fqdn']</code>	The fully-qualified domain name (DNS name) for the managed host.
<code>ansible_facts['hostname']</code>	The unqualified hostname, the string in the FQDN before the first period.

An example playbook using these facts would be:

```
---
- name: Set up web servers
  hosts: web_servers
  become: true

  tasks:
    - name: Ensure httpd is installed
      yum:
        name: httpd
        state: installed

    - name: Start and enable webserver
      service:
        name: httpd
        state: started
        enabled: yes

    - name: Notify root of server provisioning
      mail:
        subject: System {{ ansible_facts['hostname'] }} has been successfully
        provisioned.
```

Networking Information Filters

The `ipaddr` filter provides capabilities to manipulate and validate facts related to networking. You can use it to check the syntax of IP addresses, convert from VLSN subnet masks to CIDR subnet prefix notation, perform subnet math, and find the next usable address in a network range, and so on.

Comprehensive documentation for this filter is available at "ipaddr filter" [https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters_ipaddr.html], but this section will explore some of its uses.



Important

The `ipaddr` filter needs the `netaddr` Python module, provided by the `python3-netaddr` package in Red Hat Enterprise Linux 8. Make sure that the package is installed on your control node, using Ansible or the Yum package manager.

In its simplest form, the `ipaddr` filter with no arguments accepts a single value. If the value is an IP address, then the filter returns the IP address. If it is not an IP address, then the filter returns `False`.

If the value is a list, then the filter returns the valid IP addresses and not the invalid ones. If all of the items are invalid, it returns an empty list.

```
 {{ my_hosts_list | ipaddr }}
```

However, you can also pass arguments to the **ipaddr** filter. These arguments can make the filter return different information based on the data input. For example, if **my_network** has the value **10.0.0.1/23** then the following Jinja2 expression will output the variable-length subnet mask (VLSN) **255.255.254.0**:

```
 {{ my_network | ipaddr('netmask') }}
```

The **ipaddr** filter accepts the following options:

address

Validates input values are valid IP addresses. If the input includes network prefix, it is stripped out.

net

Validates input values are network ranges and return them in CIDR format.

host

Ensures IP addresses conform to the equivalent CIDR prefix format.

prefix

Validates that the input host satisfies the host/prefix or CIDR format, and returns its prefix (size of the network's mask in bits)

host/prefix

Validates the input value is in host/prefix format. That is, the host part is a usable IP address on the network and the prefix is a valid CIDR prefix.

network/prefix

Validates the input value is in network/prefix format. That is, the network part is the network address (lowest IP address on the network), and the prefix is a valid CIDR prefix.

public or private

Validates input IP addresses or network ranges are in ranges that are reserved by IANA to be public or private, respectively.

size

Transform the input network range to the number of IP addresses in that range.

Any integer number (n).

Transform a network range to the n-th element in that range. Negative numbers return the n-th element from last.

network, netmask, and broadcast

Validates that the input host satisfies the host/prefix or CIDR format, and transforms it into the network address (applies the netmask to the host), netmask, or broadcast address, respectively.

subnet

Validates that the input host satisfies the host/prefix or CIDR format, and returns the subnet containing that host.

ipv4 and ipv6

Validates input are valid network ranges and transform them into IPv4 and IPv6 formats, respectively.

The **ipaddr** filter accepts some other queries not listed here. Find more information about using this filter in the **References** note at the end of this section.

Collecting Network Information with Plug-ins

A number of lookup plug-ins are available that you can use to collect information about the network environment. These include **dig** to look up DNS information, **dnstxt** to get DNS TXT records, and more esoteric plug-ins such as **aws_service_ip_ranges** to get the IP ranges for Amazon AWS EC2 services and **nios_next_ip** to get the next available IP for a network from Infoblox.

Looking Up DNS Information

As implied by its name, the **dig** lookup is an interface for the **dig** command. The **dig** command runs queries against DNS servers and returns the resulting records. This lookup makes use of the *dnspython* python library, from the *python3-dns* package, that must be available in the controller node.

In its simplest form, the **dig** lookup queries the DNS server configured in the controller for the **A** record of the provided fully-qualified domain name (FQDN).

```
"{{ lookup('dig', 'example.com') }}"
```

To obtain a different type of record, provide an additional **qtype** parameter to the lookup, or append a slash (/) and the record type to the FQDN. The following examples obtain the same **MX** record from the **example.com** domain.

```
"{{ lookup('dig', 'example.com', 'qtype=MX') }}"
"{{ lookup('dig', 'example.com/MX') }}"
```

The **dig** lookup allows using different DNS servers than the ones configured in the controller. Provide an additional parameter with a comma-separated list of the DNS servers to use, prefixed by the @ sign as follows:

```
"{{ lookup('dig', 'example.com', '@4.4.8.8,4.4.4.4') }}"
```



Note

Some DNS records may contain multiple values for the same record type. In this case, the **dig** lookup returns a single value, containing a comma-separated list of the returned values. To obtain a list of values, use **query** with the lookup filter.

```
"{{ query('dig', 'example.com/MX') }}"
```

The **dig** lookup can provide DNS text records (**qtype=TXT**), like any other kind. In case only **TXT** records are needed, the **dnstxt** lookup can replace the **dig** lookup. The **dnstxt** lookup provides a much simpler format and depends on the Python *dns/dns.resolver* library from the *python3-dns* RPM package in Red Hat Enterprise Linux 8.

```
"{{ lookup('dnstxt', ['test.example.com']) }}"
```

Here you can see an example of these filters and functions:

```
---
- name: ipaddr example
  hosts: lb_servers

  tasks:
    - name: Determinte if host's ip address is private
      debug:
        msg: "{{ lookup('dig', ansible_facts['hostname']) | ipaddr('private') }}"
```



References

Ansible documentation for the ipaddr filter

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

Ansible documentation for the dig filter

<https://docs.ansible.com/ansible/latest/plugins/lookup/dig.html>

Ansible documentation for the dnstxt filter

<https://docs.ansible.com/ansible/latest/plugins/lookup/dnstxt.html>

► Guided Exercise

Working with Network Addresses Using Filters

In this exercise, you will use the **ipaddr** filter and **dig** lookup plug-in to manipulate networking data from managed hosts.

Outcomes

You should be able to use the **ipaddr** filter to derive networking information from IP data retrieved from gathered facts. You should be able to use the **dig** lookup plug-in to obtain DNS data.

Before You Begin

Log in as the **student** user on **workstation** and run **lab data-netfilters start**. This setup script installs the needed Python requirements and gather the lab exercise files.

```
[student@workstation ~]$ lab data-netfilters start
```

- 1. Review the project files in the **~/D0447/labs/data-netfilters** directory. Use your preferred editor to open the **~/D0447/labs/data-netfilters/roles/netfilters/tasks/main.yml** file. This file contains tasks that you complete with the **ipaddr** filter and **dig** lookup plug-in. Replace any occurrence of an ellipsis (**...**) in a fact definition with the expected value.

Most of the tasks leverage the pre-populated **ansible_facts.default_ipv4** fact. This fact contains several keys, as shown here:

```
ansible_facts['default_ipv4']: {
    address: "192.168.0.2",
    alias: "eth0",
    broadcast: "192.168.0.255",
    gateway: "192.168.0.1",
    interface: "eth0",
    macaddress: "8a:f2:c0:66:b2:cb",
    mtu: 1500,
    netmask: "255.255.255.0",
    network: "192.168.0.0",
    type: "ether"
}
```

- 1.1. The first task verifies that the **default_ipv4.address** variable contains a properly formatted IP address. The **ipaddr** filter narrows down valid IPs if it receives no other parameter.

```
server_address: "{{ ansible_facts.default_ipv4.address | ipaddr }}"
```

If the input value (`ansible_facts.default_ipv4.address`) is valid, `ipaddr` assigns the same value to the resulting fact (`server_address`). Else, `ipaddr` returns an empty string.

- 1.2. The third task's objective is to obtain the DNS name associated to the managed host. This is also known as **reverse DNS**, or **PTR record** resolution, and is commonly used to validate the DNS configuration of mail servers.

To resolve the **reverse DNS** entry, provide the `revdns` parameter to the `ipaddr` filter:

```
address_dns: "{{ server_address | ipaddr('revdns') }}"
```

In this case, there is no explicit PTR record for the managed host, so an **in-addr.arpa** name is returned.

- 1.3. The fifth task creates a fact containing a network definition by appending the `ansible_facts.default_ipv4.network` fact, a slash (/) and the `ansible_facts.default_ipv4.netmask` fact. This fact is used in the seventh task to generate the CIDR format for the network.

Update the expression for the `cidr` variable in the seventh task. Add the `ipaddr` filter with the `net` parameter:

```
cidr: "{{ net_mask | ipaddr('net') }}"
```

In this case, the `ipaddr` filter transforms the network/mask value **172.25.250.0/255.255.255.0** to the **CIDR** equivalent value **172.25.250.0/24**.

- 1.4. Modify the expression for the `address_in_range` variable in the ninth task to check that the `server_address` variable belongs to the network:

```
address_in_range: "{{ server_address | ipaddr(net_mask) }}"
```

The ninth task validates that the managed host address obtained in the first task actually belongs to the network definition created in the fifth and seventh tasks. If the `ipaddr` filter receives a valid network definition, the filter checks that the input parameter IP address belongs to the provided network. Else, the IP address is filtered out, leaving an empty result.



Note

You can also check that the IP address belongs to the network with the CIDR specification:

```
address_in_range: "{{ server_address | ipaddr(cidr) }}"
```

- 1.5. The eleventh task obtains the broadcast address for a network definition. Use the `broadcast` parameter for the `ipaddr` filter, and provide the network definition in either network/mask or CIDR format:

```
broadcast: "{{ cidr | ipaddr('broadcast') }}"
```

**Note**

In most cases, the broadcast for an IP4 network is just the last IP in its range. This can be calculated by using the **-1** parameter for the **ipaddr** filter, but this practice is discouraged as it may fail in IP6 networks.

- 1.6. The thirteenth task leverages the **dig** lookup plug-in to obtain a DNS record for the **example.com** domain.

Use the **dig** lookup plug-in to set the value of the **dig_record** variable to the MX record associated with the **example.com** domain. To do this, the **dig** lookup plug-in requires the domain and record type as parameters:

```
dig_record: "{{ lookup('dig', 'example.com.', 'qtype=MX') }}"
```

If no **qtype** parameter is provided, then the **dig** filter returns the **A** record type.

- 2. Execute the **site.yml** playbook to validate your changes. The role contains **assert** tasks to validate that each fact is set to the correct value.

```
[student@workstation ~]$ cd DO447/labs/data-netfilters
[student@workstation data-netfilters]$ ansible-playbook site.yml

PLAY [Setup netfilter guided exercise] ****
...output omitted...
PLAY RECAP ****
servera...: ok=15 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

If playbook execution fails, you can find the solution file in the **~/DO447/solutions/data-netfilters** directory.

Finish

On **workstation**, run the **lab data-netfilters finish** script to clean up and finish this exercise.

```
[student@workstation ~]$ lab data-netfilters finish
```

This concludes the guided exercise.

▶ Lab

Transforming Data with Filters and Plugins

Performance Checklist

In this lab, you will implement filters and lookup plug-ins to create a better structured playbook project.

Outcomes

You should be able to use filters and lookup plug-ins to manipulate variables in playbook and role tasks.

Before You Begin

Log in as the **student** user on **workstation** and run **lab data-review start**.

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

This script initializes the remote Git repository you need for this lab, `http://git.lab.example.com:8081/git/data-review.git`. The Git repository contains playbooks that configures a front end load balancer and a pool of backend web servers.

1. In the **/home/student/git-repos** directory, clone the project repository `http://git.lab.example.com:8081/git/data-review.git`.
2. Refactor the tasks in the **firewall** role to allow for the flexible specification of firewall rules. The role processes the **firewall_rules** variable, which defines a list of firewall rule specifications.

Ensure that the **firewall** role tasks use the following logic for firewall rules:

- If a rule's **state** attribute is not defined, then any firewall task uses a default value of **enabled** for the **state** keyword.
- If a rule's **zone** attribute is not defined, then omit the **zone** keyword for firewall tasks.
- If a rule attempts to enable or disable a port, and the **protocol** attribute is not defined, then the protocol defaults to a value of **tcp**. Additionally, ensure that the protocol value is always lowercase.

Use the **test_firewall_role.yml** playbook to test your changes but do not alter the playbook. Only make changes to the tasks of the **firewall** role. After you correctly refactor the role, the **test_firewall_role.yml** playbook executes without errors.

3. Replace **load_balancer_ip_addr** in the **templates/apache_firewall_rules.yml.j2** template with a lookup plug-in that retrieves the IP address of the **load_balancer** variable. Make sure to install the necessary RPM package for the lookup plug-in to work.

4. Use the `templates/apache_firewall_rules.yml.j2` template to define the `firewall_rules` variable in the `deploy_apache.yml` playbook.

You need a Jinja2 expression that uses a lookup plug-in to render the template as a string, followed by a filter that converts from a YAML string to a data structure.

5. Execute the `site.yml` playbook to test your changes. If you make all of the necessary changes, then the playbook executes without any errors.

Verify that requests to the load balancer on `servera` from `workstation` succeed, while direct requests to the backend web servers, `serverb` and `serverc`, from `workstation` are denied.

When testing requests to backend servers, recall that the `apache_port` variable configures the port of the backend Apache servers, and it is set to a value of **8008**.

6. Save, commit, and push your changes to the remote repository.

Evaluation

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



Note

Make sure you commit and push your changes to the Git repository before rerunning the script.

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

Finish

From `workstation`, run the `lab data-review finish` command to complete this lab.

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

This concludes the lab.

► Solution

Transforming Data with Filters and Plugins

Performance Checklist

In this lab, you will implement filters and lookup plug-ins to create a better structured playbook project.

Outcomes

You should be able to use filters and lookup plug-ins to manipulate variables in playbook and role tasks.

Before You Begin

Log in as the **student** user on **workstation** and run **lab data-review start**.

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

This script initializes the remote Git repository you need for this lab, `http://git.lab.example.com:8081/git/data-review.git`. The Git repository contains playbooks that configures a front end load balancer and a pool of backend web servers.

1. In the **/home/student/git-repos** directory, clone the project repository `http://git.lab.example.com:8081/git/data-review.git`.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos
[student@workstation ~]$ cd /home/student/git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/data-review.git
...output omitted...
[student@workstation git-repos]$ cd data-review
[student@workstation data-review]$
```

2. Refactor the tasks in the **firewall** role to allow for the flexible specification of firewall rules. The role processes the **firewall_rules** variable, which defines a list of firewall rule specifications.

Ensure that the **firewall** role tasks use the following logic for firewall rules:

- If a rule's **state** attribute is not defined, then any firewall task uses a default value of **enabled** for the **state** keyword.
- If a rule's **zone** attribute is not defined, then omit the **zone** keyword for firewall tasks.
- If a rule attempts to enable or disable a port, and the **protocol** attribute is not defined, then the protocol defaults to a value of **tcp**. Additionally, ensure that the protocol value is always lowercase.

Use the **test_firewall_role.yml** playbook to test your changes but do not alter the playbook. Only make changes to the tasks of the **firewall** role. After you correctly refactor the role, the **test_firewall_role.yml** playbook executes without errors.

- 2.1. Add **| default('enabled')** at the end of each Jinja2 expression for the **state** keyword of all three **firewall** tasks. After you make these changes, the content of the **roles/firewall/tasks/main.yml** file is:

```
- name: Ensure Firewall Sources Configuration
  firewalld:
    source: "{{ item.source }}"
    zone: "{{ item.zone }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.source is defined
  notify: reload firewalld

- name: Ensure Firewall Service Configuration
  firewalld:
    service: "{{ item.service }}"
    zone: "{{ item.zone }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.service is defined
  notify: reload firewalld

- name: Ensure Firewall Port Configuration
  firewalld:
    port: "{{ item.port }}/{{ item.protocol }}"
    zone: "{{ item.zone }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.port is defined
  notify: reload firewalld
```

- 2.2. Add **| default(omit)** at the end of each Jinja2 expression for the **zone** keyword of all three **firewall** tasks. After you make these changes, the content of the **roles/firewall/tasks/main.yml** file is:

```

- name: Ensure Firewall Sources Configuration
  firewalld:
    source: "{{ item.source }}"
    zone: "{{ item.zone | default('omit') }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.source is defined
  notify: reload firewalld

- name: Ensure Firewall Service Configuration
  firewalld:
    service: "{{ item.service }}"
    zone: "{{ item.zone | default('omit') }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.service is defined
  notify: reload firewalld

- name: Ensure Firewall Port Configuration
  firewalld:
    port: "{{ item.port }}/{{ item.protocol }}"
    zone: "{{ item.zone | default('omit') }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.port is defined
  notify: reload firewalld

```

- 2.3. Replace `{{ item.protocol }}` in the third `firewall` task with `{{ item.protocol | default('tcp') | lower }}`. After you make these changes, the content of the third task is:

```

- name: Ensure Firewall Port Configuration
  firewalld:
    port: "{{ item.port }}/{{ item.protocol | default('tcp') | lower }}"
    zone: "{{ item.zone | default('omit') }}"
    permanent: yes
    state: "{{ item.state | default('enabled') }}"
  loop: "{{ firewall_rules }}"
  when: item.port is defined
  notify: reload firewalld

```

- 2.4. Use the `test_firewall_role.yml` playbook to test your changes:

```
[student@workstation data-review]$ ansible-playbook test_firewall_role.yml

PLAY [Test Firewall Role] ****
...output omitted...

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

3. Replace **load_balancer_ip_addr** in the **templates/apache_firewall_rules.yml.j2** template with a lookup plug-in that retrieves the IP address of the **load_balancer** variable. Make sure to install the necessary RPM package for the lookup plug-in to work.

Use the **dig** plug-in in the **templates/apache_firewall_rules.yml.j2** template to retrieve the IP address of each load balancer:

```
- port: {{ apache_port }}
  protocol: TCP
  zone: internal
{% for load_balancer in groups['lb_servers'] %}
- zone: internal
  source: "{{ lookup('dig', load_balancer) }}"
{% endfor %}
```

Save the template. Recall that to use the **dig** lookup plug-in, you must also install the **python3-dns** package on the host where you execute the playbook:

```
[student@workstation data-review]$ sudo yum install python3-dns
```

Enter the password for the **student** user, **student**, when you are prompted. Review the packages that will install, if any, and enter a **y** character when prompted to begin package installation.

4. Use the **templates/apache_firewall_rules.yml.j2** template to define the **firewall_rules** variable in the **deploy_apache.yml** playbook.

You need a Jinja2 expression that uses a lookup plug-in to render the template as a string, followed by a filter that converts from a YAML string to a data structure.

The correct **firewall_rules** variable definition is:

```
- name: Ensure Apache is deployed
  hosts: web_servers
  force_handlers: True
  gather_facts: no

  roles:
    # Use the apache_firewall_rules.yml.j2 template to
    # generate the firewall rules.
    - role: apache
      firewall_rules: "{{ lookup('template', 'apache_firewall_rules.yml.j2') |
        from_yaml }}"
```

**Note**

The `firewall_rules` variable is defined on a single line. To improve readability, use ad hoc variables to keep line length under 80 characters. The below variation is also valid:

```
- name: Ensure Apache is deployed
  hosts: web_servers
  force_handlers: True
  gather_facts: no

  roles:
    # Use the apache_firewall_rules.yml.j2 template to
    # generate the firewall rules.
    - role: apache
      firewall_rules: "{{ lookup('template', tfile) | from_yaml }}"
      tfile: apache_firewall_rules.yml.j2
```

- Execute the `site.yml` playbook to test your changes. If you make all of the necessary changes, then the playbook executes without any errors.
- Verify that requests to the load balancer on `servera` from `workstation` succeed, while direct requests to the backend web servers, `serverb` and `serverc`, from `workstation` are denied.
- When testing requests to backend servers, recall that the `apache_port` variable configures the port of the backend Apache servers, and it is set to a value of `8008`.

```
[student@workstation data-review]$ ansible-playbook site.yml
...output omitted...
[student@workstation data-review]$ echo $?
0
[student@workstation data-review]$ curl servera
This is serverb. (version v1.0)
[student@workstation data-review]$ curl servera
This is serverc. (version v1.0)
[student@workstation data-review]$ curl serverb
curl: (7) Failed to connect to serverb port 80: No route to host
[student@workstation data-review]$ curl serverc:8008
curl: (7) Failed to connect to serverc port 8008: No route to host
```

- Save, commit, and push your changes to the remote repository.

```
[student@workstation data-review]$ git status
On branch master
Your branch is up to date with 'origin/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:   deploy_apache.yml
        modified:   roles/firewall/tasks/main.yml
```

```
modified: templates/apache_firewall_rules.yml.j2

no changes added to commit (use "git add" and/or "git commit -a")
[student@workstation data-review]$ git add deploy_apache.yml \
> roles/firewall/tasks/main.yml templates/apache_firewall_rules.yml.j2
[student@workstation data-review]$ git commit \
> -m "Added Filters and Plugins"
...output omitted...
[student@workstation data-review]$ git push
```

Evaluation

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



Note

Make sure you commit and push your changes to the Git repository before rerunning the script.

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

Finish

From **workstation**, run the **lab data-review finish** command to complete this lab.

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

This concludes the lab.

Summary

In this chapter, you learned:

- Use filters inside Jinja2 expressions and templates to format, transform, and parse Ansible data. Filters are applied one after another in the same Jinja2 expression using a pipe (|) character.
- Lookup plug-ins allow you to interact with external sources to retrieve or enrich Ansible data.
- Lookup plug-ins and filters execute on the Ansible controller, not on a remote host. Some filters and plug-ins require the installation of additional packages on the controller.
- Use a combination of filters and lookup plug-ins to transform a complex data structure into a list for the **loop** keyword.
- To force the output of a lookup plug-in into a list, use the **query** instruction, instead of **lookup**.

Chapter 5

Coordinating Rolling Updates

Goal

Use advanced features of Ansible to manage rolling updates in order to minimize downtime, and to ensure maintainability and simplicity of Ansible Playbooks.

Objectives

- Run a task for a managed host on a different host, and control whether facts gathered by that task are delegated to the managed host or to the other host.
- Tune behavior of the serial directive when batching hosts for execution, abort the play if too many hosts fail, and create tasks that run only once for each batch or for all hosts in the inventory.

Sections

- Delegating Tasks and Facts (and Guided Exercise)
- Managing Rolling Updates (and Guided Exercise)

Lab

Coordinating Rolling Updates

Delegating Tasks and Facts

Objectives

After completing this section, you should be able to run a task for a managed host on a different host, and control whether facts gathered by that task are delegated to that host or the other host.

Delegating Tasks

Sometimes, when Ansible is running a play to ensure the correct configuration of a system, it might need to perform one or more tasks on a different system on the managed host's behalf. For example, you might need to log into a network device to change a DHCP configuration, or make sure certain groups exist in an Active Directory domain, or communicate with the API of a service using tools that are not available on the managed host.

In a play, you can *delegate* a task to run on a different host instead of the current managed host.

A task delegates the action to a host using the **delegate_to** directive. This directive points Ansible to the host that will execute the task in place of the corresponding target.

One of the most common places you might delegate a task is on **localhost**, the Ansible control node. For example, you might do this if you need to talk to an API for a service that can not be reached from the managed host for some reason, but can from the control node.

The following simple example runs the **uname -a** command on each host in the play, and then runs the **uname -a** command on **localhost** on behalf of each host in the play.

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

  tasks:
    - name: Get system information
      command: uname -a
      register: server

    - name: Display servera system information
      debug:
        msg: {{ server }}

    - name: Get system information
      command: uname -a
      delegate_to: localhost
      register: local

    - name: Display localhost system information
      debug:
        msg: {{ local }}
```

In the following practical example, the first task is delegated to each of the HAProxy load balancers in the Ansible group **lbervers** in turn, removing the managed host from all the load

balancers. Then, the second task, which is not delegated, stops the web server on the managed host. This task runs for each host in the play.

```
- name: Remove the server from HAProxy
  haproxy:
    state: disabled
    host: "{{ ansible_facts['fqdn'] }}"
    socket: /var/lib/haproxy/stats
  delegate_to: "{{ item }}"
  loop: "{{ groups['lbervers'] }}"

- name: Make sure Apache HTTPD is stopped
  service:
    name: httpd
    state: stopped
```

Delegating Facts

In the preceding example, the fact `ansible_facts['fqdn']` was used. The FQDN for the managed host is used, not `localhost`.

When you delegate a task, use the host variables and facts for the managed host (the current `inventory_hostname`) for which the task is running. So, if the task is running for `servera`, but has been delegated to `localhost`, then use the variables and facts for `servera`. This is usually exactly what you want.

However, sometimes you might want to assign the facts collected by a delegated task to the host to which the task was delegated. To change this setting, set the `delegate_facts` directive to `true`.

```
- hosts: localhost
gather_facts: no
tasks:
  - name: Set a fact in delegated task on servera
    set_fact:
      myfact: Where am I set?
    delegate_to: servera.lab.example.com
    delegate_facts: True

  - name: Display the facts from servera.lab.example.com
    debug:
      msg: "{{ hostvars['servera.lab.example.com']['myfact'] }}"
```

The preceding play runs for `localhost`, but the first task is delegated to `servera.lab.example.com`. Using the `delegate_facts` directive on that task instructs Ansible to gather facts into the `hostvars['servera.lab.example.com']` namespace for the delegated host, instead of the default `hostvars['localhost']` namespace for the current managed host.



References

[Delegation, Rolling Updates, and Local Actions – Ansible Documentation](https://docs.ansible.com/ansible/latest/user_guide/playbooks_delegation.html)

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

► Guided Exercise

Delegating Tasks and Facts

In this exercise you will run a playbook that includes a task that is delegated to another host.

Outcomes

You should be able to delegate a task to run on another host.

Before You Begin

Open a terminal on the **workstation** machine as the **student** user, and then run the following command:

```
[student@workstation ~]$ lab update-delegation start
```

- ▶ 1. Clone the Git repository `http://git.lab.example.com:8081/git/update-delegation.git` in the `/home/student/git-repos` directory, and change to the cloned project directory.
 - 1.1. From a terminal, create the directory `/home/student/git-repos` if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- 1.3. Clone the repository:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/update-delegation.git
Cloning into 'update-delegation'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 11 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (11/11), done.
[student@workstation update-delegation]$ cd update-delegation
```

- ▶ 2. Review the contents of the `inventory.yml` inventory file. Verify that `web_servers` host group contains servers A to F.

```
[student@workstation update-delegation]$ cat inventory.yml
web_servers:
  hosts:
    server[a:f].lab.example.com:
```

► 3. Add tasks to the **query_times.yml** playbook to:

- Query the time on each server in the **web_servers** host group.
 - Store the result in the **/tmp/times.txt** file on the **workstation** machine.
- 3.1. Open the **query_times.yml** file in an editor of your choice:

```
[student@workstation update-delegation]$ vi query_times.yml
```

- 3.2. Add a task to execute the **date** command on each server, which retrieves the current time. Register the result in a variable named **date**. Because the task does not change the state of the server, you must include **changed_when: false** in the task.

The task should display as follows:

```
- name: Take time on server
  shell: 'date'
  register: date
  changed_when: false
```

- 3.3. Add a task that adds a line of text for each server to the **/tmp/times.txt** file on **workstation**. Each line contains the name of a web server and the registered time for that server. Delegate this task to **localhost**. Ensure that each line of text appends to the **/tmp/times.txt** file.

The task displays as follows:

```
- name: Save times to localhost
  vars:
    record: "{{ inventory_hostname }} time: {{ date.stdout_lines[0] }}"
  shell: "echo '{{ record }}' >> /tmp/times.txt"
  delegate_to: localhost
```



Note

A **vars** section is not necessary for this task. A variable keeps each line of the task under 80 characters in length, making it easier to read.

The content of the **query_times.yml** file follows:

```
---
- name: Query server times and store them locally
  hosts: web_servers
  gather_facts: false

  tasks:
```

```

- name: Take time on server
  shell: 'date'
  register: date
  changed_when: false

- name: Save times to localhost
  vars:
    record: "{{ inventory_hostname }} time: {{ date.stdout_lines[0] }}"
  shell: "echo '{{ record }}' >> /tmp/times.txt"
  delegate_to: localhost

```

- ▶ 4. Run the playbook using the **ansible-playbook** command to test the delegation of tasks:

```
[student@workstation update-delegation]$ ansible-playbook query_times.yml

PLAY [Query server times and store them locally] ****

TASK [Take time on server] ****
ok: [servera.lab.example.com]
...output omitted...
ok: [serverf.lab.example.com]

TASK [Save times to localhost] ****
changed: [servera.lab.example.com]
...output omitted...
changed: [serverf.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 ...
serverc.lab.example.com : ok=2    changed=1    unreachable=0    failed=0 ...
serverd.lab.example.com : ok=2    changed=1    unreachable=0    failed=0 ...
servere.lab.example.com : ok=2    changed=1    unreachable=0    failed=0 ...
serverf.lab.example.com : ok=2    changed=1    unreachable=0    failed=0 ...
```

- ▶ 5. Verify that the playbook has gathered the times correctly, and that Ansible stored the information in the **workstation** machine:

```
[student@workstation update-delegation]$ cat /tmp/times.txt
servera.lab.example.com time: jue jun 6 16:52:55 EDT 2019
serverb.lab.example.com time: jue jun 6 16:52:55 EDT 2019
servere.lab.example.com time: jue jun 6 16:52:55 EDT 2019
serverc.lab.example.com time: jue jun 6 16:52:55 EDT 2019
serverd.lab.example.com time: jue jun 6 16:52:55 EDT 2019
serverf.lab.example.com time: jue jun 6 16:52:57 EDT 2019
```

The order of hosts in your file may differ.

Finish

On **workstation**, run the **lab update-delegation finish** script to complete this lab.

```
[student@workstation update-delegation]$ lab update-delegation finish
```

This concludes the exercise.

Managing Rolling Updates

Objectives

After completing this section, you should be able to tune behavior of the `serial` directive when batching hosts for execution, abort the play if too many hosts fail, and create tasks that run only once for each batch or for all hosts in the inventory.

Overview

Ansible has several features that enable *rolling updates*, a strategy that staggers deployments to batches of servers. With this strategy, infrastructure updates are deployed with zero downtime.

When an unforeseen problem occurs, Ansible can halt the deployment and any errors can be limited to only those servers in a particular batch. With tests and monitoring in place, you can configure playbook tasks to:

- Rollback configuration for hosts in the affected batch.
- Quarantine affected hosts to enable analysis of the failed deployment.
- Send notifications of the deployment to stakeholders.

Controlling Batch Size

By default, Ansible runs a play by running one task for all hosts in the play before starting execution of the next task. If a task fails, then all hosts will be only partway through the task. This could mean that none of your hosts are working correctly, which could lead to an outage.

Ideally, you could process some of your hosts all the way through the play before starting the next batch of hosts. If too many hosts fail, then you can abort the entire play. In the following paragraphs, you configure your play to do this.

Setting a Fixed Batch Size

To process hosts through a play in batches, use the `serial` keyword in your play. The `serial` keyword specifies how many hosts should be in each batch. Ansible will process each batch of hosts all the way through the play before starting the next batch. If all hosts in the current batch fail, the entire play is aborted and Ansible does not start the next batch.

Consider the beginning portion of the following play:

```
---
- name: Update Webservers
  hosts: web_servers
  serial: 2
```

In this example, the `serial` keyword instructs Ansible to process hosts in the `web_servers` host group in batches of two hosts. If the play executes without error, the play is repeated with a new batch.

This process continues until all the hosts in the play are processed. As a result, the last batch may contain fewer hosts than the indicated value of the **serial** keyword, if the total number of hosts in the play is not divisible by the batch size. In the previous example, the last batch contains one host if the total number of web servers is an odd value.

Remember, if you use an integer with the **serial** keyword, then as the number of hosts in the play increases, the number of batches needed to complete the play also increases. With a **serial** value of **2**, a host group with 200 hosts will require 100 batches to complete, while a host group of 20 hosts will require 10 batches to complete.

Setting Batch Size as a Percentage

You can also specify a percentage for the value of the **serial** keyword:

```
---
- name: Update Webservers
  hosts: web_servers
  serial: 25%
```

If you specify a percentage, that percentage of the hosts in the play will be processed in each batch. If the value of **serial** is 25%, then four batches are required to complete the play for all hosts, regardless of whether the **web_servers** group contains 20 hosts, or 200 hosts.

Ansible applies the percentage to the total number of hosts in the host group. If that resulting value is not an integer number of hosts, then the value is truncated (rounded down) to the nearest integer. The remaining hosts are run in a final, smaller batch. The batch size can not be zero hosts. If the truncated value is zero, Ansible changes the batch size to one host.

Description	Host Group 1	Host Group 2	Host Group 3
Number of Hosts	3	13	19
serial value	25%	25%	25%
Percentage Applied	0.75	3.25	4.25
Truncated Percentage	0	3	4
Batch Size	1	3	4
Number of Batches	3	5	5
Size of Last Batch	1	1	3

Setting Batch Sizes to Change

You can change the batch size as the play runs. For example, you could test a play on a batch of one host, and if that host fails, then the entire play aborts. However, if the play is successful on one host, you could increase the batch size to ten percent of your hosts, then fifty percent of the managed hosts, and then the remainder of the managed hosts.

You can gradually change the batch size by setting the **serial** keyword to a list of values. This list can contain any combination of integers and percentages, and resets the size of each batch in

sequence. If the value is a percentage, then Ansible computes the batch size based the total size of the host group, and not the size of the group of unprocessed hosts.

Consider the following example:

```
---  
- name: Update Webservers  
  hosts: web_servers  
  serial:  
    - 1  
    - 10%  
    - 100%
```

The first batch contains just a single host.

The second batch contains 10% of the total number of hosts in the **web_servers** host group. Ansible computes the actual value according to the previously discussed rules.

The third batch contains all the remaining unprocessed hosts in the play. This allows Ansible to efficiently process all the remaining hosts.

If there are unprocessed hosts remaining after the last batch corresponding to the last entry of the **serial** keyword, the last batch size is repeated until all hosts are processed. Consider the following play, which executes against a **web_servers** host group with 100 hosts:

```
---  
- name: Update Webservers  
  hosts: web_servers  
  serial:  
    - 1  
    - 10%  
    - 25%
```

The first batch contains one host, while the second batch contains 10 hosts (10% of 100). The third batch processes 25 hosts (25% of 100), leaving 64 unprocessed hosts ($1 + 10 + 25$ processed). Ansible continues executing the play in batch sizes of 25 hosts (25% of 100), until there are fewer than 25 unprocessed hosts remaining. In this example, the remaining 14 hosts are processed in the final batch ($1 + 10 + 25 + 25 + 25 + 14 = 100$).

Aborting the Play

Now that you know how to gradually increase the batch size, you must learn how to abort the play if too many hosts fail.

By default, Ansible tries to get as many hosts to complete a play as possible. If a task fails for a host, then it is dropped from the play, but Ansible will continue to run the remaining tasks in the play for other hosts. The play only stops if all hosts fail.

However, if you organize hosts into batches using the **serial** keyword, then if all hosts in the current batch fail, Ansible will stop the play for *all remaining hosts*, not just those remaining hosts in the current batch. If the execution of the play is stopped due to a failure of all hosts in a batch, then the next batch will not be started.

Ansible keeps a list of the active servers for each batch in the **ansible_play_batch** variable. Any host that fails a task is removed from the **ansible_play_batch** list. Ansible updates this list after every task.

Consider the following hypothetical playbook, which executes against a **web_servers** host group with 100 hosts:

```
---
- name: Update Webservers
  hosts: web_servers
  tasks:
    - name: Step One
      shell: /usr/bin/some_command
    - name: Step Two
      shell: /usr/bin/some_other_command
```

If ninety-nine of the web servers fail the first task, but one host succeeds, Ansible continues to the second task. When Ansible executes the second task, Ansible only executes the task for the one host that previously succeeded.

If you use the **serial** keyword, playbook execution continues only as long as there are hosts remaining in the current batch with no failures. Consider this modification to the hypothetical playbook:

```
---
- name: Update Webservers
  hosts: web_servers
  serial: 2
  tasks:
    - name: Step One
      shell: /usr/bin/some_command
    - name: Step Two
      shell: /usr/bin/some_other_command
```

If the first batch of two contains a host that succeeds and a host that fails, then the batch completes and Ansible moves on to the second batch of two. If both hosts in the second batch fail on a task in the play, then Ansible aborts the entire play and starts no more batches.

In this scenario, after the playbook executes:

- One host successfully completes the play.
- Three hosts might be in an error state.
- The rest of the hosts remain unaltered.

Specifying Failure Tolerance

By default, Ansible only halts play execution when all hosts in a batch experience a failure. However, you might want a play to abort if more than a certain percentage of hosts in the inventory have failed, even if no entire batch has failed. It is also possible to "fail fast" and abort the play if any tasks fail.

Alter Ansible's failure behavior by adding the **max_fail_percentage** keyword to a play. When the number of failed hosts in a batch exceed this percentage, Ansible halts playbook execution.

Consider the following hypothetical playbook, which executes against the `web_servers` host group that contains 100 hosts:

```
---
- name: Update Webservers
  hosts: web_servers
  max_fail_percentage: 30%
  serial:
    - 2
    - 10%
    - 100%
  tasks:
    - name: Step One
      shell: /usr/bin/some_command
    - name: Step Two
      shell: /usr/bin/some_other_command
```

The first batch contains two hosts. Because 30% of 2 is 0.6, a single host failure causes execution to stop.

If both hosts in the first batch succeed, then Ansible continues with the second batch of 10 hosts. Because 30% of 10 is 3, more than 3 host failures must occur to cause Ansible to stop playbook execution. If three or fewer hosts experience errors in the second batch, Ansible continues with the third batch.

To implement a "fail fast" strategy, set the `max_fail_percentage` to a value of zero.



Important

To summarize Ansible's failure behavior:

- If the `serial` keyword and the `max_fail_percentage` value are not set, all hosts are run through the play in one batch. If all hosts fail, then the play fails.
- If the `serial` keyword is set, then hosts are run through the play in multiple batches and the play fails if all hosts in any one batch fail.
- If `max_fail_percentage` keyword is set, then the play fails if more than that percentage of hosts in a batch fail.

If a play fails, all remaining plays in the playbook are aborted.

Running a Task Once

In certain scenarios, you may only need to run a task once for an entire batch of hosts, rather than once for each host in the batch. To do so, add the `run_once` keyword to the task with a Boolean `true` (or `yes`) value.

Consider the following hypothetical task:

```
- name: Reactivate Hosts
  shell: /sbin/activate.sh {{ active_hosts_string }}
  run_once: yes
  delegate_to: monitor.example.com
  vars:
    active_hosts_string: "{{ ansible_play_batch | join(' ')}}"
```

This task runs once and executes on the **monitor.example.com** host. The task uses the **active_hosts_string** variable to pass a list of active hosts as command-line arguments to an activation script. The variable contains only those hosts in the current batch that have succeeded for all previous tasks.



Note

Setting the **run_once: yes** keyword causes a task to run once for each batch. If you only need to run a task once for all hosts in a play, and the play has multiple batches, then you can add the following conditional to the task:

```
when: inventory_hostname == ansible_play_hosts[0]
```

This conditional runs the task only for the first host in the play.



References

Delegation, Rolling Updates, and Local Actions – Ansible Documentation

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

► Guided Exercise

Managing Rolling Updates

In this exercise, you will run a playbook that uses unequal batch sizes with the **serial** keyword, aborts if too many hosts fail, and runs a specific task once per batch.

Outcomes

You should be able to control the update process of an existing **haproxy** cluster by controlling the update with the **serial** directive, which determines the size of the batch to use.

Before You Begin

Log in as the **student** user on the **workstation** machine, and run the **lab update-management start** command.

```
[student@workstation ~]$ lab update-management start
```

- ▶ 1. Clone the Git repository <http://git.lab.example.com:8081/git/update-management.git> in the **/home/student/git-repos** directory.
 - 1.1. From a terminal, create the directory **/home/student/git-repos**, if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- 1.3. Clone the repository:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/update-management.git
Cloning into 'update-management'...
...output omitted...
[student@workstation git-repos]$ cd update-management
[student@workstation update-management]$
```

- ▶ 2. Execute the **site.yml** playbook to deploy a front-end load balancer and a set of back-end web servers.

```
[student@workstation update-management]$ ansible-playbook site.yml
...output omitted...
PLAY RECAP ****server.lab.example.com : ok=7    changed=6    ... failed=0   skipped=2  ...
```

```
serverb.lab.example.com : ok=16    changed=11    ... failed=0   skipped=2 ...
serverc.lab.example.com : ok=14    changed=9     ... failed=0   skipped=2 ...
serverd.lab.example.com : ok=14    changed=9     ... failed=0   skipped=2 ...
servere.lab.example.com : ok=14    changed=9     ... failed=0   skipped=2 ...
serverf.lab.example.com : ok=14    changed=9     ... failed=0   skipped=2 ...
```

► 3. Review the **issue_requests.sh** script.

In a different terminal tab, execute the **issue_requests.sh** script. The script continually sends periodic requests to the load balancer, displaying the response. The script output indicates five servers are providing version **v1.0** of the web application.

You stop the execution of this script in a later step.

3.1. Review the **issue_requests.sh** script:

```
#!/bin/bash
SERVER=servera.lab.example.com
WAIT_TIME_SECS=0.5
LOG_FILE=curl_output.log

rm -f $LOG_FILE

while true; do
#Print curl response to stdout and also write to log file.
curl -s $SERVER | tee -a $LOG_FILE

sleep $WAIT_TIME_SECS
done
```

The script executes a **curl servera.lab.example.com** command every 0.5 seconds, and logs the response to the terminal screen and to the **curl_output.log** log file.

3.2. In a different terminal, make the **issue_requests.sh** script executable and execute it:

```
[student@workstation ~]$ cd ~/git-repos/update-management
[student@workstation update-management]$ chmod +x issue_requests.sh
[student@workstation update-management]$ ./issue_requests.sh
This is serverb. (version v1.0)
This is serverc. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverf. (version v1.0)
This is serverb. (version v1.0)
This is serverc. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverf. (version v1.0)
...output omitted...
```

The load balancer rotates requests to all five web servers, and back-end servers are serving version **v1.0** of the web application.

- 4. The **update_webapp.yml** playbook performs a rolling update of the web application hosted on back-end web servers. Review the **update_webapp.yml** playbook and the use of the **serial** keyword.

4.1. Review the top section of the playbook:

```
---
- hosts: web_servers
  serial:
    - 1
    - 25%
    - 100%
```

The **update_webapp.yml** playbook acts on all hosts in the **web_servers** host group. The **serial** keyword specifies that tasks in the play are processed in three batches.

The first batch contains 1 host. After this batch finishes, four hosts still require processing.

The second batch contains 1 host, because 25% of the host group size is 1.25 and that truncates to 1.

The third batch contains 3 hosts, because 100% of the hosts group size is 5 but only 3 hosts remain unprocessed.

4.2. Review the **pre_tasks** section of the play:

```
pre_tasks:
  - name: Remove web server from service during the update
    haproxy:
      state: disabled
      backend: app
      host: "{{ inventory_hostname }}"
      delegate_to: "{{ item }}"
      with_items: "{{ groups['lb_servers'] }}"
```

Before the playbook updates the web application on each server, the **haproxy** module disables the server in all load balancers. With this task, external clients are not exposed to errors discovered after application deployment.

4.3. Review the **roles** section of the play:

```
roles:
  - role: webapp
```

The **webapp** role deploys a web application that is hosted in a Git repository. The **webapp_repo** variable specifies the URL of the web application Git repository. The **webapp_version** variable specifies a branch or version tag in the application's repository. These variables are defined in the **group_vars/web_servers/webapp.yml** file.

```
[student@workstation update-management]$ cat group_vars/web_servers/webapp.yml
```

```
webapp_repo: http://git.lab.example.com:8081/git/simple-webapp.git
webapp_version: v1.0
```

- 4.4. Review the **post_tasks** section of the play:

```
post_tasks:
  # Firewall rules dictate that requests to backend web
  # servers must originate from a load balancer.
  - name: Smoke Test - Ensure HTTP 200 OK
    uri:
      url: "http://{{ inventory_hostname }}:{{ apache_port }}"
      status_code: 200
      delegate_to: "{{ groups['lb_servers'][0] }}"
      become: no

  # If the test fails, servers are not re-enabled
  # in the load balancers, and the update process halts.
  - name: Enable healthy server in load balancers
    haproxy:
      state: enabled
      backend: app
      host: "{{ inventory_hostname }}"
    delegate_to: "{{ item }}"
    with_items: "{{ groups['lb_servers'] }}"
```

After the playbook deploys the web application, a smoke test ensures that each back-end web server responds with a **200** HTTP status code. The firewall rules on each web server only allow web requests from a load balancer, so all smoke tests are delegated to a load balancer.

If the smoke test fails for a server, then further processing of that server halts and the web server is not re-enabled in the load balancer.

When the smoke test passes, the second task enables the server in the load balancer.

- 5. Use the **update_webapp.yml** playbook to deploy version **v1.1.0** of the web application. This version of the web application is a PHP application, instead of basic HTML pages.

As the playbook executes, monitor the output of the **issue_requests.sh** script in the other terminal tab. Deployment of the web application fails, but the load balancer continues to provide access to the web application service.

- 5.1. Execute the **update_webapp.yml** playbook with the **-e webapp_version=v1.1.0** option:

```
[student@workstation update-management]$ ansible-playbook update_webapp.yml \
> -e webapp_version=v1.1.0

PLAY [web_servers] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
...output omitted...
```

The playbook will fail, but do not wait for it to finish and proceed to the next step.

- 5.2. Monitor the output of the **issue_requests.sh** script during the execution of the **update_webapp.yml** playbook.

```
This is serverf. (version v1.0)
This is serverb. (version v1.0)
This is serverc. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverf. (version v1.0)
This is serverc. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverf. (version v1.0)
This is serverc. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverf. (version v1.0)
...output omitted...
```

From an external client perspective, the load balancer continues to provide responses from version **v1.0** of the web application. However, responses from one of the servers, **serverb**, stop appearing in the output.

- 5.3. Return to the output from the execution of the **update_webapp.yml** playbook. Verify that the smoke test fails for **serverb** when you deploy version **v1.1.0** of the web application.

```
...output omitted...
TASK [Smoke Test - Ensure HTTP 200 OK] ****
fatal: [serverb.lab.example.com]: FAILED! => {"changed": false, "connection": "close", "content": "", "content_length": "0", "content_type": "text/html; charset=UTF-8", "date": "Fri, 31 May 2019 04:17:04 GMT", "elapsed": 0, "msg": "Status code was 500 and not [200]: HTTP Error 500: Internal Server Error", "redirected": false, "server": "Apache/2.4.37 (Red Hat Enterprise Linux)", "status": 500, "url": "http://serverb.lab.example.com:8008", "x_powered_by": "PHP/7.2.11"}
```

```
PLAY RECAP ****
serverb.lab.example.com : ok=9    changed=5    unreachable=0    failed=1 ...
```

The new application deployment results in a **500** HTTP status code, known as an **Internal Server Error**.

- 5.4. *Optional.* Clone the web application source code repository at `http://git.lab.example.com:8081/git/simple-webapp.git` in the `/home/student/git-repos` directory. Check out version **v1.1.0** of the application, and determine the cause of the error:

```
[student@workstation update-management]$ cd ~/git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/simple-webapp.git
[student@workstation git-repos]$ cd simple-webapp
[student@workstation simple-webapp]$ git checkout v1.1.0
...output omitted...
[student@workstation simple-webapp]$ ls
index.php
```

This version of the application uses PHP. The PHP code contains syntax errors:

```
[student@workstation simple-webapp]$ cat index.php
<?php
This is {{ ansible_hostname }}. (version {{ webapp_version}})
?>
```

Do not fix the code. The correct PHP code is:

```
<?php
echo "This is {{ ansible_hostname }}. (version {{ webapp_version}})\n";
?>
```

Return to the `~/git-repos/update-management` directory:

```
[student@workstation simple-webapp]$ cd ~/git-repos
[student@workstation git-repos]$ cd update-management
[student@workstation update-management]$
```

- ▶ 6. After diagnosing the deployment errors with version **v1.1.0**, the web development team pushes a fix for the web application.

Use the `update_webapp.yml` playbook to deploy the patched web application, version **v1.1.1**.

As the playbook executes, monitor the output of the `issue_requests.sh` script in the other terminal tab. The output indicates that the back-end servers gradually switch over to the new version of the application.

- 6.1. Execute the `update_webapp.yml` playbook with the `-e webapp_version=v1.1.1` option:

```
[student@workstation update-management]$ ansible-playbook update_webapp.yml \
> -e webapp_version=v1.1.1

PLAY [web_servers] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
...output omitted...
```

Do not wait for the playbook to complete; proceed to the next step.

- 6.2. Monitor the output of the `issue_requests.sh` script during the execution of the `update_webapp.yml` playbook. Verify that the playbook removes the web servers from the load balancer service in batches defined with the `serial` keyword, upgrades the web application, and returns the web servers to the load balancer.

Initially, responses cycle through the 4 hosts remaining in service after the failed upgrade to version **v1.1.0**.

```
...output omitted...
This is serverc. (version v1.0)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
```

```
This is servere. (version v1.0)
This is serverc. (version v1.0)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
...output omitted...
```

Eventually, the smoke test passes for the new application and **serverb** is put back into service with an updated version:

```
...output omitted...
This is serverc. (version v1.0)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverc. (version v1.0)
This is serverb. (version v1.1.1)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
...output omitted...
```

The playbook processes the next batch, which only contains **serverc**.

The playbook removes **serverc** from service:

```
...output omitted...
This is servere. (version v1.0)
This is serverb. (version v1.1.1)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
This is serverb. (version v1.1.1)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
...output omitted...
```

Eventually, the smoke test passes for **serverc** and the server is put back into service:

```
...output omitted...
This is servere. (version v1.0)
This is serverb. (version v1.1.1)
This is serverf. (version v1.0)
This is serverd. (version v1.0)
This is servere. (version v1.0)
```

```
This is serverc. (version v1.1.1)
This is serverb. (version v1.1.1)
This is serverf. (version v1.0)
...output omitted...
```

The last batch processes all remaining web servers. The playbook first disables all three of these servers, leaving only **serverb** and **serverc** to handle requests:

```
...output omitted...
This is serverf. (version v1.0)
This is serverd. (version v1.0)
This is serverc. (version v1.1.1)
This is serverb. (version v1.1.1)
This is serverc. (version v1.1.1)
This is serverb. (version v1.1.1)
This is serverc. (version v1.1.1)
This is serverb. (version v1.1.1)
...output omitted...
```

Eventually, each server passes the smoke test and is put back into service:

```
...output omitted...
This is serverb. (version v1.1.1)
This is servere. (version v1.1.1)
This is serverf. (version v1.1.1)
This is serverd. (version v1.1.1)
This is serverc. (version v1.1.1)
This is serverb. (version v1.1.1)
This is servere. (version v1.1.1)
This is serverf. (version v1.1.1)
This is serverd. (version v1.1.1)
This is serverc. (version v1.1.1)
...output omitted...
```

6.3. Press **Ctrl+C** to stop the execution of the **issue_requests.sh** script.

```
This is servere. (version v1.1.1)
This is serverf. (version v1.1.1)
This is serverd. (version v1.1.1)
This is serverc. (version v1.1.1)
^C
[student@workstation update-management]$
```

- ▶ 7. Return to the terminal that contains the playbook output. Verify that the playbook completes without any errors.

```
...output omitted...
TASK [Smoke Test - Ensure HTTP 200 OK] ****
ok: [serverd.lab.example.com]
ok: [servere.lab.example.com]
ok: [serverf.lab.example.com]
```

```
TASK [Enable healthy server in load balancers] ****
changed: [serverd.lab.example.com] => (item=servera.lab.example.com)
changed: [servere.lab.example.com] => (item=servera.lab.example.com)
changed: [serverf.lab.example.com] => (item=servera.lab.example.com)

PLAY RECAP ****
serverb.lab.example.com : ok=10  changed=4  unreachable=0  failed=0  ...
serverc.lab.example.com : ok=11  changed=6  unreachable=0  failed=0  ...
serverd.lab.example.com : ok=11  changed=6  unreachable=0  failed=0  ...
servere.lab.example.com : ok=9   changed=4  unreachable=0  failed=0  ...
serverf.lab.example.com : ok=9   changed=4  unreachable=0  failed=0  ...
```

Finish

On the **workstation** machine, run the **lab update-management finish** script to clean up and finish this exercise.

```
[student@workstation ~]$ lab update-management finish
```

This concludes the guided exercise.

▶ Lab

Coordinating Rolling Updates

Performance Checklist

In this lab, you will modify a playbook to implement Rolling Updates.

Outcomes

You should be able to delegate tasks to other hosts, implement deployment batches with the `serial` keyword, and limit failures with the `max_fail_percentage` keyword.

Before You Begin

Log in as the `student` user on the `workstation` machine, and run `lab update-review start`.

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

This script initializes the remote Git repository you need for this lab: `http://git.lab.example.com:8081/git/update-review.git`. The Git repository contains playbooks that configure a front-end load balancer and a pool of back-end web servers.

1. Clone the repository. Execute the `site.yml` playbook to deploy the web application infrastructure.
 2. Add a `pre_task` task to disable the web servers from the `HAProxy` load balancer to the `update_webapp.yml` playbook. Without this task, external clients may experience problems due unforeseen deployment issues with the web application.

Configure the new task as follows:

 - Use the `haproxy` module.
 - Disable the host, using the `inventory_hostname` variable in the `app` back end.
 - Delegate each action to the only load balancer. The `{{ groups['lb_servers'][0] }}` Jinja2 expression provides the name of this load balancer.
 3. In the `update_webapp.yml` playbook, the first task in the `post_tasks` section is a smoke test. That task verifies that the web application on each web server responds to requests that originate from that same server. This is not a realistic test, as requests to the web server only originate from the load balancer. If you only test the response of a web server from the web server itself, you do not test any network-related functions.

Using delegation, modify the smoke test task so that each request to a web server originates from the load balancer. Use the `inventory_hostname` to connect to each web server.

 4. Add a `post_task` task after the smoke test to re-enable each web server in the `HAProxy` load balancer. This task is similar in structure to the `haproxy` task in the `pre_tasks` section.
 5. Configure the `update_webapp.yml` playbook to use batches, to mitigate the effects of unforeseen deployment errors. Ensure that the playbook uses no more than 3 batches to

complete the upgrade of all web server hosts. Set the first batch to contain 5% of the hosts, and the second batch 35% of the hosts.

Add an appropriate keyword to the play to ensure that playbook execution stops if any host fails a task during the upgrade.

6. Run the **update_webapp.yml** playbook to perform the rolling update.
7. Commit and push your changes back to the remote Git repository.

Evaluation

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



Note

Make sure to commit and push your changes to the Git repository before rerunning the script.

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

Finish

From the **workstation** machine, run the **lab update-review finish** command to complete this lab.

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

This concludes the lab.

► Solution

Coordinating Rolling Updates

Performance Checklist

In this lab, you will modify a playbook to implement Rolling Updates.

Outcomes

You should be able to delegate tasks to other hosts, implement deployment batches with the `serial` keyword, and limit failures with the `max_fail_percentage` keyword.

Before You Begin

Log in as the `student` user on the `workstation` machine, and run `lab update-review start`.

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

This script initializes the remote Git repository you need for this lab: `http://git.lab.example.com:8081/git/update-review.git`. The Git repository contains playbooks that configure a front-end load balancer and a pool of back-end web servers.

- Clone the repository. Execute the `site.yml` playbook to deploy the web application infrastructure.
 - From a terminal, create the directory `/home/student/git-repos` if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- Change to this directory:

```
[student@workstation ~]$ cd /home/student/git-repos/
```

- Clone the repository:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/update-review.git
Cloning into 'update-review'...
remote: Enumerating objects: 85, done.
remote: Counting objects: 100% (85/85), done.
remote: Compressing objects: 100% (58/58), done.
remote: Total 85 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (85/85), done.
[student@workstation update-review]$ cd update-review
```

- Use the `ansible-playbook` command to run the `site.yml` playbook.

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

PLAY [Ensure HAProxy is deployed] ****
...output omitted...

PLAY RECAP ****
servera.lab.example.com : ok=27   changed=14   unreachable=0    failed=0 ...
serverb.lab.example.com : ok=16    changed=8    unreachable=0    failed=0 ...
serverc.lab.example.com : ok=16    changed=8    unreachable=0    failed=0 ...
serverd.lab.example.com : ok=16    changed=8    unreachable=0    failed=0 ...
servere.lab.example.com : ok=16    changed=8    unreachable=0    failed=0 ...
serverf.lab.example.com : ok=16    changed=8    unreachable=0    failed=0 ...
```

2. Add a **pre_task** task to disable the web servers from the **HAProxy** load balancer to the **update_webapp.yml** playbook. Without this task, external clients may experience problems due unforeseen deployment issues with the web application.

Configure the new task as follows:

- Use the **haproxy** module.
- Disable the host, using the **inventory_hostname** variable in the **app** back end.
- Delegate each action to the only load balancer. The **{{ groups['lb_servers'][0] }}** Jinja2 expression provides the name of this load balancer.

- 2.1. Edit the **update_webapp.yml** file with your favorite editor:

```
[student@workstation update-review]$ vi update_webapp.yml
```

Add the **pre_task** task to disable the web server in the load balancer:

```
pre_tasks:
  - name: Remove web server from service during the update
    haproxy:
      state: disabled
      backend: app
      host: "{{ inventory_hostname }}"
    delegate_to: "{{ groups['lb_servers'][0] }}"
```

3. In the **update_webapp.yml** playbook, the first task in the **post_tasks** section is a smoke test. That task verifies that the web application on each web server responds to requests that originate from that same server. This is not a realistic test, as requests to the web server only originate from the load balancer. If you only test the response of a web server from the web server itself, you do not test any network-related functions.

Using delegation, modify the smoke test task so that each request to a web server originates from the load balancer. Use the **inventory_hostname** to connect to each web server.

- 3.1. Add the directive **delegate_to** to the smoke test task pointing to the load balancer. Change the testing URL to point to the **inventory_hostname**:

```
- name: Smoke Test - Ensure HTTP 200 OK
  uri:
    url: "http://{{ inventory_hostname }}:{{ apache_port }}"
    status_code: 200
  become: no
  delegate_to: "{{ groups['lb_servers'][0] }}"
```

4. Add a **post_task** task after the smoke test to re-enable each web server in the **HAProxy** load balancer. This task is similar in structure to the **haproxy** task in the **pre_tasks** section.

- 4.1. Add the task after the smoke test with the **state: enabled** directive.

```
- name: Enable healthy server in load balancers
  haproxy:
    state: enabled
    backend: app
    host: "{{ inventory_hostname }}"
  delegate_to: "{{ groups['lb_servers'][0] }}"
```

5. Configure the **update_webapp.yml** playbook to use batches, to mitigate the effects of unforeseen deployment errors. Ensure that the playbook uses no more than 3 batches to complete the upgrade of all web server hosts. Set the first batch to contain 5% of the hosts, and the second batch 35% of the hosts.

Add an appropriate keyword to the play to ensure that playbook execution stops if any host fails a task during the upgrade.

- 5.1. Add the **max_fail_percentage** directive and set the value to 0, to stop the execution at any failure. Add the **serial** directive and the value to a list of three elements: 5%, 35% and 100% to ensure that all servers are updated in the last batch. The beginning of the play should look like:

```
- name: Upgrade Web Application
  hosts: web_servers
  max_fail_percentage: 0
  serial:
    - 5%
    - 35%
    - 100%
```

6. Run the **update_webapp.yml** playbook to perform the rolling update.

- 6.1. Use the **ansible-playbook** command to run the **update_webapp.yml** playbook.

```
[student@workstation update-review]$ ansible-playbook update_webapp.yml
...output omitted...

PLAY RECAP ****
servera.lab.example.com : ok=10    changed=4      unreachable=0      failed=0 ...
serverb.lab.example.com : ok=10    changed=4      unreachable=0      failed=0 ...
serverc.lab.example.com : ok=8     changed=2      unreachable=0      failed=0 ...
```

```
serverd.lab.example.com    : ok=10    changed=4    unreachable=0    failed=0 ...
servere.lab.example.com    : ok=8     changed=2    unreachable=0    failed=0 ...
serverf.lab.example.com    : ok=8     changed=2    unreachable=0    failed=0 ...
```

7. Commit and push your changes back to the remote Git repository.

- 7.1. Add the changed files, commit the changes, and push them to the Git repository.

```
[student@workstation update-review]$ git add .
[student@workstation update-review]$ git commit -m "Rolling updates"
[master 8d38803] Rolling updates
 1 files changed, 26 insertions(+), 12 deletions(-)
[student@workstation update-review]$ git push
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 931 bytes | 931.00 KiB/s, done.
Total 7 (delta 3), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/update-review.git
  a36da15..5feb08e master -> master
```

Evaluation

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



Note

Make sure to commit and push your changes to the Git repository before rerunning the script.

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

Finish

From the **workstation** machine, run the **lab update-review finish** command to complete this lab.

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

This concludes the lab.

Summary

In this chapter, you learned:

- Delegating a task makes Ansible run a task on a different host than the current host, as specified by the **inventory_hostname** variable.
- When a task is delegated, any facts that are gathered by the delegated host are assigned to the current host for the task, as specified by the **inventory_hostname** variable.
- Use the **serial** keyword to run hosts through a play in multiple batches.
- Use the **max_fail_percentage** keyword to abort the play if more than a certain percentage of hosts in the current batch fail.
- Use the **run_once** keyword to run a task once for an entire batch rather than once for each host in the batch.

Chapter 6

Installing and Accessing Ansible Tower

Goal

Explain what Red Hat Ansible Tower is and demonstrate a basic ability to navigate and use its web user interface.

Objectives

- Describe the architecture, use cases, and installation requirements of Red Hat 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, you 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, there are often 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 shared access. Additionally, although playbooks may allow for the delegation of complex tasks, executing those tasks 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. The 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, Red Hat 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.

Support for Red Hat Ansible Tower is included along with Red Hat Ansible Engine support as part of the Red Hat Ansible Automation subscription. More information is available at <https://www.redhat.com/en/technologies/management/ansible>.

Red Hat Ansible Tower Architecture

Red Hat Ansible Tower is a Django web application designed to run on a Linux server as an on-premise, self-hosted solution that layers on top of an enterprise's existing Ansible infrastructure.

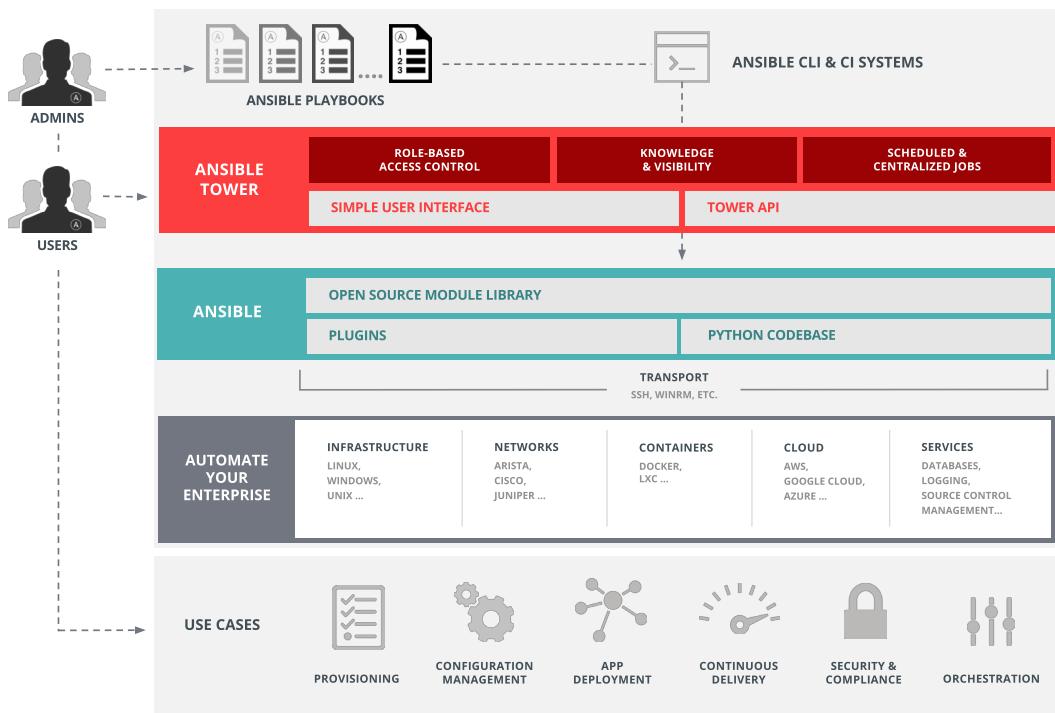


Figure 6.1: Ansible Tower architecture

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 that performs 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 then 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 differs from the installation procedure for 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

The following are some of the many features offered by Red Hat Ansible Tower for controlling, securing, and managing Ansible in an enterprise environment:

Visual Dashboard

The Ansible Tower web UI displays a Dashboard that 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. This system simplifies the delegation of user access to Ansible Tower objects, such as Organizations, Projects, and Inventories.

Graphical Inventory Management

Use the Ansible Tower web UI to create inventory groups and add inventory hosts. 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

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 routines that are easy to use.

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. Ansible Tower 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 job executions succeed or fail. Ansible Tower can deliver notifications using many applications, including email, Slack, and HipChat.

Multi-playbook Workflows

Complex operations often involve the serial execution of multiple playbooks. Ansible Tower multi-playbook 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 multi-playbook 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 Red Hat 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 Red Hat Ansible Tower? (Choose two)
 - a. Playbook development
 - b. Remote execution
 - c. Multiplay workflows
 - d. Monitoring
 - e. Version control
 - f. Graphical inventory management

- ▶ 3. Which two of the following access methods are supported in Red Hat 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 Red Hat 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

► 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 Red Hat 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 Red Hat Ansible Tower? (Choose two)

- a. Playbook development
- b. Remote execution
- c. Multiplay workflows
- d. Monitoring
- e. Version control
- f. Graphical inventory management

► 3. Which two of the following access methods are supported in Red Hat 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 Red Hat 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

Installing Red Hat Ansible Tower

Objectives

After completing this section, you should be able to install Red Hat Ansible Tower in a single-server configuration.

Installation Requirements

Red Hat Ansible Tower can be installed and is supported on 64-bit x86_64 versions of Red Hat Enterprise Linux and CentOS. The following are the requirements for installing Ansible Tower on a Red Hat Enterprise Linux 8 system. Details may vary slightly for other supported operating systems.

Operating System

For Red Hat Enterprise Linux installations, the Red Hat Ansible Tower 3.5 server is supported on systems running Red Hat Enterprise Linux 8.0 or later, or Red Hat Enterprise Linux 7.4 or later, on the 64-bit x86_64 processor architecture.

Web Browser

The current supported version of Mozilla Firefox or Google Chrome is required for connecting to the Red Hat 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 Red Hat Ansible Tower host.

The actual memory requirement depends on the maximum number of hosts that Red Hat 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 Red Hat Ansible Tower services.

Disk Storage

At least 20 GB of hard disk space is required for Red Hat Ansible Tower, and 10 GB of this space must be available to the **/var** directory.



Note

If you are installing in an Amazon EC2 instance, then use at least the **m4.1.large** instance type; use **m4.xlarge** if you are managing more than 100 hosts.

Red Hat Ansible Engine

Installation of Red Hat Ansible Tower is performed by executing a shell script that runs an Ansible Playbook. The current installation process automatically attempts to install Red Hat Ansible Engine and its dependencies if they are not already present.

For Red Hat Ansible Tower to work correctly, the latest stable version of Red Hat Ansible Engine should be installed using your distribution's package manager (such as **yum**). The installation is ideally performed by the Red Hat Ansible Tower installation program.

SELinux

Red Hat 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 Red Hat 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 Red Hat Ansible Tower server.



Note

If you are deploying Red Hat Ansible Tower as a pod to a Red Hat OpenShift cluster, then 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 Red Hat 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>.

Red Hat Ansible Tower is included as part of the simplified Red Hat Ansible Automation Platform subscription. This subscription includes support for Red Hat Ansible Tower and Red Hat Ansible Engine, access to SaaS-based Automation Analytics, support for use cases related to network devices and servers, access to supported Ansible Content Collections through the Automation Hub, and more.

Licenses for Ansible Tower Components

Red Hat Ansible Tower makes use of various software components, some of which may be provided under varying open source licenses. Licenses for each of these specific components are provided under the **/usr/share/doc/ansible-tower** directory.

Red Hat 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 Red Hat Ansible Tower 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 bundled installer for RHEL 8 is available at <http://releases.ansible.com/ansible-tower/setup-bundle/ansible-tower-setup-bundle-latest.el8.tar.gz>. This archive includes an initial set of RPM packages for Red Hat Ansible Tower for RHEL 8, so that it may be

Chapter 6 | Installing and Accessing Ansible Tower

installed on systems disconnected from the internet. Those systems may need to get software packages for Red Hat Enterprise Linux 8 channel from reachable sources. Administrators in high security environments may prefer to limit access to online package repositories. A RHEL 7 bundled installer is also available.

Installing Ansible Tower

The following procedure applies to the bundled installer used to install Ansible Tower on a single Red Hat Enterprise Linux 8.0 or later system. The subsequent exercise provides additional details.

1. As the **root** user, download the Ansible Tower setup bundle.
2. Extract the Ansible Tower setup bundle and then change to the directory containing the extracted contents.

```
[root@towerhost ~]# tar xzf ansible-tower-setup-bundle-latest.el8.tar.gz  
[root@towerhost ~]# cd ansible-tower-setup-bundle-3.5.0-1.el8
```

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.

4. Run the **setup.sh** script to start the Ansible Tower installer.

```
[root@towerhost ansible-tower-setup-bundle-3.5.0-1.el8]# ./setup.sh
[warn] Will install bundled Ansible
...output omitted...
The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2019-05-27-10:52:44.log
```

5. After 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.
6. 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.
7. When logging 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 display. The next section provides a more detailed orientation to the Ansible Tower interface.



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 start** to configure **tower.lab.example.com** with a Yum repository containing package dependencies from the Red Hat Enterprise Linux and Extras repositories, as required for Ansible Tower installation.

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

- 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 --remote-name** command to download the Ansible Tower setup bundle.

```
[root@tower ~]# curl --remote-name \
> http://content.example.com/ansible2.8/x86_64/dvd/setup-bundle/ansible-tower-
setup-bundle-latest.el8.tar.gz
```

- 2. Extract the Ansible Tower setup bundle. Change to the directory that contains the extracted contents.

```
[root@tower ~]# tar xzf ansible-tower-setup-bundle-latest.el8.tar.gz
[root@tower ~]# cd ansible-tower-setup-bundle-3.5.0-1.el8
```

- 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-3.5.0-1.el8]# 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-3.5.0-1.el8]# ./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. After the installer finishes successfully, exit the console session on the **tower** system.

```
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]# 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 logging in to the Ansible Tower web UI for the first time, you must 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 downloaded in the preceding step.
 - 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, you should be able to navigate and describe the Red Hat Ansible Tower web UI, and successfully launch a job using the provided job template, project, credentials, and inventory.

Using the Ansible Tower Web User Interface

This section provides an overview of how to use the Ansible Tower web UI to launch a job using 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.

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, the Ansible Tower Dashboard displays. The Dashboard is the main control center for Ansible Tower.

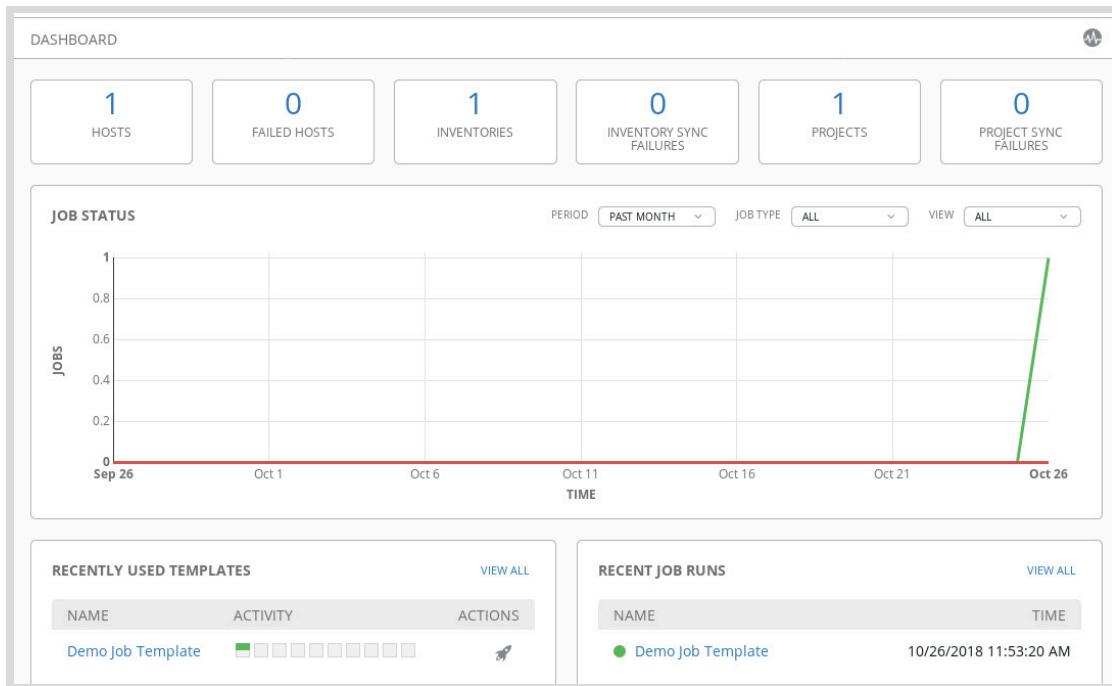


Figure 6.2: The Ansible Tower Dashboard

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. 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. Mouse over a colored square to display the job ID number and when it was run. Click a square to open the Job Details view for that job. Monitoring job template activity is discussed in more detail later in this section.
- Under the **ACTIONS** column are controls for using the job template.
- Click the **VIEW ALL** link to display all job templates, and 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. 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 resources 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 used 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

Chapter 6 | Installing and Accessing Ansible Tower

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 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. Use this playbook 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 6.3: 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

Click the book icon to display the Ansible Tower documentation website in a new window.

Log Out

Use the power icon to log out of the Ansible Tower web UI.

Ansible 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.

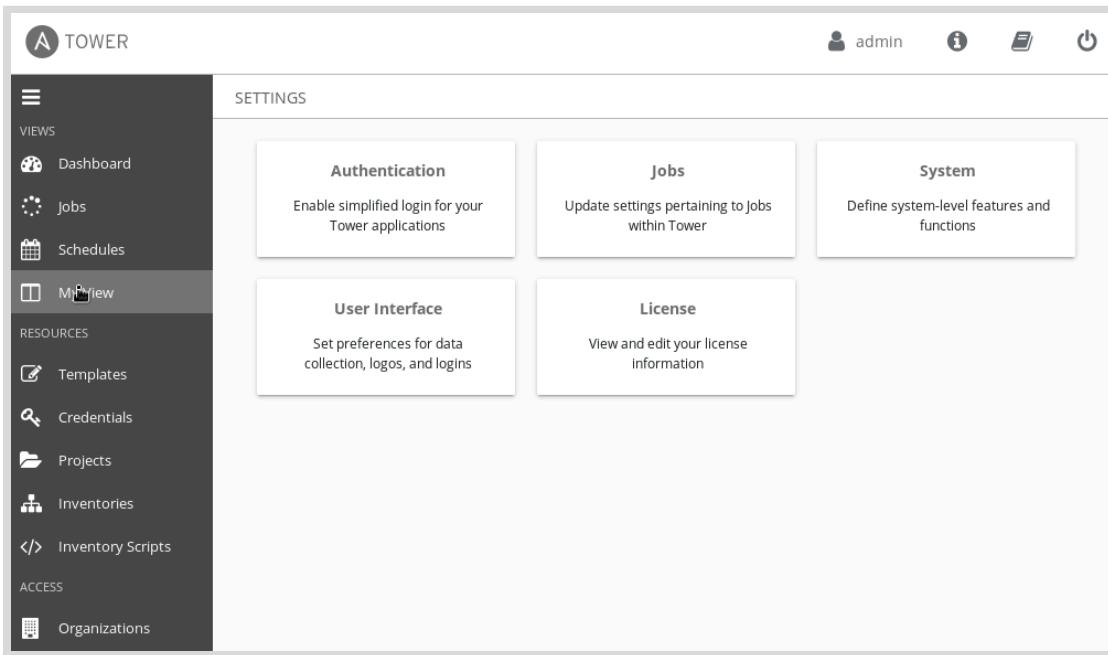


Figure 6.4: The Settings page in Ansible Tower

The different categories available on the **Settings** page are described below.

Authentication

The Authentication category contains settings used to configure simplified authentication for 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. Use these settings 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.

The screenshot shows two main sections of the Ansible Tower web UI. The top section is a modal titled 'NEW TEAM' under 'TEAMS / CREATE TEAM'. It contains three tabs: 'DETAILS' (selected), 'USERS', and 'PERMISSIONS'. The 'NAME' field is set to 'Developers', the 'DESCRIPTION' field contains 'Dev Team', and the 'ORGANIZATION' field is set to 'Default'. Below the form are 'CANCEL' and 'SAVE' buttons. The bottom section shows a list of teams with one item: 'Devops' under 'Default'. The list includes columns for 'NAME', 'ORGANIZATION', and 'ACTIONS' (with edit and delete icons). Above the list is a search bar and a 'KEY' button.

Figure 6.5: Breadcrumb Navigation and activity stream 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 trail 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, clicking the **View Activity Stream** icon on the **Projects** page displays a list of project-related events, including the event time and the user who initiated it.

Search Fields

Search fields that can be used to search or filter through data sets are present throughout the Ansible Tower web UI.

Click the **Key** button next to each search field to display a guide that describes the correct syntax and 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 template. The exercise that follows this section provides a more detailed hands-on walk-through of the example job template.

- 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 the inventory variable **ansible_connection: local** is set.

The screenshot shows two main sections of the Ansible Tower web interface. The top section is a modal window titled 'Demo Inventory' under the 'HOSTS' tab. It displays a single host entry: 'localhost' with status 'ON'. Below this is a 'RELATED GROUPS' section. The bottom section is a list of inventories, with 'INVENTORIES' selected in the tabs. It shows a single inventory named 'Demo Inventory' with type 'Inventory' and organization 'Default'. Both sections include search bars, key generation buttons, and action buttons for editing and deleting.

Figure 6.6: Details of Demo Inventory

- 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.

Demo Credential

DETAILS **PERMISSIONS**

* NAME ? **Demo Credential** DESCRIPTION ? ORGANIZATION

* CREDENTIAL TYPE ? **Machine**

TYPE DETAILS

USERNAME **admin** PASSWORD Prompt on launch

SSH PRIVATE KEY HINT: Drag and drop private file on the field below.

Figure 6.7: Details of Demo Credential

- Under **Projects** in the left navigation bar, a project called **Demo Project** has been configured. This project is configured to get a test Ansible Playbook from a GitHub repository that the Ansible team manages.

Demo Project

DETAILS **PERMISSIONS** **NOTIFICATIONS** **JOB TEMPLATES** **SCHEDULES**

* NAME **Demo Project** DESCRIPTION * ORGANIZATION **Default**

* SCM TYPE **Git**

* SCM URL ? **https://github/ansible/ansible-tower.git** SCM BRANCH/TAG/COMMIT SCM CREDENTIAL

SCM UPDATE OPTIONS CACHE TIMEOUT (SECONDS) ? **0**

CLEAN **0** DELETE ON UPDATE **0** UPDATE REVISION ON LAUNCH **0**

CANCEL **SAVE**

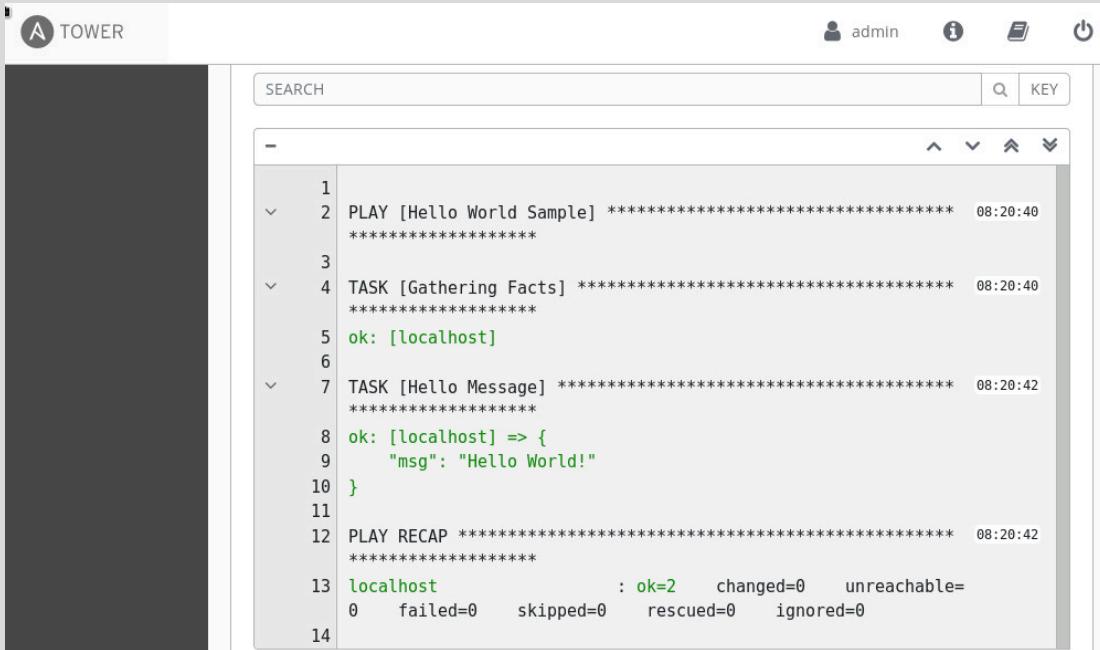
Figure 6.8: Details of Demo Project

- 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 information necessary to authenticate. 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. At the top, there's a navigation bar with the 'TOWER' logo, user 'admin', and various icons. Below the header is a search bar labeled 'SEARCH'. The main area displays a log of a completed job run:

```

1
2 PLAY [Hello World Sample] ****
3
4 TASK [Gathering Facts] ****
5 ok: [localhost]
6
7 TASK [Hello Message] ****
8 ok: [localhost] => {
9   "msg": "Hello World!"
10 }
11
12 PLAY RECAP ****
13 localhost : ok=2    changed=0    unreachable=
14   failed=0   skipped=0   rescued=0   ignored=0

```

Figure 6.9: Demo Job Template

Launching a Job

After creating and configuring 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 this icon launches the job using the settings in the job template.

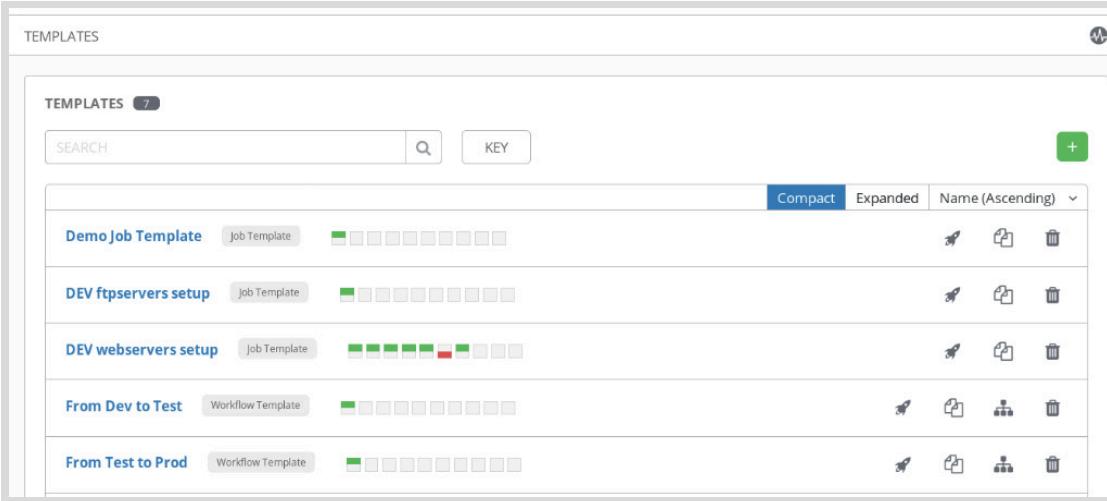


Figure 6.10: Launching a job

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 displays 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 output can be used for troubleshooting purposes. In the following example, the play and tasks in the playbook ran successfully.

JOBS / 44 - Demo Job Template

DETAILS	Demo Job Template
STATUS	Successful
STARTED	5/30/2019 12:37:10 PM
FINISHED	5/30/2019 12:37:25 PM
JOB TEMPLATE	Demo Job Template
JOB TYPE	Run
LAUNCHED BY	admin
INVENTORY	Demo Inventory
PROJECT	Demo Project
PLAYBOOK	hello_world.yml
CREDENTIAL	Demo Credential
ENVIRONMENT	/var/lib/awx/venv/ansible
EXECUTION NODE	localhost
INSTANCE GROUP	tower
EXTRA VARIABLES	YAML JSON
<pre>1 2 PLAY [Hello World Sample] **** 3 4 TASK [Gathering Facts] **** 5 ok: [localhost] 6 7 TASK [Hello Message] **** 8 ok: [localhost] => { 9 "msg": "Hello World!" 10 } 11 12 PLAY RECAP **** 13 localhost : ok=2 changed=0 14 unreachable=0 failed=0 skipped=0 rescued= 0 ignored=0</pre>	

Figure 6.11: 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 a job listed on this page to display the status page logged for that job.

The screenshot shows the Ansible Tower interface with the 'JOBS' tab selected in the sidebar. The main area displays a table of recent job runs, each with a green circular icon, a job ID, a name, and a type. The table includes columns for 'Compact', 'Expanded', and 'Finish Time (Descending)'. Each row has edit and delete icons. The jobs listed are:

Job ID	Job Name	Type	Actions
44	Demo Job Template	Playbook Run	
37	From Test to Prod	Workflow Job	
42	PROD webservers setup	Playbook Run	
41	My Webservers PROD	SCM Update	
39	TEST webservers setup	Playbook Run	
38	My Webservers TEST	SCM Update	
36	My Webservers PROD	SCM Update	
33	DEV webservers setup	Playbook Run	
34	My Webservers DEV	SCM Update	
25	From Dev to Test	Workflow job	
31	TEST webservers setup	Playbook Run	

Figure 6.12: 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 Red Hat 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 Red Hat Ansible Tower web UI to successfully launch a job.

Before You Begin

You should have a Red Hat 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 start**. This setup script configures the **tower** virtual machine for the exercise.

```
[student@workstation ~]$ lab tower-webui start
```

- ▶ 1. Log in to the Red Hat 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. The **lab** setup script has changed those parameters because by default the **Demo Project** uses Git to get a test playbook from GitHub.
- ▶ 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**.

**Note**

Even though there is no admin Linux user, the job can still work because it is a localhost job.

- ▶ 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 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 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 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 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 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 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 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.
- Red Hat 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 on left side of the Red Hat Ansible Tower web UI provides access to commonly used resources.

Chapter 7

Managing Access with Users and Teams

Goal

Create user accounts and organize them into teams in Red Hat Ansible Tower. Then, assign the users and teams permissions to administer and access resources in the Ansible Tower service.

Objectives

- Create new users in the web UI, and explain the different types of user in Ansible Tower.
- Create new teams in the web UI, assign users to them, and explain the different roles that can be assigned to users.

Sections

- Creating and Managing Ansible Tower Users (and Guided Exercise)
- Managing Users Efficiently with Teams (and Guided Exercise)

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.

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, or need access to change anything in the Ansible Tower installation.

The Ansible Tower interface has a built-in administrative user, **admin**, that has superuser access to the entire Ansible 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 Ansible Tower. Roles are 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.

Ansible Tower 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 Ansible 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. Ansible 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. Certain departments may 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 those Ansible Tower resources that they are expected to use.

As part of the Ansible Tower installation, a Default Organization is created. The following is the procedure for creating additional Organizations using the Ansible Tower web interface.

- Log in to the Ansible Tower web interface as the **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 the 'NEW ORGANIZATION' dialog box. At the top, there are three tabs: 'DETAILS' (selected), 'USERS', and 'PERMISSIONS'. Below the tabs are four input fields: 'NAME' (with a red asterisk), 'DESCRIPTION', 'INSTANCE GROUPS' (with a question mark icon), and 'MAX HOSTS' (set to 0). At the bottom right are 'CANCEL' and 'SAVE' buttons.

Figure 7.1: Complete information about the new Organization

- Click **SAVE** to finalize the creation of the new Organization.

User Types

By default, the Ansible Tower installer creates an **admin** user account with full control of the Ansible Tower installation. Using the special **admin** account, an Ansible Tower administrator can log in to the web interface and create additional users.

The three *user types* in Ansible Tower are:

System Administrator

The **System Administrator** user type (also known as *Superuser*) provides unrestricted access to perform any action within the entire Ansible Tower installation. **System Administrator** is a special *singleton role*, which has read-write permission on all objects in all Organizations on the Ansible Tower.

The **admin** user created by the installer also has the System Administrator singleton role and should therefore only be used by the Ansible Tower administrator.

System Auditor

The **System Auditor** user type also has a special singleton role, which has read-only access to the entire Ansible 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 Ansible 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 Ansible 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' interface. At the top, there's a navigation bar with 'USERS / CREATE USER'. Below it is a title 'NEW USER'. There are four tabs: 'DETAILS' (which is active and highlighted in dark grey), 'ORGANIZATIONS', 'TEAMS', and 'PERMISSIONS'. The 'DETAILS' tab contains the following fields:
 - First Name: 'New'
 - Last Name: 'User'
 - Organization: A dropdown menu with a search icon and the value 'Default'.
 - Email: 'email@example.com'
 - Username: 'newuser'
 - Confirm Password: 'redacted'
 - Password: 'redacted'
 - Show Password: A checkbox labeled 'SHOW' which is checked.
 - User Type: A dropdown menu set to 'Normal User'.
 At the bottom right of the dialog are two buttons: 'CANCEL' and 'SAVE' (in a green box).

Figure 7.2: Fill in information 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 Ansible Tower objects. An Organization is itself one of these objects. There are four roles that users can be assigned on an Organization:

Organizational 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 Ansible 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 Ansible Tower.

Users with the **System Auditor** singleton role inherit the **Auditor** role on every Organization within Ansible Tower.

A user created with the Normal User type is assigned a **Member** role on the Organization they were assigned to at the time of the user's creation in Ansible 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 Ansible Tower web interface as **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 **Permissions** 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:

- In the **ADD USERS/TEAMS** screen, under **USERS**, 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.

The screenshot shows the 'ORGANIZATIONS / Default / PERMISSIONS' page. At the top, there are tabs for 'DETAILS', 'USERS', 'PERMISSIONS' (which is selected), and 'NOTIFICATIONS'. Below the tabs is a search bar and a 'KEY' button. A green '+' button is located in the top right corner of the main content area. The main content is a table with columns 'USER', 'ROLE', and 'TEAM ROLES'. The data in the table is as follows:

USER	ROLE	TEAM ROLES
admin	SYSTEM ADMINISTRATOR	
daniel	PROJECT ADMIN	
david	MEMBER	
donnie	MEMBER	

Figure 7.3: 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 at `https://tower.lab.example.com` 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 in the top right corner to log out.

► **4.** Create a user, **sylvia**, as a System Auditor.

- 4.1. Log back in as **admin** user with password of **redhat**.
- 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	sylvia@lab.example.com
USERNAME	sylvia
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
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** 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 Ansible 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, an Ansible Tower administrator can assign roles to the Team representing a group of users.

In Ansible Tower, users exist as objects at a Tower-wide level. A user can have roles in multiple Organizations. A Team belongs to exactly one Organization, but an Ansible 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 Ansible Tower web interface as the **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 Ansible 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

Users can be assigned roles for a particular Team. These roles control whether the user is considered part of that Team, can administer it, or can view its membership.

A user can be assigned one or more of the following Team roles:

member

The Team **member** role allows users to inherit roles on Ansible Tower resources granted to the Team. It also grants users the ability to see the Team's users and associated 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.

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 must have the **admin** role for both the Team and the Project.

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 Ansible Tower resources.



Note

In practice, most organizations do not use Team roles other than **member**, and the current Red Hat Ansible Tower web UI makes it more difficult to set these other roles. Instead, Team membership is managed through an external authentication source, or the **Organization Administrator** and **System Administrator** roles are used for administrative purposes and **System Auditor** for auditing requirements instead of **read** on individual Teams.

Adding Users to a Team

After Team creation is complete, you can add users to the Team. To add users with the **Member** role to a Team in an Organization, execute the following procedure:

- Log in to the Ansible Tower web interface as the **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.

USER	FIRST NAME	LAST NAME	ROLE
admin			SYSTEM ADMINISTRATOR
daniel	Daniel	George	ADMIN
david	David	Jackobs	MEMBER
donnie	Donnie	Jameson	READ

Figure 7.4: Add a user to a team with the Member role

- On the **ADD USERS** screen, select one or more users to add to Team. Then, click **SAVE** to save your edits, adding the user to the Team.

Setting Team Roles

Since Red Hat Ansible Tower 3.4, adding users with either the **admin** or the **read** roles to a Team in an Organization requires the **tower-cli** tool, a command-line tool for the Red Hat Ansible Tower REST API, or directly with the Red Hat Ansible Tower API. The **tower-cli role grant** command grants a user, specified with the **--user** flag, a role, specified with the **--type** flag, on a team, specified with the **--target-team** flag. The following example grants the **admin** role to **joe** on the **Operators** team.

```
[student@workstation ~]$ tower-cli role grant --user 'joe' \
> --target-team 'Operators' --type 'admin'
```

The following example grants the **read** role to **jennifer** on the **Architects** team.

```
[student@workstation ~]$ tower-cli role grant --user 'jennifer' \
> --target-team 'Architects' --type 'read'
```



Note

To use **tower-cli**, you need to run the **tower-cli config** command to specify the host for Red Hat Ansible Tower, and the username and password to access it. You can allow unverified SSL connections with the **tower-cli config verify_ssl false** command.



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>

You can find further information in the Introduction to tower-cli section of the *Ansible Tower API Guide*

<https://docs.ansible.com/ansible-tower/latest/html/towerapi/>

► 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 will be able to organize users into Teams and explain the different roles that users can have on a Team.

Before you begin

You 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.

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab org-team start
```

This script configures the environment to use the **tower-cli** tool.

► 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

15. 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
ORGANIZATION	Default
EMAIL	daniel@lab.example.com
USERNAME	daniel
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. Open a terminal, and use the **tower-cli** tool to assign the **admin** role on the **Developers** Team to the **daniel** user.

```
[student@workstation ~]$ tower-cli role grant --user 'daniel' \
> --target-team 'Developers' --type 'admin'
== ===== ==
id user type target_team
== ===== ==
38      5 admin N/A
== ===== ==
```

In the previous output, **N/A** is expected in the **target_team** column.

- 4. Create the **donnie** user and grant her the **Read** role for the **Developers** Team.

- 4.1. Back in the Tower web interface, 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
ORGANIZATION	Default
EMAIL	donnae@lab.example.com
USERNAME	donnae
PASSWORD	redhat123
CONFIRM PASSWORD	redhat123
USER TYPE	Normal User

- 4.4. Click **SAVE** to create the new user.
- ▶ 5. Open a terminal, and use the **tower-cli** tool to assign the **donnae** user the **read** role for the **Developers** Team.

```
[student@workstation ~]$ tower-cli role grant --user 'donnae' \
> --target-team 'Developers' --type 'read'
```

- ▶ 6. Create a user, **david**, to be granted **Member** role to the **Developers** Team.
- 6.1. Back in the Tower web interface, 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	Jackobs
ORGANIZATION	Default
EMAIL	david@lab.example.com
USERNAME	david
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. Check the box next to **david** to select this user.
 - 7.6. 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. Click **Cancel**.
 - 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 have an Ansible Tower instance installed and configured from an earlier exercise running on the **tower** system.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

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

This script configures the environment to use the **tower-cli** tool.

- 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

- Create a Team called **Operations** in the **Default** Organization.
- Add **Oliver** as a **Member** to the **Operations** Team.
- Add **Ophelia** as an **Admin** to the **Operations** Team.
- Click the **Log Out** icon to log out of the Tower web interface.
- 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 have an Ansible Tower instance installed and configured from an earlier exercise running on the **tower** system.

Before You Begin

Open a terminal on **workstation** as the **student** user and run the following command:

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

This script configures the environment to use the **tower-cli** tool.

- 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

- Log in to the Tower web interface as the **admin** user with the **redhat** password.
- Click **Users** in the left navigation bar to manage users.
- Click the **+** button to add a new user.
- 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
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.
- 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. Check the box next to **oliver** to select that user.
 - 3.4. Click **SAVE**.
 - 3.5. You can verify on the next screen that **oliver** has been assigned the role of **Member**.
4. Add **Ophelia** as an **Admin** to the **Operations** Team.
- 4.1. Open a terminal, and use the **tower-cli** tool to assign the **ophelia** user the **admin** role for the **Operations** Team.

```
[student@workstation ~]$ tower-cli role grant --user 'ophelia' \
> --target-team 'Operations' --type 'admin'
== ===== ==
id user type target_team
== ===== ==
41   9 admin N/A
== ===== ==
```

In the previous output, **N/A** is expected in the **target_team** column.

- 4.2. Back in to the Tower web interface, refresh the web page to update the list of users for the **Operations** Team, and verify that **ophelia** has the **admin** role associated.
5. Click the **Log Out** icon to log out of the Tower web interface.
6. Run the command **lab org-review grade** on **workstation** to grade your exercise.

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

This concludes the lab.

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.
- The roles assigned to users control what access users have on objects.

Chapter 8

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.

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 defines this inventory as either 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. The file 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 part of any group other than **all**. In the preceding example, the only ungrouped host is **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 the inventory as a static inventory, and use RBAC to control access to it, 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. Access to an inventory object in Tower is determined by the roles a user has 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.



Important

Your Red Hat Ansible Tower license determines the maximum number of hosts you may define in your inventory. Clicking on **Settings** in the left pane and selecting **License** from the Settings page will display how many hosts your license supports (under **HOSTS AVAILABLE**) and how many are remaining (under **HOSTS REMAINING**).

If more managed nodes are in your Ansible Tower inventory than are supported by the license, you will be unable to start any Jobs in Ansible Tower. If a dynamic inventory sync causes Ansible Tower to exceed the managed node count specified in the license, the dynamic inventory sync will fail.

If you have multiple hosts in inventory that have the same name, such as **webserver1**, they will be counted for licensing purposes as a single node. Note that this differs from the **Hosts** count on the Dashboard, which counts hosts in separate inventories separately. Note that this behavior is case-sensitive; **webserver1** and **WebServer1** will be treated as different nodes.

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.

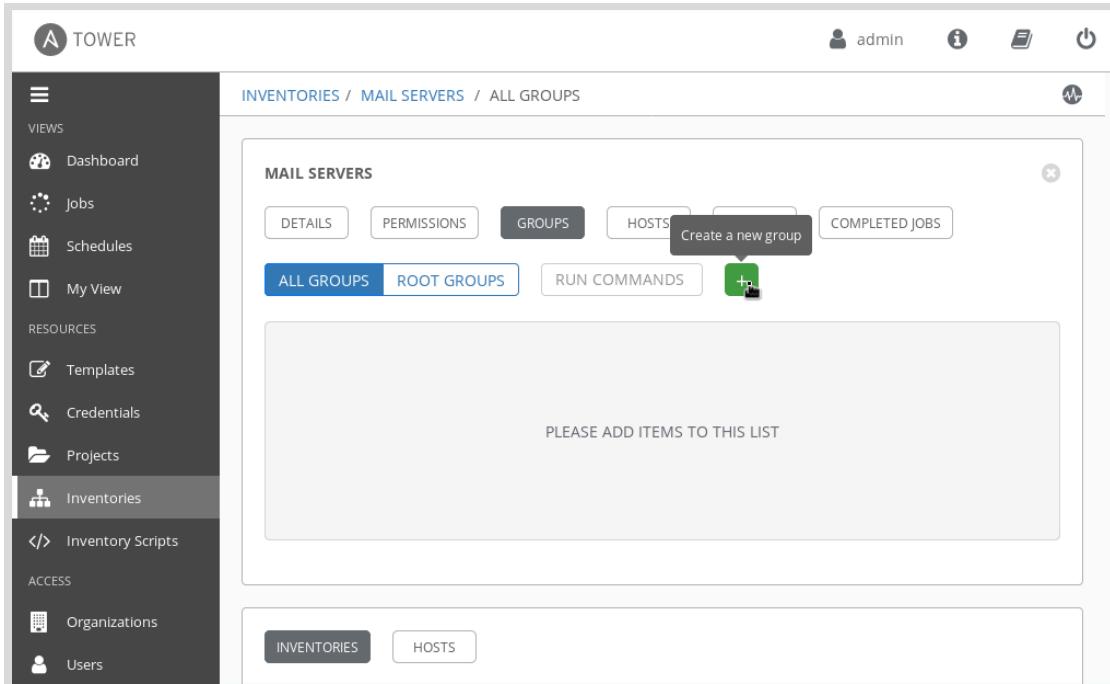


Figure 8.1: New host group

- Add the **northeast** and **southeast** groups as child groups of the **east** group, previously 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.

This screenshot shows the Ansible Tower interface with the same sidebar as Figure 8.1. The main content area shows the 'east' group detail screen under 'INVENTORIES / MAIL SERVERS / ALL GROUPS / east / ASSOCIATED GROUPS'. The 'east' group is selected, and its detail screen is shown with tabs for DETAILS, GROUPS (selected and highlighted in dark grey), and HOSTS. A 'RUN COMMANDS' button and a green '+' icon are also present. A large text box below says 'PLEASE ADD ITEMS TO THIS LIST'. Below this is another 'MAIL SERVERS' section with tabs for DETAILS, PERMISSIONS, GROUPS (selected), HOSTS, SOURCES, and COMPLETED JOBS. The top right corner shows the user 'admin' and system icons.

Figure 8.2: 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 8.3: New child host groups

- Add the **raleigh1.example.com** and **atlanta1.example.com** hosts to the **southeast** child group, previously 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 shows the TOWER interface with the left sidebar expanded. The 'Inventories' option is selected. The main area displays two child group details screens: 'southeast' and 'MAIL SERVERS'. The 'southeast' screen has tabs for DETAILS, GROUPS, and HOSTS, with the HOSTS tab selected. It shows a search bar, a key field, and two hosts: 'atlanta1.example.com' and 'raleigh1.example.com', both with the 'ON' status. The 'MAIL SERVERS' screen below it has tabs for DETAILS, PERMISSIONS, GROUPS, HOSTS, SOURCES, and COMPLETED JOBS, with the GROUPS tab selected. It shows tabs for ALL GROUPS and ROOT GROUPS, and a 'RUN COMMANDS' button.

Figure 8.4: Partially configured child groups

- 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.

The screenshot shows the TOWER interface with the left sidebar expanded. The 'Inventories' option is selected. The main area displays the 'east' child group detail screen. The 'HOSTS' tab is selected, showing three hosts: 'atlanta1.example.com', 'boston1.example.com', and 'raleigh1.example.com', all with the 'ON' status. Below this is the 'MAIL SERVERS' section with its respective tabs.

Figure 8.5: Completed host group detail screen

9. The completed **Mail Servers** inventory screen looks similar to the following:

HOSTS	RELATED GROUPS	ACTIONS
<input type="radio"/> ON atlanta1.example.com	southeast X	Edit Delete
<input type="radio"/> ON boston1.example.com	northeast X	Edit Delete
<input type="radio"/> ON london1.example.com		Edit Delete
<input type="radio"/> ON raleigh1.example.com	southeast X	Edit Delete

Figure 8.6: 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, then it is deleted from all host groups in the inventory.

If you click on a trash can icon next to a host group, then 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, a role granting permissions to work with an inventory indirectly may be inherited.

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, click it 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, then they are honored. However, you cannot 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 these have different values, then it is difficult to predict which value Tower will use.

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:

INVENTORIES / MAIL SERVERS

MAIL SERVERS

DETAILS PERMISSIONS GROUPS HOSTS SOURCES COMPLETED JOBS

* NAME: MAIL SERVERS DESCRIPTION: * ORGANIZATION: Default

INSIGHTS CREDENTIAL: INSTANCE GROUPS: ?

VARIABLES: ? YAML JSON EXPAND

```
1: ---
```

CANCEL SAVE

Figure 8.7: 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. The group variables 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:

INVENTORIES / MAIL SERVERS / ALL GROUPS / southeast

southeast

DETAILS GROUPS HOSTS

* NAME: southeast DESCRIPTION:

VARIABLES: ? YAML JSON

```
1: ---
2: ntp: ntp-se.example.com
```

CANCEL SAVE

Figure 8.8: 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. Alternatively, click on the **Edit host** (pencil) icon next to the host's name in the inventory, once created, to define the host variables. Variables defined in this manner only apply to the specific host.

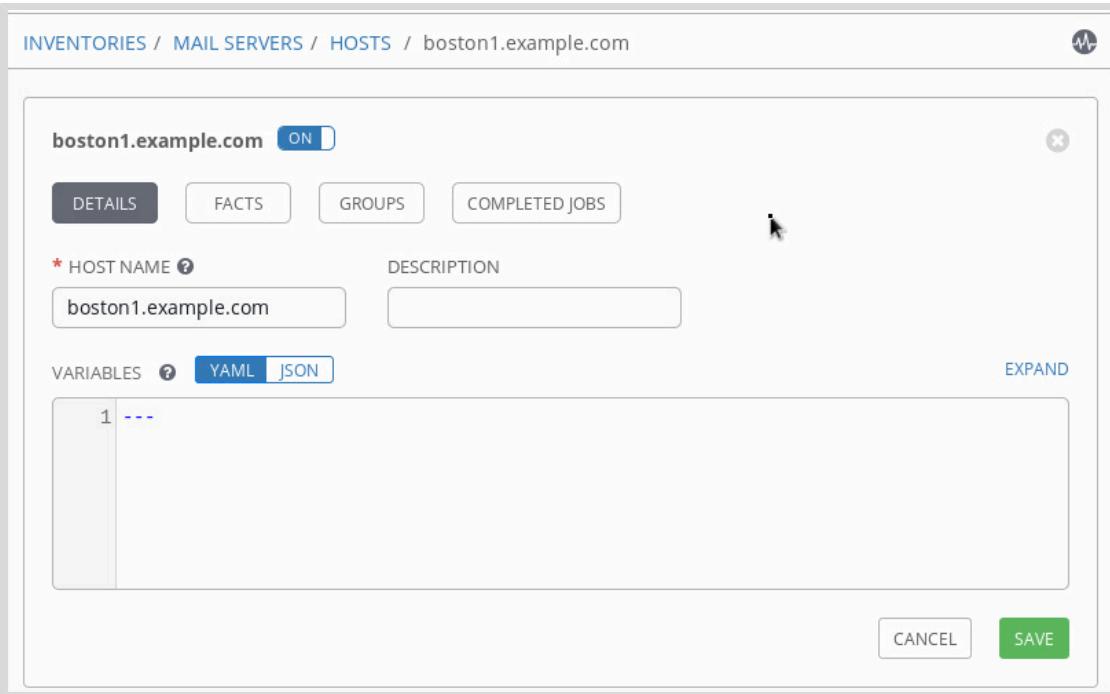


Figure 8.9: Variables for an individual host



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>

A discussion of how node counting works for licenses is available at

<https://docs.ansible.com/ansible-tower/latest/html/administration/licensing-support.html#node-counting-in-licenses>

► 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.

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 host-inventory start**.

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

This setup script creates 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 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.

Credentials

Credentials are Ansible Tower objects used to authenticate to remote systems. They may provide secrets, such as passwords and 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. Although users and teams can be assigned privileges to use Credentials, 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 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 that owns 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. Other users and teams cannot be granted roles.



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.

1. 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.
2. Click **Credentials** to enter 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. A **NAME** is required, and the **TYPE** drop-down menu should be used to select **Machine**. If the user has Organization **Admin** privileges, then 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.
5. For Machine Credentials, additional fields appear in the **TYPE DETAILS** section as shown in the following illustration:

The screenshot shows the 'Edit Credential' page for a 'Mail Server Credential'. The top navigation bar includes 'CREDENTIALS / EDIT CREDENTIAL' and a gear icon. The main form has tabs for 'DETAILS' (selected) and 'PERMISSIONS'. Under 'DETAILS', fields include:

- * NAME**: Mail Server Credential
- * CREDENTIAL TYPE**: Machine
- DESCRIPTION**: (empty)
- ORGANIZATION**: (button to 'SELECT AN ORGANIZATION')

 A section titled 'TYPE DETAILS' contains:

- USERNAME**: playbook
- PASSWORD**: (input field with eye icon)
- Prompt on launch

 An 'SSH PRIVATE KEY' section shows a file named 'ENCRYPTED' with a small preview icon.

Figure 8.10: 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 this 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, then login as the user who created the Credential. If editing an Organization Credential, then 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 cannot 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 in which the Credential was created.
2. Click **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. On 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.

NEW CREDENTIAL

DETAILS **PERMISSIONS**

* NAME ? DESCRIPTION ? ORGANIZATION ?

* CREDENTIAL TYPE ?

TYPE DETAILS

USERNAME PASSWORD Prompt on launch

SSH PRIVATE KEY HINT: Drag and drop private file on the field below.

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpaIBAAKCAQEAxrVFT4rJh8rgluAg3U1TjjqumsknbWHp1xVlsexyWuFU3qZB
tNGmiyHfaHK2tdIXPjAdBkyTAsmmw794vDSnYuhgx/2ji1bKfxMMvAxM2TAZSQwr
eL9oup8FOJYsOioBBZGbPuYZ61PHeEM5XchieaJ6/gBrxTihuvAktAQZeNGdbcH9
R4QssfJ0p8w9ql5ZMVkRWsjwBa7TwgVb0A5Dn1RWRfpM2ks7YAI2xWeAPY6kM1G
```

Figure 8.11: 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 that 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. Under the 'DETAILS' tab, a new credential is being created with the name 'Finance DBA', credential type 'Machine', and organization 'Default'. The 'TYPE DETAILS' section includes a 'USERNAME' field set to 'findba', a 'PASSWORD' field with the 'SHOW' button checked and the 'Prompt on launch' checkbox selected, and an 'SSH PRIVATE KEY' field which is currently empty. Below these, there are fields for 'PRIVATE KEY PASSPHRASE' and 'PRIVILEGE ESCALATION METHOD', both with their respective checkboxes. A 'PRIVILEGE ESCALATION USERNAME' field is also present. At the bottom right are 'CANCEL' and 'SAVE' buttons.

Figure 8.12: 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.

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 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** and **david** 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.

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 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.

Evaluation

As the **student** user on **workstation**, run the **lab host-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 host-review grade
```

This concludes the lab.

► 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.

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 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
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 | servera.lab.example.com |
| 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 | serverb.lab.example.com |
| 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** and **david** are assigned the **Admin** role on the **Developers** Credential.
- 7.9. Click the **Log Out** icon to exit the Tower web interface.

Evaluation

As the **student** user on **workstation**, run the **lab host-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 host-review grade
```

This concludes the lab.

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 9

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 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

- 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

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.

Projects

An Ansible project represents at least one playbook and its associated collection of related playbooks and roles. Whether 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, 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`.

However, this is not a recommended practice. 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 project 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.

The screenshot shows a 'CREATE PROJECT' dialog box. At the top, it says 'PROJECTS / CREATE PROJECT'. Below that, the title 'NEW PROJECT' is displayed. There are four tabs: 'DETAILS' (which is selected), 'PERMISSIONS', 'JOB TEMPLATES', and 'SCHEDULES'. Under the 'DETAILS' tab, there are three main sections: 'NAME' (with a placeholder 'Project Name'), 'DESCRIPTION' (with a placeholder 'A brief description of the project'), and 'ORGANIZATION' (with a dropdown menu showing 'Default'). Below these is a section for 'SCM TYPE' with a dropdown menu containing 'Choose an SCM Type'. At the bottom right of the dialog are 'CANCEL' and 'SAVE' buttons.

Figure 9.1: 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.

USER	ROLE	TEAM ROLES
admin	SYSTEM ADMINISTRATOR	
simon	SYSTEM ADMINISTRATOR	
sylvia	SYSTEM AUDITOR	

ITEMS 1 - 3

Figure 9.2: 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, then there are no specific role requirements.
 - If creating an SCM Credential belonging to an Organization, then 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.

- 4.1. Enter a unique name for the Credential in the **NAME** field.
- 4.2. 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.
- 4.3. 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 display 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' interface. At the top, it says 'CREDENTIALS / CREATE CREDENTIAL'. Below that, a modal window titled 'NEW CREDENTIAL' is open. It has two tabs: 'DETAILS' (which is selected) and 'PERMISSIONS'. In the 'DETAILS' tab, there are three main input fields: 'NAME' (containing 'New'), 'DESCRIPTION' (containing 'Credentials'), and 'ORGANIZATION' (with a search icon and a button labeled 'SELECT AN ORGANIZATION'). Below these, a dropdown menu shows 'Source Control' as the selected 'CREDENTIAL TYPE'. Under the 'TYPE DETAILS' section, there are two input fields: 'USERNAME' and 'PASSWORD', each with a search icon. At the bottom of the modal, there is a note: 'SCM PRIVATE KEY HINT: Drag and drop private file on the field below.' and a corresponding empty file input field.

Figure 9.3: 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 have 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	

ITEMS 1 - 3

Figure 9.4: 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 an 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 project 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.

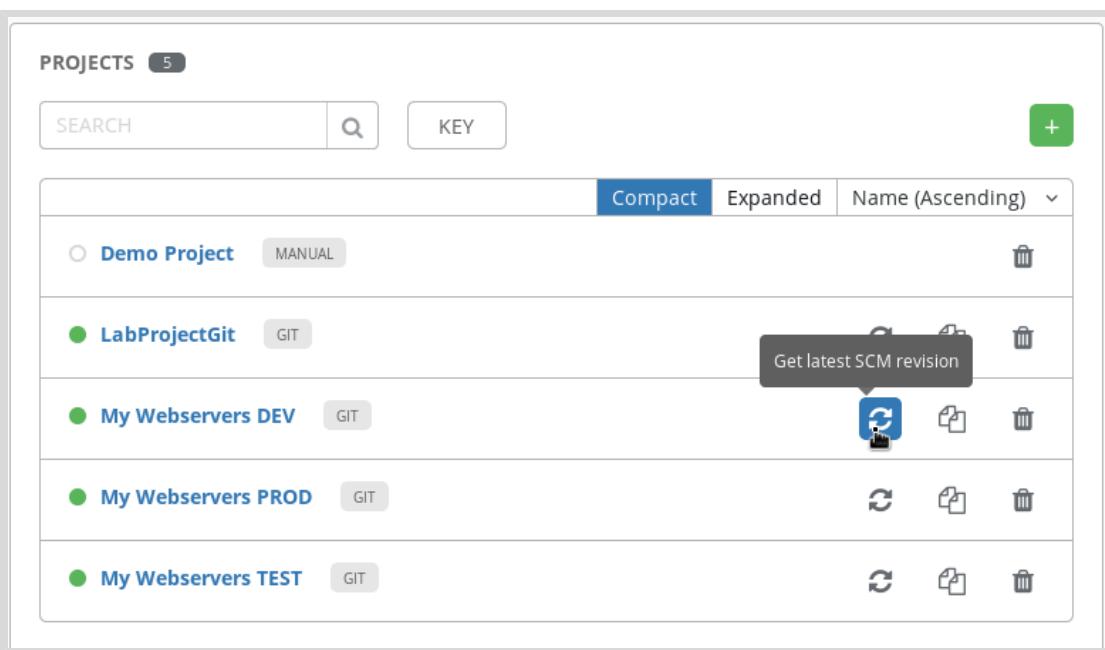


Figure 9.5: Updating an SCM project



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 will be able to create a Project which will allow Ansible Tower to leverage an external Git repository containing a playbook.

Before You Begin

You 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 start**. This setup script ensures that the **workstation** and **tower** virtual machines are started.

```
[student@workstation ~]$ lab provision-project start
```

- ▶ 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 /home/student/.ssh/lab_rsa private key file on workstation into this field

- 2.4. 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/var/opt/gitlab/git-data/repositories/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. Click **My Webservers DEV** to edit the Project settings.
 - 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.

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 will 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.
 - 6.1. Click the magnifying glass icon for the **INVENTORY** field.
 - 6.2. Select the desired Inventory and then click **SELECT**.
7. Specify the Project containing the playbook that the job will execute.
 - 7.1. Click the magnifying glass icon for the **PROJECT** field.
 - 7.2. Select the desired Project and then click **SELECT**.
8. Specify the playbook that the job will execute.
 - 8.1. Click the drop-down menu for the **PLAYBOOK** field. All playbooks contained in the Project selected in the previous field are listed.
 - 8.2. Select the desired playbook.
9. Specify the Credential required for authenticate against the managed hosts.
 - 9.1. Click the magnifying glass icon for the **CREDENTIAL** field.
 - 9.2. 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.

NEW JOB TEMPLATE

DETAILS **PERMISSIONS** **COMPLETED JOBS** **SCHEDULES** **ADD SURVEY**

* NAME <input type="text"/>	DESCRIPTION <input type="text"/>	* JOB TYPE <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH Run
* INVENTORY <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH <input type="text"/>	* PROJECT <small>?</small> <input type="text"/>	* PLAYBOOK <small>?</small> <input type="text"/> Choose a playbook
CREDENTIAL <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH <input type="text"/>	FORKS <small>?</small> <input type="text"/> 0	LIMIT <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH <input type="text"/>
* VERBOSITY <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH 0 (Normal)	JOB TAGS <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH <input type="text"/>	SKIP TAGS <small>?</small> <input type="checkbox"/> PROMPT ON LAUNCH <input type="text"/>
LABELS <small>?</small> <input type="text"/>	INSTANCE GROUPS <small>?</small> <input type="text"/>	JOB SLICING <small>?</small> <input type="text"/> 2

Figure 9.6: 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, a single job template that has the **Prompt on launch** option enabled for the Inventory field will suffice. 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 can only select from Inventories that they have been assigned the **Use** role on.

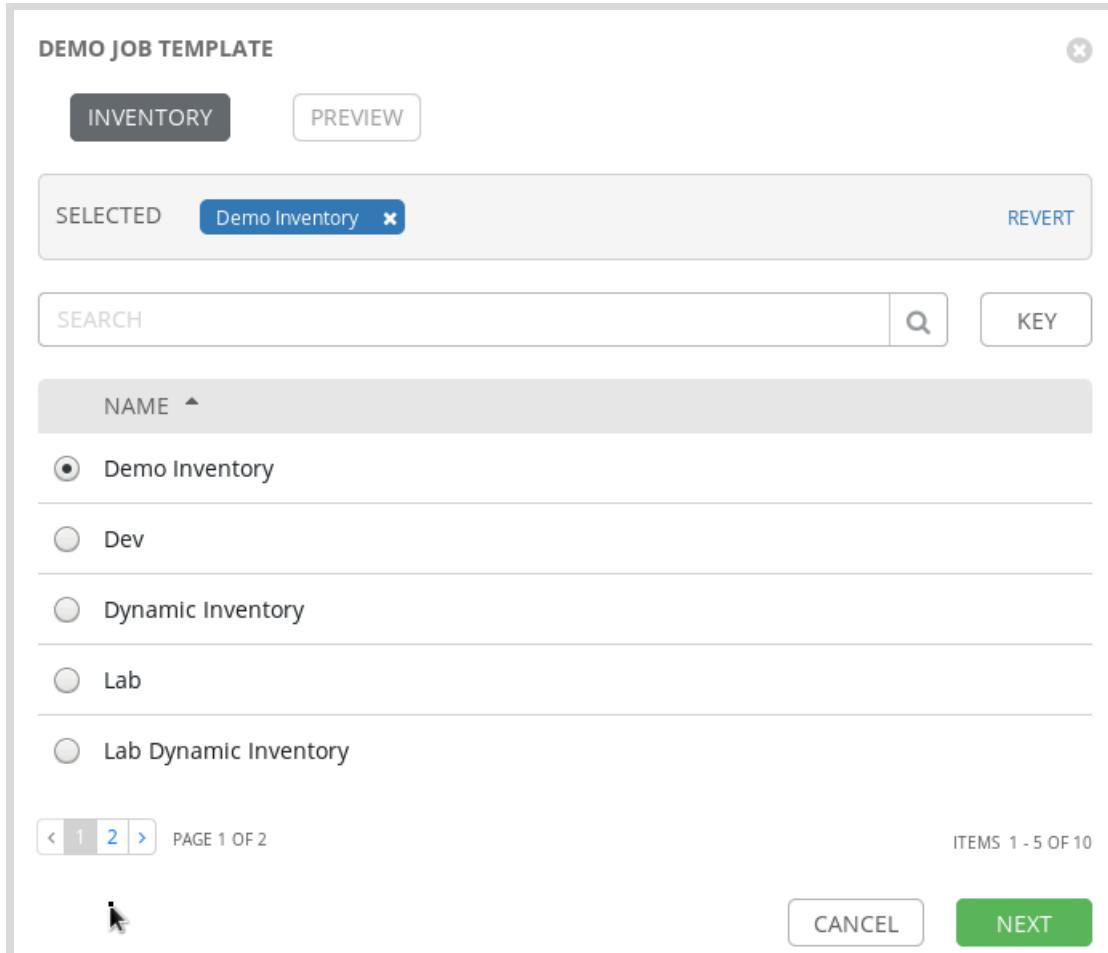


Figure 9.7: Prompting for Inventory on launch

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 it is created.

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' (which is selected), 'NOTIFICATIONS', 'COMPLETED JOBS', and 'SCHEDULES'. Below the tabs is a search bar and a 'KEY' button. A green '+' button is located in the top right corner. The main area contains a table with columns 'USER', 'ROLE', and 'TEAM ROLES'. The data in the table is:

USER	ROLE	TEAM ROLES
admin	SYSTEM ADMINISTRATOR	
simon	SYSTEM ADMINISTRATOR	
sylvia	SYSTEM AUDITOR	

At the bottom right of the table, it says 'ITEMS 1 - 3'.

Figure 9.8: 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. At the top, there is a search bar and a 'KEY' button. A green '+' button is in the top right. The main area lists one template: 'Demo Job Template' (Job Template). To the right of the template name is a context menu with options: 'Compact', 'Expand', 'Start a job using this template', and 'Delete'. A tooltip for the 'Start a job using this template' option says 'Start a job using this template'. At the bottom right, it says 'ITEMS 1 - 1'.

Figure 9.9: Launching a Job

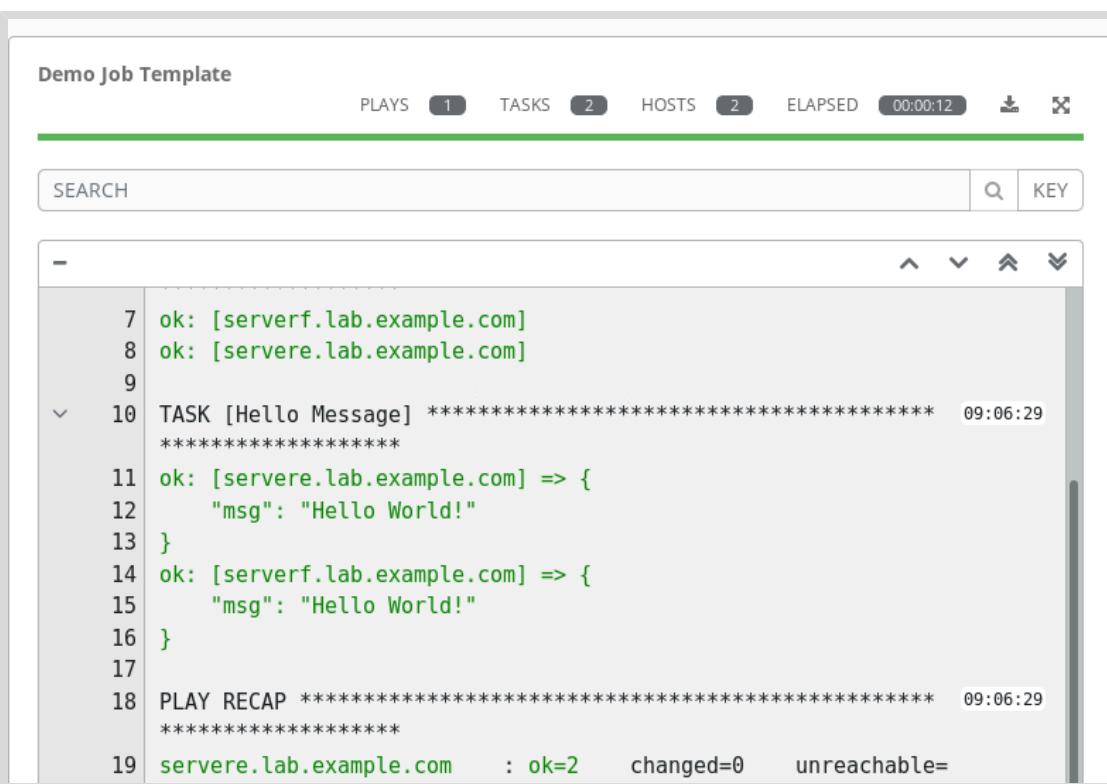
Evaluating the Results of a Job

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.



The screenshot shows the 'Demo Job Template' output pane. At the top, there are summary statistics: PLAYS 1, TASKS 2, HOSTS 2, and ELAPSED 00:00:12. Below these are search and filter controls. The main area displays a log of task execution. Task 10, 'Hello Message', is expanded, showing its individual steps (ok: [serverf.lab.example.com], ok: [servere.lab.example.com]) and its execution time (09:06:29). Task 18, 'PLAY RECAP', is also shown. Task 19, 'servere.lab.example.com : ok=2 changed=0 unreachable=' is partially visible at the bottom. On the right side of the log area, there are scroll controls (up, down, first, last) and a vertical scrollbar.

```

Demo Job Template
PLAYS 1 TASKS 2 HOSTS 2 ELAPSED 00:00:12
SEARCH   KEY 

-
7 ok: [serverf.lab.example.com]
8 ok: [servere.lab.example.com]
9
10 TASK [Hello Message] *****
***** 09:06:29
11 ok: [servere.lab.example.com] => {
12   "msg": "Hello World!"
13 }
14 ok: [serverf.lab.example.com] => {
15   "msg": "Hello World!"
16 }
17
18 PLAY RECAP *****
***** 09:06:29
19 servere.lab.example.com : ok=2 changed=0 unreachable=


```

Figure 9.10: 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 will be able to create a Job Template and launch a Job from the Ansible Tower web interface.

Before You Begin

You 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 start**. This setup script ensures that the **workstation** and **tower** virtual machines are started.

```
[student@workstation ~]$ lab provision-job start
```

- ▶ 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	DEV webservers setup
DESCRIPTION	Setup apache on DEV webservers
JOB TYPE	Run
INVENTORY	Dev
PROJECT	My Webservers DEV
PLAYBOOK	apache-setup.yml
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 new tab and go to **<http://servera.lab.example.com>**. You should see the following output:

```
This is a test message RedHat 8.0
Current Host: servera
Server list:
serverb.lab.example.com
servera.lab.example.com
```

- 5.2. Open a new tab and go to **<http://serverb.lab.example.com>**. You should see the following output:

```
This is a test message RedHat 8.0
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 should exist. Pull the latest changes from the **my_webservers_DEV** Git repository.

6.1. Change directory to the **/home/student/git-repos/my_webservers_DEV**.

```
[student@workstation ~]$ cd ~/git-repos/my_webservers_DEV
```



If the directory does not exist, clone the project instead:

```
[student@workstation ~]$ cd ~/git-repos
[student@workstation git-repo]$ git clone \
> http://git@git.lab.example.com:8081/git/my_webservers_DEV.git
[student@workstation git-repo]$ cd 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/D0447/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/D0447/labs/provision-job/** directory to the current working directory.

```
[student@workstation my_webservers_DEV]$ cp -R -v ~/D0447/labs/provision-job/* .
'/home/student/D0447/labs/provision-job/ansible-vsftpd.yml' -> './ansible-vsftpd.yml'
'/home/student/D0447/labs/provision-job/ftpclient.yml' -> './ftpclient.yml'
'/home/student/D0447/labs/provision-job/site.yml' -> './site.yml'
'/home/student/D0447/labs/provision-job/templates/vsftpd.conf.j2' -> './templates/vsftpd.conf.j2'
'/home/student/D0447/labs/provision-job/vars' -> './vars'
'/home/student/D0447/labs/provision-job/vars/defaults-template.yml' -> './vars/defaults-template.yml'
'/home/student/D0447/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 3a0643f] Adding files
 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
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (10/10), 1.69 KiB | 866.00 KiB/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
  ac5d142..3a0643f master -> master
```

► **8.** Trigger an SCM update of the project **My Webservers DEV**.

- 8.1. Go back to the Ansible Tower web interface. You should still be logged in as user **daniel**.
- 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.
- 8.5. Click the **Log Out** icon to log out of the Ansible Tower web interface.

► **9.** Log in to the Ansible Tower web interface running on the **tower** system using the **admin** account and the **redhat** password.

► **10.** Create a new Job Template called **DEV ftpservers setup**.

- 10.1. In the left navigation bar, click the **Templates** quick navigation link.
- 10.2. Click the **+** button to add a new Job Template.
- 10.3. From the drop-down menu, select **Job Template**.
- 10.4. On the next screen, fill in the details as follows:

Field	Value
NAME	DEV ftpperservers 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.

- 10.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.
- ▶ 11. Give the **Developers** team **Admin** role on the Job Template **DEV ftpperservers setup**.
 - 11.1. Click the **PERMISSIONS** button to manage the Job Template's permissions.
 - 11.2. Click the **+** button on the right to add permissions.
 - 11.3. Click **TEAMS** to display the list of available teams.
 - 11.4. 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.
 - 11.5. In the second section below, select the **Admin** role from the drop-down menu.
 - 11.6. Click **SAVE** to make the role assignment. This redirects you to the list of permissions for the Job Template **DEV ftpperservers setup**, which now shows that all members of the **Developers** team are assigned the **Admin** role on the Job Template.
- ▶ 12. Launch a Job using the Job Template **DEV ftpperservers setup**, as a member of the **Developers** team.
 - 12.1. Click the **Log Out** icon in the top right corner to logout and log back in as **daniel** with a password of **redhat123**.
 - 12.2. In the left navigation bar, click the **Templates** quick navigation link.
 - 12.3. On the same line as the Job Template **DEV ftpperservers setup**, click the rocket icon on the right to launch the Job. This redirects you to a detailed status page of the running job.
 - 12.4. Observe the live output of the running job for a minute.
 - 12.5. Verify that the **STATUS** of the Job in the **DETAILS** section displays **Successful**.

- 13. Verify that the FTP servers are up and running on **servera.lab.example.com** and **serverb.lab.example.com**.

- 13.1. On **workstation** use the **nc servera 21** command to verify that FTP server is running on **servera**.

```
[student@workstation ~]$ nc servera 21  
220 (vsFTPd 3.0.3)  
CTRL+C
```

- 13.2. On **workstation** use the **nc serverb 21** command to verify that FTP server is running on **serverb**.

```
[student@workstation ~]$ nc serverb 21  
220 (vsFTPd 3.0.3)  
CTRL+C
```

- 14. 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.

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 provision-review start**. This setup script creates a new Git repository needed for the exercise.

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

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	ssh://git.lab.example.com/var/opt/gitlab/git-data/repositories/git/my_webservers_TEST.git
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
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 `http://git.lab.example.com:8081/git/my_webservers_TEST.git`. 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_facts.memtotal_mb }} <br>
Current Free Memory: {{ ansible_facts.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`.

Evaluation

As the **student** user on **workstation**, run the **lab provision-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 provision-review grade
```

This concludes the comprehensive review.

► 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.

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 provision-review start**. This setup script creates a new Git repository needed for the exercise.

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

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	ssh://git.lab.example.com/var/opt/gitlab/git-data/repositories/git/my_webservers_TEST.git
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
DESCRIPTION	Test Webservers Project
ORGANIZATION	Default
SCM TYPE	Git
SCM URL	ssh://git.lab.example.com/var/opt/gitlab/git-data/repositories/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. Click **My Webservers TEST** to edit the Project settings.
 - 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 web servers
JOB TYPE	Run
INVENTORY	Test
PROJECT	My Webservers TEST
PLAYBOOK	apache-setup.yml
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 web servers |
| 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 web servers 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 `http://git.lab.example.com:8081/git/my_webservers_TEST.git`. 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_facts.memtotal_mb }} <br>
Current Free Memory: {{ ansible_facts.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 \
> http://git.lab.example.com:8081/git/my_webservers_TEST.git
Cloning into 'my_webservers_TEST'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), 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_facts.memtotal_mb }} <br>  
Current Free Memory: {{ ansible_facts.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 05c6726] Added Ansible facts  
 1 file changed, 2 insertions(+)
```

6.9. Push the changes to the remote repository using **git push**.

```
[student@workstation my_webservers_TEST]$ git push  
Enumerating objects: 7, done.  
Counting objects: 100% (7/7), done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (4/4), 462 bytes | 462.00 KiB/s, done.  
Total 4 (delta 1), reused 0 (delta 0)  
To http://git.lab.example.com:8081/git/my_webservers_TEST.git  
 57c4e1b..d4be168 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 8.0
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 8.0
Current Memory: 991
Current Free Memory: 529
Current Host: serverd
Server list:
serverd.lab.example.com
serverc.lab.example.com
```

Evaluation

As the **student** user on **workstation**, run the **lab provision-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 provision-review grade
```

This concludes the comprehensive review.

Summary

In this chapter, you learned:

- 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 10

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.

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)

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.

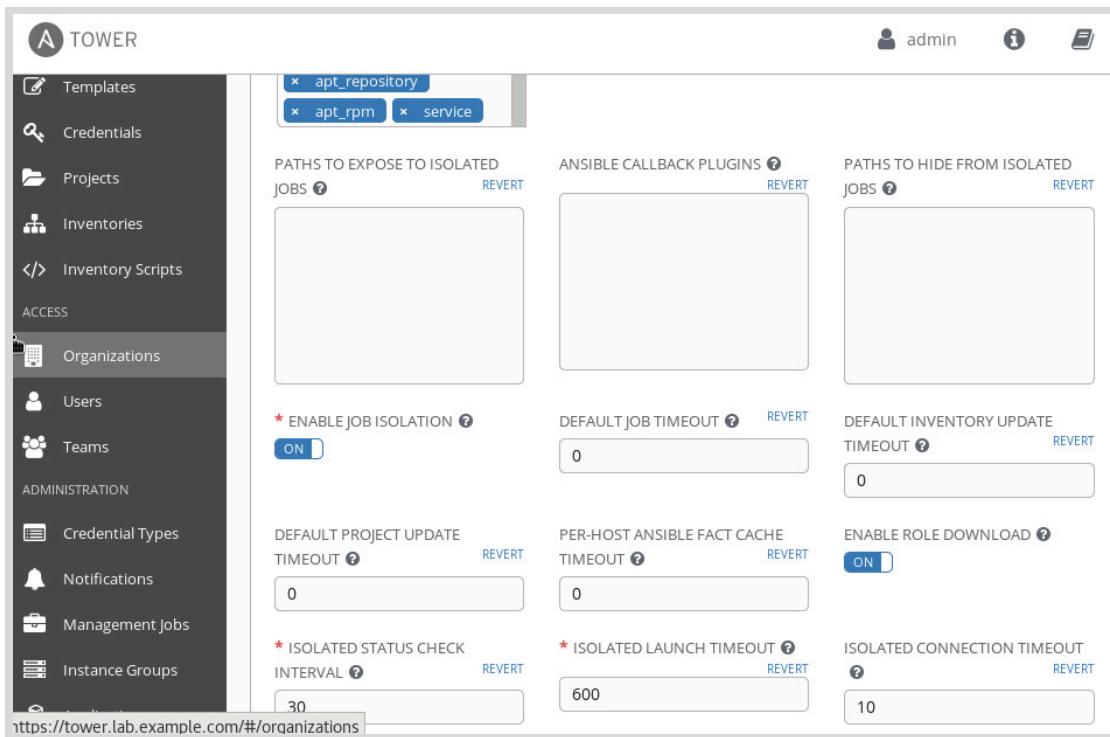


Figure 10.1: Setting Ansible Fact Cache Timeout

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**.

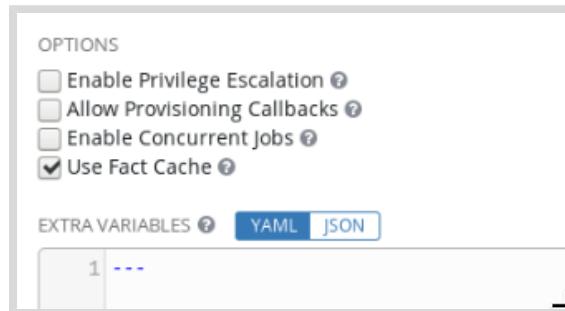


Figure 10.2: Enabling fact caching

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/latest/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 start**.

```
[student@workstation ~]$ lab project-facts start
```

- ▶ 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 | 1 +  
 1 file changed, 1 insertion(+)
```

- 4.4. Edit the **apache-setup.yml** playbook and change the **gather_facts:** variable value to **yes** to enable fact gathering.

- 4.4.1. Open the **apache-setup.yml** playbook for editing.

```
[student@workstation my_webservers_DEV]$ vi apache-setup.yml
```

- 4.4.2. 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...
```

- 4.4.3. 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 6910be1] 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
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
 1cab808..6910be1 master -> master
```

- ▶ 5. From 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. Scroll down and 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.

- 6.1. Open the **apache-setup.yml** playbook for editing.

```
[student@workstation my_webservers_DEV]$ vi apache-setup.yml
```

- 6.2. Change the **gather_facts** variable by setting its value to **no**.

```
---
- hosts: all
  name: Install the web server and start it
  become: yes
  gather_facts: no
  ...output omitted...
```

- 6.3. Save and close the file.
- 6.4. Use **git add --all** to add all files to the staging area.

```
[student@workstation my_webservers_DEV]$ git add --all
```

- 6.5. Use **git commit** to commit the changes; use the comment **Disabling facts gathering**. Push the changes to the remote repository.

```
[student@workstation my_webservers_DEV]$ git commit -m "Disabling facts gathering"
[master 58ba9ea] Disabling facts gathering
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation my_webservers_DEV]$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 299 bytes | 299.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
 6910be1..58ba9ea master -> master
```

- ▶ 7. Launch another job using the **DEV webservers setup** Job Template. 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.
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.
 - 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. 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.

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, then 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:

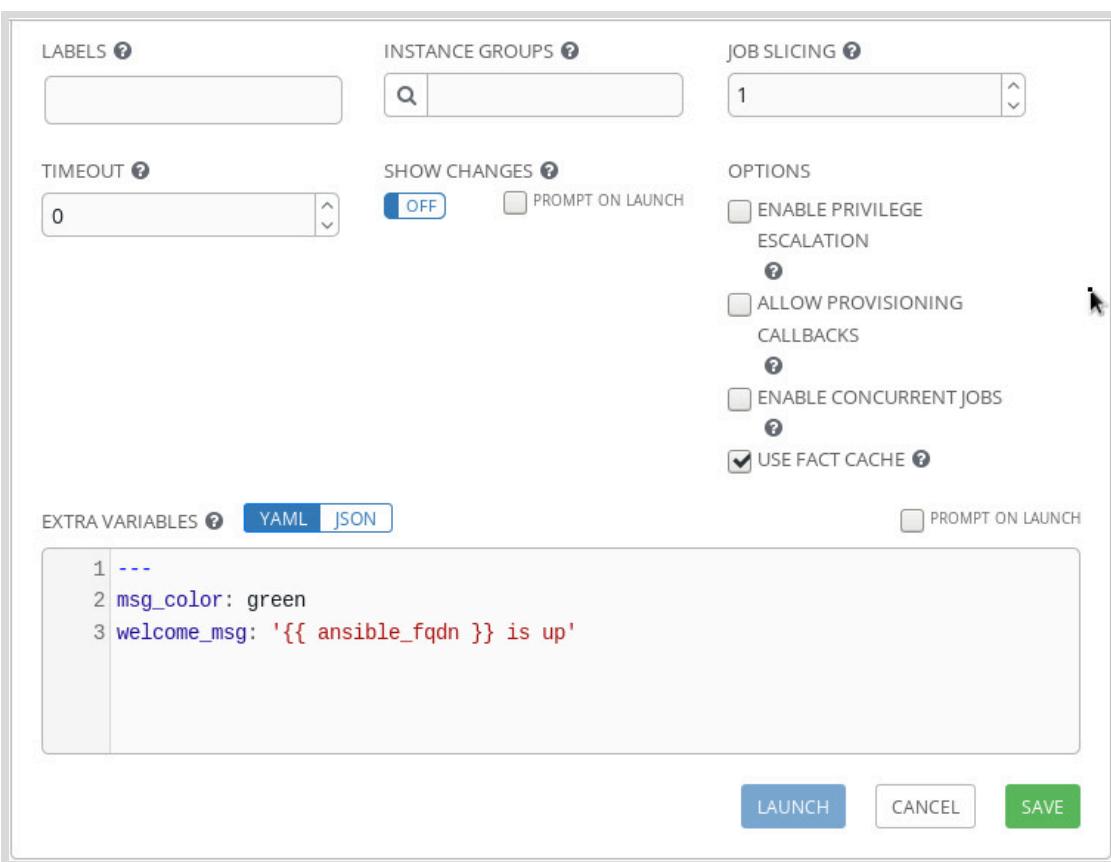


Figure 10.3: Adding extra variables to a Job Template

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:

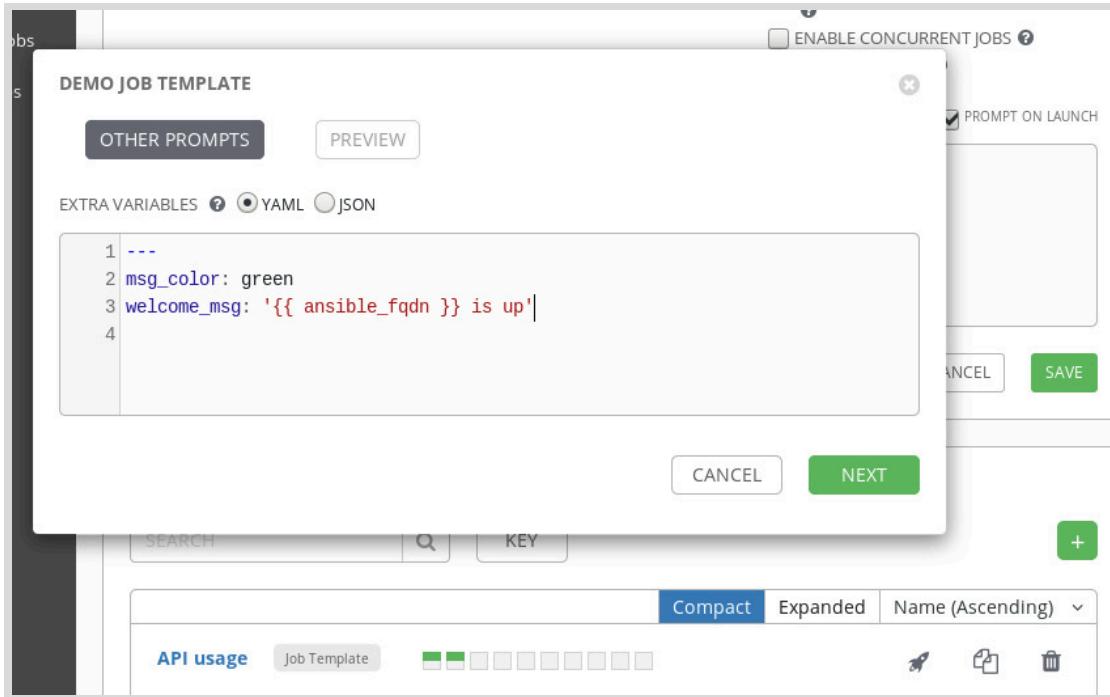


Figure 10.4: Adjusting extra variables on job launch

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.

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, prompting 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.

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.
 - 4.1. In the **PROMPT** field, enter the question to display to the user.

- 4.2. In the **ANSWER VARIABLE NAME** field, enter the name of the extra variable to assign the user's response to.
 - 4.3. From the **ANSWER TYPE** list, select the desired answer type for the user's response.
 - 4.4. If using a list answer type, define the list by entering one list item per line in the **MULTIPLE CHOICE OPTIONS** field.
 - 4.5. 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.
 - 4.6. 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.
 - 4.7. Select **REQUIRED** to specify that a response is required. Clear this field if a response is optional.
 - 4.8. 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 'DEV ftpservers setup | SURVEY' dialog box. The 'SURVEY' toggle switch is set to 'ON'. The 'ADD SURVEY PROMPT' section contains the following fields:

- PROMPT:** Which version do you want to deploy?
- DESCRIPTION:** (empty)
- ANSWER VARIABLE NAME:** version
- ANSWER TYPE:** Text
- MINIMUM LENGTH:** 0
- MAXIMUM LENGTH:** 1024
- DEFAULT ANSWER:** (empty)
- REQUIRED:** checked

The 'PREVIEW' section displays the message: PLEASE ADD A SURVEY PROMPT.

At the bottom of the dialog are 'CLEAR', '+ ADD' (highlighted in green), 'CANCEL', and 'SAVE' buttons.

Figure 10.5: Adding Surveys 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.



Figure 10.6: Survey Prompt



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 will be able to add a Survey to an existing Job Template and launch a Job using a Survey from the Ansible Tower web UI.

Before You Begin

You 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 start** command. This script ensures that the **workstation** and **tower** virtual machines are started.

```
[student@workstation ~]$ lab project-survey start
```

- ▶ 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	deployment_version
ANSWER TYPE	Text
MINIMUM LENGTH	1
MAXIMUM LENGTH	40
DEFAULT ANSWER	v1.0
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. Pull the latest changes from the remote repository.

```
[student@workstation my_webservers_DEV]$ git pull
...output omitted...
```

- 3.3. Edit the **index.html.j2** template.

```
[student@workstation my_webservers_DEV]$ vi templates/index.html.j2
```

- 3.4. Append the following line to the bottom of the file.

```
Deployment Version: {{ deployment_version }} <br>
```

- 3.5. Save the file.

- 3.6. 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"
[master 1c520d5] Depl index page
 1 file changed, 1 insertion(+)
[student@workstation my_webservers_DEV]$ git push
Enumerating objects: 12, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 712 bytes | 712.00 KiB/s, done.
Total 6 (delta 3), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/my_webservers_DEV.git
 8e5d54a..181ec3e 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 will be able to create a Workflow Job Template and launch multiple Ansible jobs as a single workflow.

## 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 serially. 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.

## 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.

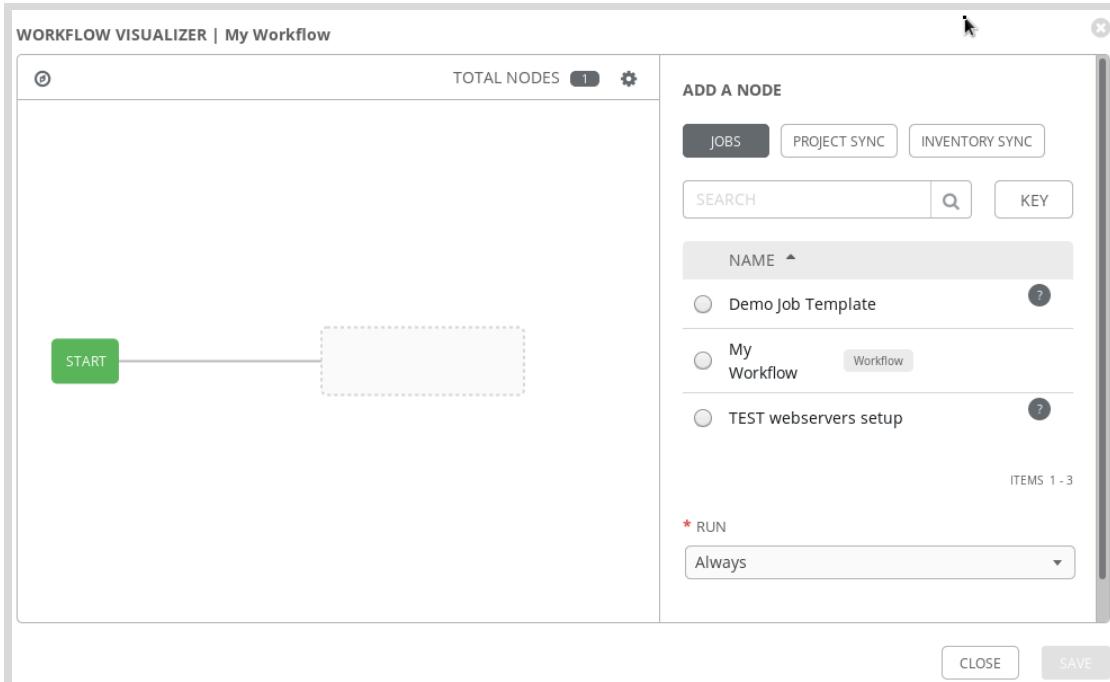
**Figure 10.7: Creating a Workflow Job Template**

Workflow Job Templates are created similarly 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

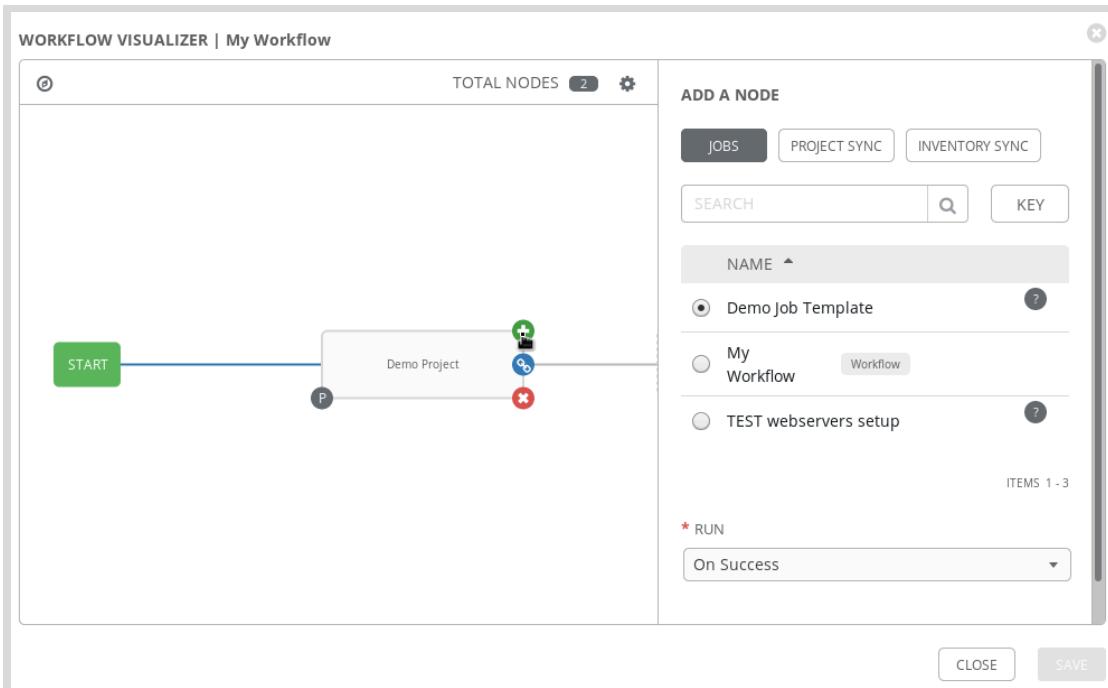
After you have created a Workflow Job Template, the **WORKFLOW VISUALIZER** becomes active in the Workflow Job Template editing screen. The Workflow Editor opens in a new window. The Workflow Visualizer is a graphical interface for defining the job templates to incorporate in a workflow as well as the decision tree structure, which should be used to chain the job templates together.



**Figure 10.8: 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.

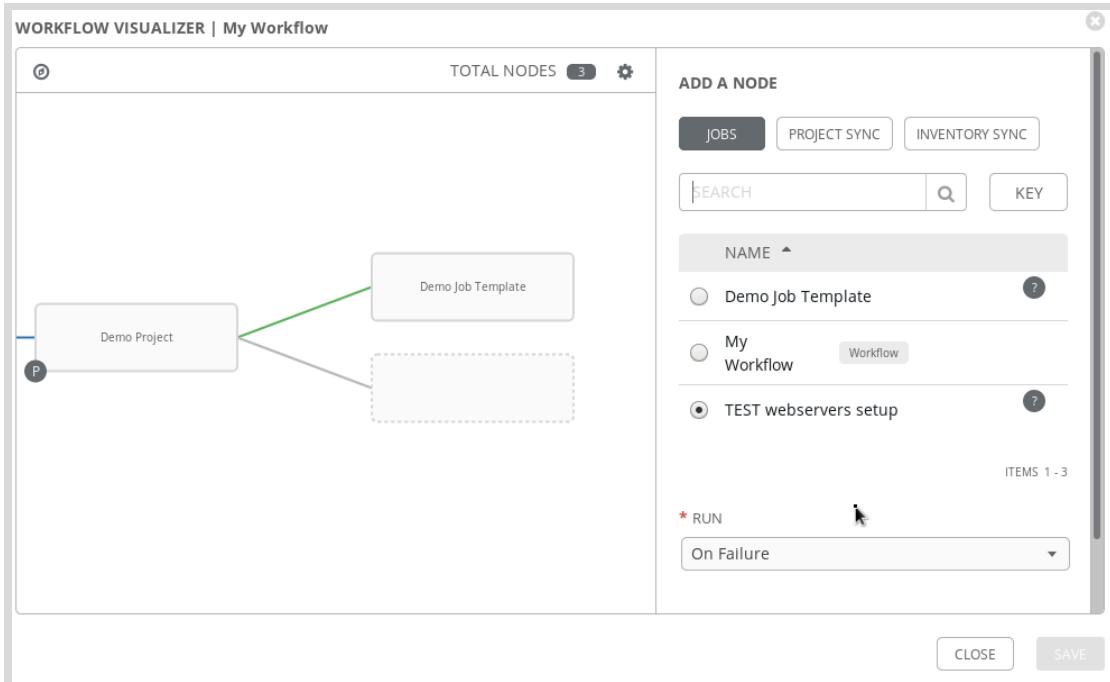


**Figure 10.9: Adding an 'On Success' node to the workflow**

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.

| <b>RUN</b>        | <b>Node Relationship</b>                                                                                   |
|-------------------|------------------------------------------------------------------------------------------------------------|
| <b>On Success</b> | The node resource is executed upon successful completion of the actions associated with the previous node. |
| <b>On Failure</b> | The node resource is executed upon failure of the actions associated with the previous node.               |
| <b>Always</b>     | 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 10.10: 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 10.11: 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.

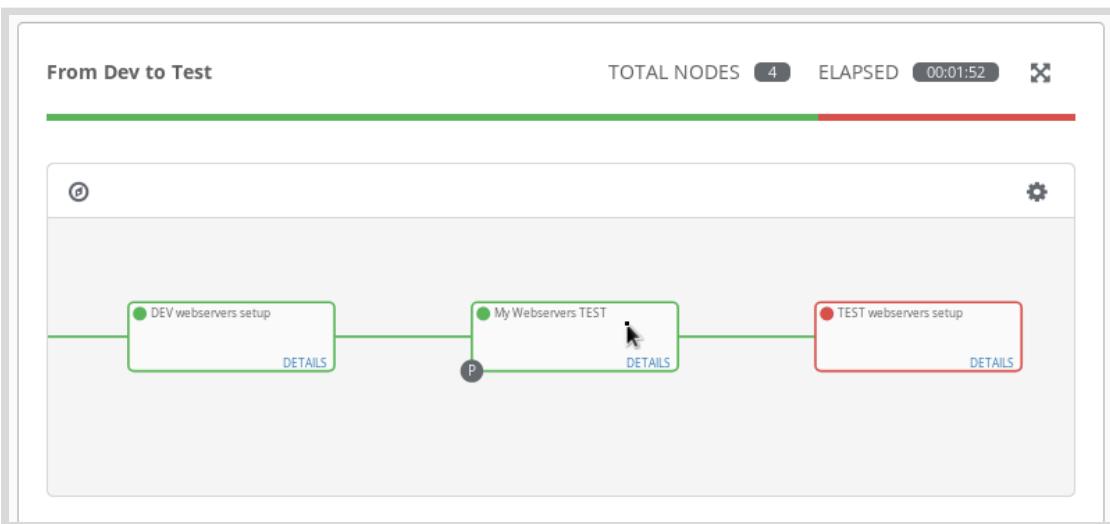
The procedure to launch a Workflow Job Template is similar to how to launch a Job Template

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 10.12: 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.

## Before You Begin

You 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 start**. This script ensures that the **workstation** and **tower** virtual machines are started.

```
[student@workstation ~]$ lab project-workflow start
```

- ▶ 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 | deployment_version: "v1.1"                  |

- 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. The **WORKFLOW VISUALIZER** opens automatically.

- 3.2. Click **START** to add the first action. 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.

## 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.

The screenshot shows the 'CREATE SCHEDULE' dialog for a 'Fact caching job run'. The 'NAME' field is set to 'Fact caching job run'. The 'START DATE' is '6/18/2019' and the 'START TIME' is '0:0:0'. The 'LOCAL TIME ZONE' is 'UTC' and the 'REPEAT FREQUENCY' is 'Day'. In the 'FREQUENCY DETAILS' section, 'EVERY' is set to '1 DAYS' and 'END' is set to 'Never'. The 'SCHEDULE DESCRIPTION' box contains the text 'every day'. At the bottom, it shows 'OCCURRENCES (Limited to first 10)' with '06-18-2019 00:00:00' listed twice.

Figure 10.13: 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.

The screenshot shows the 'NOTIFICATIONS / CREATE NOTIFICATION TEMPLATE' interface. At the top, there's a header with the title and a back arrow. Below it is a 'NEW NOTIFICATION TEMPLATE' form with fields for NAME (with a red asterisk), DESCRIPTION, and ORGANIZATION (set to 'Default'). There's also a 'TYPE' dropdown labeled 'Choose a type'. At the bottom right of the form are 'CANCEL' and 'SAVE' buttons. Below this is a 'NOTIFICATION TEMPLATES' list with one item: 'Notify on Job Success and Failure' (Email type, marked as successful). The list includes columns for NAME, TYPE, and ACTIONS (edit, delete).

**Figure 10.14: 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.

| NAME                              | TYPE  | SUCCESS                             | FAILURE                  |
|-----------------------------------|-------|-------------------------------------|--------------------------|
| Notify on Job Success and Failure | Email | <input checked="" type="checkbox"/> | <input type="checkbox"/> |

Figure 10.15: 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.

## Before You Begin

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 start**. This script will set up the SMTP server.

```
[student@workstation ~]$ lab project-notification start
```

- ▶ 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           | <b>Notify on Job Success and Failure</b>              |
| DESCRIPTION    | <b>Sends an email to notify the status of the Job</b> |
| ORGANIZATION   | <b>Default</b>                                        |
| TYPE           | <b>Email</b>                                          |
| HOST           | <b>localhost</b>                                      |
| RECIPIENT LIST | student@localhost                                     |
| SENDER EMAIL   | system@tower.lab.example.com                          |
| PORT           | <b>25</b>                                             |

- 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
```
- 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@localhost
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 May 23 09:43:37 2019
Return-Path: <system@tower.lab.example.com>
X-Original-To: student@localhost
Delivered-To: student@localhost
Received: from tower.lab.example.com (localhost [IPv6:::1])
 by tower.lab.example.com (Postfix) with ESMTP id 65CAB44ADB2
 for <student@localhost>; Thu, 23 May 2019 09:43:37 +0000 (UTC)
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Job #43 'DEV webservers setup' succeeded:
https://tower.lab.example.com/#/jobs/playbook/43
From: system@tower.lab.example.com
To: student@localhost
Date: Thu, 23 May 2019 09:43:37 -0000
Message-ID: <20190523094337.1609.73858@tower.lab.example.com>

Job #43 had status successful, view details at https://tower.lab.example.com/#/jobs/playbook/43

{
 "id": 43,
 "name": "DEV webservers setup",
 "url": "https://tower.lab.example.com/#/jobs/playbook/43",
 "created_by": "admin",
 "started": "2019-05-23T09:42:58.781684+00:00",
 "finished": "2019-05-23T09:43:36.077699+00:00",
 "status": "successful",
 "traceback": "",
 "inventory": "Dev",
 "project": "My Webservers DEV",
 "playbook": "apache-setup.yml",
 "credential": "Developers",
 "limit": "",
 "extra_vars": "{\"deployment_version\": \"v1.2\"}",
 "hosts": {
 "servera.lab.example.com": {
 "failed": false,
 "changed": 1,
 "dark": 0,
 "failures": 0,
 "ok": 6,
 "processed": 1,
 "skipped": 0
 },
 "serverb.lab.example.com": {
 "failed": false,
 "changed": 1,
 "dark": 0,
 "failures": 0,
 "ok": 6,
 "processed": 1,
 "skipped": 0
 }
 }
}

```

```

 },
 "friendly_name": "Job"
}
CTRL+C
```

- 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       | <b>Automatic job run</b>                                                                                                                                          |
| START TIME | Look at the current time on <b>workstation</b> (possibly by using the <b>date</b> 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 jobs to be executed.

This concludes the guided exercise.

## ▶ 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 will 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.

### Before You Begin

You 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 start**. This script creates all the necessary objects (Git repository, Project, and Job Template) for use with the Prod Inventory.

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

1. Log in to the Ansible Tower web UI running on **tower** as **admin** using **redhat** as the password.
2. 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            | From Test to Prod                            |
| DESCRIPTION     | Deploy to Test and on success deploy to Prod |
| ORGANIZATION    | Default                                      |
| EXTRA VARIABLES | deployment_version: "v1.3"                   |

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.
5. Add a Survey containing the following information to the **From Test to Prod** Workflow Job Template. Make sure this is a required survey.

| 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                                                                   |

- 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.
- Launch a Workflow using the **From Test to Prod** Workflow Job Template. When prompted by the Survey, enter **v1.3** for the deployment version.
- Verify that the Workflow triggers an email notification after completion.
- 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**.

## Evaluation

As the **student** user on **workstation**, run the **lab project-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 project-review grade
```

This concludes the comprehensive review.

## ► 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 will 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.

### Before You Begin

You 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 start**. This script creates all the necessary objects (Git repository, Project, and Job Template) for use with the Prod Inventory.

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

1. Log in to the Ansible Tower web UI running on **tower** as **admin** using **redhat** as the password.
2. 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. If not already selected, 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. If not already selected, 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            | From Test to Prod                            |
| DESCRIPTION     | Deploy to Test and on success deploy to Prod |
| 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            | <b>From Test to Prod</b>                            |
| DESCRIPTION     | <b>Deploy to Test and on success deploy to Prod</b> |
| ORGANIZATION    | <b>Default</b>                                      |
| EXTRA VARIABLES | <b>deployment_version: "v1.3"</b>                   |

- 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** and then 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 **JOB**s 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 **J OBS** 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. Make sure this is a required survey.

| 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                                                                   |

- 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                                                                     |
| 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.

## Evaluation

As the **student** user on **workstation**, run the **lab project-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 project-review grade
```

This concludes the comprehensive review.

# 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 notification mechanisms.
- Ansible Tower provides a browsable REST API that can easily be used to automate Ansible Tower operations and integrate it with third-party products.



# Communicating with APIs using Ansible

### Goal

Interact with REST APIs in Ansible Playbooks, and control Red Hat Ansible Tower using its REST API.

### Objectives

- Control Ansible Tower by accessing its API with `curl`, and in Ansible Playbooks.
- Write playbooks that interact with a REST API to get information from a web service, and to trigger events.

### Sections

- Launching Jobs with the Ansible Tower API (and Guided Exercise)
- Interacting with APIs using Ansible Playbooks (and Guided Exercise)

### Lab

Communicating with APIs using Ansible

# 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.

## 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. The REST API 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 all features of the web UI may not 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/o/"}, {"custom_logo":""}, {"custom_login_info":""}]}
```

The output of the API request is in JSON format, which is readily parseable by computer programs, but may be a little challenging for a human to read.

The Red Hat 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/`. 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 the `curl` command. 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 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" : [
 {
 ...
 },
 ...
],
 "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" : ""
}
```

**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 the value is **null**, you are on the last page. Likewise, the value of **previous** is the URI for the previous page of results, and if the value is **null**, then 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: 1.376s

{
 "count": 61,
 "next": "/api/v2/activity_stream/?page=2",
 "previous": null,
 "results": [
 {
 "id": 1,
 ...
 }
]
}

```

**Figure 11.1: Activity stream API output**

You can click various links in the API to explore related resources.

The screenshot shows a browsable API interface for Ansible Tower. At the top, it says "REST API / Version 2". Below that, "Version 2" has a question mark icon next to it. To the right are buttons for "OPTIONS", "GET", and a dropdown menu. A "GET /api/v2/" button is also present. The main content area shows the response to a GET request:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
X-API-Node: localhost
X-API-Time: 0.013s

{
 "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/",
```

Figure 11.2: 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.



### Important

The documentation available to graphical browsers through the ? icon is very useful when trying to understand how to use the Ansible Tower API. Make use of this resource.

The screenshot shows a REST API documentation page for the 'Job List' endpoint. At the top, there are navigation links for 'REST API / Version 2 / Job List' and a 'Show/Hide Description' button. To the right are 'OPTIONS' and 'GET' method buttons. Below the header, the title 'Job List' is displayed with a question mark icon. On the far right, there is a gear icon. The main content area contains the following text:

**List Jobs:**  
Make a GET request to this resource to retrieve the list of jobs.

The resulting data structure contains:

```
{
 "count": 99,
 "next": null,
 "previous": null,
 "results": [
 ...
]
}
```

The `count` field indicates the total number of jobs found for the given query. The `next` and `previous` fields provide links to additional results if there are more than will fit on a single page. The `results` list contains zero or more job records.

## Results

Each job data structure includes the following fields:

- `id` : Database ID for this job. (integer)

Figure 11.3: 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.

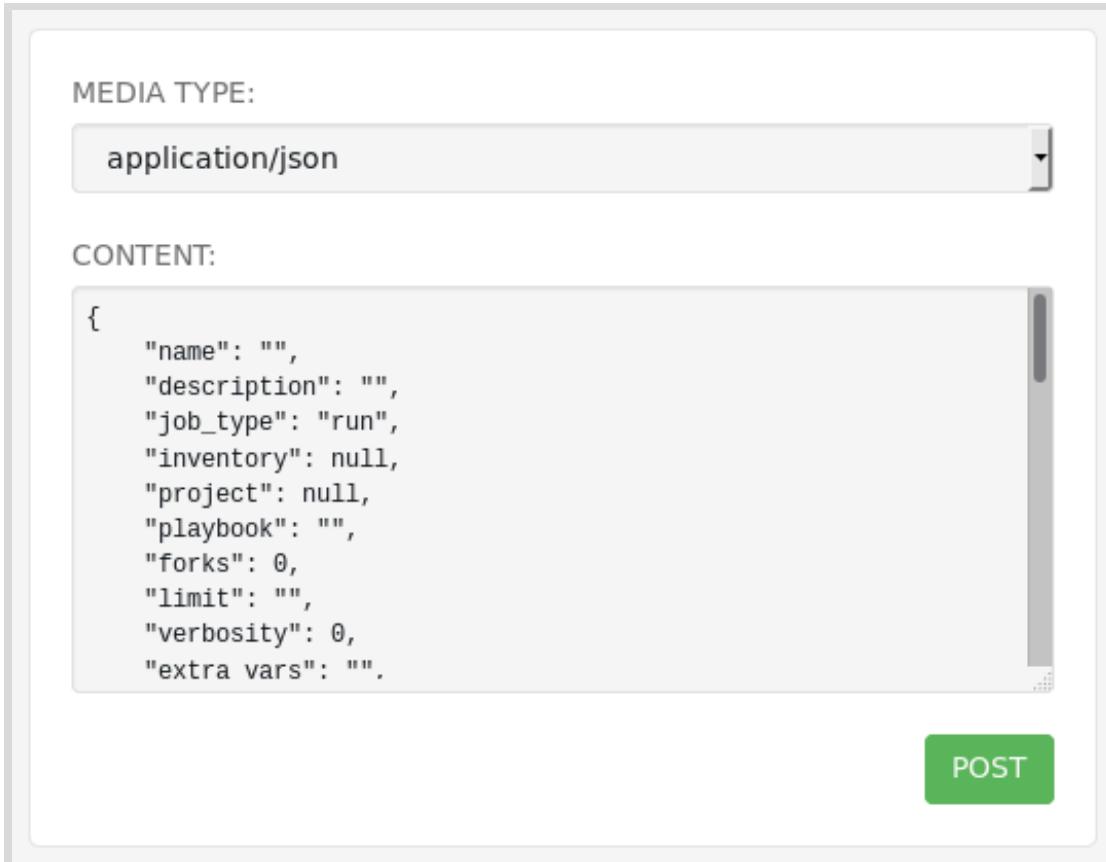


Figure 11.4: Example of POST method text field

## 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 use this method on the **Demo Job Template** Job Template. Because the name of the Job Template contains spaces, you must 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, access it with the GET method to get information about any parameters or data that you need to launch the job. Then, access it with the POST method to launch the job.

The following example uses the GET method to get the parameters that you need 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.

Because no extra information is needed, the job can be launched by accessing the URI with a POST method. Information about the job is returned. In particular, note that the job **id** (72) returns a pending **status**, because it has been launched but has not yet completed.

```
[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/extracredentials/",
"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 is 72. You can use the job id to retrieve updated status information, such as whether the job has completed. For job 72, use the URI **/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 using only the ID number.

First, 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 and use it to launch another Job Template as one of its tasks.

In the playbook, specify the correct URL to your Job Template, using the ID or the Named URL. You must also 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 requires 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 requires the URL of the actual API to which Ansible must connect. 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. You can also use the **urlencode** filter: **tower\_job: "{{ 'Demo Job Template' | urlencode }}"**
- ③ Ansible can access the Ansible Tower API from the Ansible Tower server 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 must set up a Vault Credential in Ansible Tower that can decrypt those files. You must also configure any Job Templates that use the Project with a Vault Credential, in addition to any machine credentials or other credentials that Project needs.

First, 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' interface. A modal window is open for a credential named 'vault-credential-demo'. The 'DETAILS' tab is active. The 'NAME' field contains 'vault-credential-demo', 'DESCRIPTION' contains 'Vault Credentials', and 'ORGANIZATION' is set to 'Default'. Under 'CREDENTIAL TYPE', 'Vault' is selected. In the 'TYPE DETAILS' section, 'VAULT PASSWORD' is set to 'Prompt on launch' and 'ENCRYPTED' is checked. 'VAULT IDENTIFIER' is empty. At the bottom are 'CANCEL' and 'SAVE' buttons.

Figure 11.5: 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. This Job Template must include the Vault Credential required to decrypt the project files as one of its Credentials.

Figure 11.6: 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.

## Token-based Authentication

Since Red Hat Ansible Tower 3.3, the API uses OAuth 2 to provide token-based authentication. Any call to the API with a valid token in the headers of the request will be authenticated. There are two kinds of tokens: *Application Tokens* and *Personal Access Tokens*.

Application Tokens are requested for an application that was previously created in Tower, and represents a client application that will access the API frequently with multiple users. Personal Access Tokens (PAT) are a much simpler mechanism, providing access to the API for a single user.

This example shows a request for a Personal Access Token, as well as how to use the issued token to launch a Job Template.

```

- name: Tower API
 hosts: localhost
 gather_facts: false

 vars:
 tower_user: admin
 tower_pass: redhat
 tower_host: tower.lab.example.com
 template_name: DEV ftpservers setup

 tasks:
 - name: Get the token
 uri:
 url: "https://{{ tower_host }}/api/v2/users/1/personal_tokens/"
 method: POST
 validate_certs: false
 return_content: true
 user: "{{ tower_user }}"
 password: "{{ tower_pass }}"
 force_basic_auth: true 1
 status_code: 201
 register: response 2

 - name: Use the token
 uri:
 url: "https://{{ tower_host }}/api/v2/job_templates/{{ template_name }}|urlencode|/launch/"
 method: POST
 validate_certs: false
 return_content: true
 status_code: 201
 headers:
 Authorization: "Bearer {{ response['json']['token'] }}" 3
 Content-Type: "application/json"
 register: launch

```

- 1** We use the regular authentication mechanism.
- 2** Save the result for later use.
- 3** Use the token in the response variable to provide the authentication.



## References

### Ansible Tower API Guide

<https://docs.ansible.com/ansible-tower/latest/html/towerapi/>

### Token-Based Authentication

[https://docs.ansible.com/ansible-tower/latest/html/administration/oauth2\\_token\\_auth.html](https://docs.ansible.com/ansible-tower/latest/html/administration/oauth2_token_auth.html)

## ► 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:

- Launch a job from an existing job template by running an Ansible Playbook.

## Before You Begin

Ensure that the **workstation** and **tower** virtual machines are started.

Log in to the **workstation** machine as the **student** user, using **student** as the password.

On the **workstation** machine, run the **lab api-tower start** command, which verifies that Ansible Tower services are running and all the required resources are available. The command also creates additional Vault credentials and uploads a new playbook into the Git repository.

```
[student@workstation ~]$ lab api-tower start
```

- 1. In the first part of this exercise, you will directly use the REST API provided by Ansible Tower.

On the **workstation** machine, use Firefox to view the resources available from your Ansible Tower server's API.

- 1.1. 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 the user **admin**, using **redhat** as the password.
- 1.2. Click the **/api/v2/** link to access the browsable list of all the available resources accessible through the API.

```
...output omitted...
{
 "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 URI can 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 that provides a link with the Job Template name, which 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 link sends a GET request to the **launch** resource for that Job Template, providing details about what information is needed to launch a job from the template.
  - 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 the user **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 the **curl** command, ensure the **perl-JSON-PP** package is installed on **workstation**.

```
[student@workstation ~]$ sudo yum install perl-JSON-PP
```

Enter the password for the **student** user, **student**, when you are prompted. Review the packages that will install, if any, and enter a **y** character when prompted to begin package installation.

- 3.2. Use the API with a **name** filter to search for the **Demo Job Template** Job Template. Determine the Job Template's ID. This ID should be the same as the one that you saw using Firefox earlier in this exercise. For your convenience, you can copy and paste the following command from the **~/DO447/labs/api-tower/curl\_commands** file.

```
[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
```

In the previous command, the **-k (--insecure)** option instructs **curl** to operate even if it cannot verify the SSL certificate. With the **-s (--silent)** option, **curl** only displays the returned data and not the progress status or the error messages.

- 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...
```

Alternatively, you can refer to the job template by its name in the API: [https://tower.lab.example.com/api/v2/job\\_templates/"Demo Job Template"/](https://tower.lab.example.com/api/v2/job_templates/'Demo Job Template'/)

- 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 the **curl** command should now be at the top of the list of jobs that have 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 uses the Ansible Tower API to remotely launch a new job based on an existing Job Template. Because the login and the password to access the API are stored in this playbook, it is encrypted using the **ansible-vault** command with a password of **redhat**.

- 4.1. *Optional.* Clone the repository at `http://git.lab.example.com:8081/git/my_webservers_DEV.git` in the `/home/student/git-repos` directory. If you have already cloned the repository in a previous exercise, pull the new contents instead.

```

[student@workstation ~]$ cd ~/git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/my_webservers_DEV.git
[student@workstation git-repos]$ cd my_webservers_DEV
[student@workstation my_webservers_DEV]$ git pull

```

Inspect the contents of the **tower\_api.yml** playbook and notice that the file is encrypted:

```

[student@workstation my_webservers_DEV]$ cat tower_api.yml
$ANSIBLE_VAULT;1.1;AES256
383731626263376661623562393439393533343265323161656634396462633664336237393830
61353031313931363866353839336664623134666563640a303465346463643632386231636634
33393435333633363265363835353032663566313831626365386666333963373434663830356337
3765313537643265610a616331383464386432643266326431363665386430656537366664613339
...output omitted...

```

Use the **ansible-vault view** command to display the contents of the playbook:

```

[student@workstation my_webservers_DEV]$ ansible-vault view tower_api.yml
Vault password: redhat
...output omitted...
vars:
 tower_user: admin

```

```

tower_pass: redhat
tower_host: tower.lab.example.com
tasks:
 - 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
...output omitted...

```

- 4.2. 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.

Click **Templates** in the left navigation bar.

- 4.3. Click the **API usage** Job Template to edit the settings.
- 4.4. Click the magnifying glass button under **CREDENTIAL**.
- 4.5. In the **CREDENTIALS** window, click the **CREDENTIAL TYPE** list to access the list of available credential types.
- 4.6. 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 decrypt the provided and encrypted **tower\_api.yml** playbook to access its own API to start a new job.

- 4.7. 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.

In addition to the **API usage** and the **DEV ftpservers setup** jobs, you may notice that Ansible Tower launches other jobs to update the associated Git projects.

## Finish

On **workstation**, run the **lab api-tower finish** script to clean up this exercise.

```
[student@workstation ~]$ lab api-tower finish
```

This concludes the guided exercise.

# Interacting with APIs using Ansible Playbooks

## Objectives

After completing this section, students should be able to write playbooks that interact with a REST API to get information from a web service and trigger events.

### Interacting with an API

Red Hat Ansible Engine can interact with any HTTP API provided by a service, including RESTful APIs. This interaction may be required for a variety of reasons. For example, the service might be external to the network running the Ansible managed systems, there might be no specific Ansible module to interact with the service, or the Ansible module might not expose needed functionality.

To access these APIs, Ansible offers the **uri** module. This module connects to a given URL, controlling the parameters of the connection and operating on the response. The only required parameter is **url**, which indicates the full HTTP or HTTPS URL to which to connect.

This is a very simple example task using the **uri** module:

```
- name: Check that the page is reachable and returns a status 200 using GET
 uri:
 url: http://www.example.com
```

However, usually you must specify the **method** parameter to the Ansible module, which is used to specify the HTTP method used to connect to the server. The most common options available for the **method** parameter are:

#### GET

Method to obtain an entity from a service identified by the URL of the request. This is the default value.

#### POST

Ask the service to store the entity contained in the body of the request under the resource identified by the URL.

#### PUT

Ask the service to store the entity sent in the body of the request as the resource identified by the URL, modifying it, if it exists.

#### DELETE

Deletes an entity in the service identified by the URL of the request.

#### PATCH

Modify the entity identified by the request URL with the values in the body. Only modified values must be in the body.

The next option available to control how to connect to the service uses the **headers** parameter. This parameter is a dictionary where custom HTTP headers can be added to the request. For example:

```
headers:
 Cookie: type=Test
```

The following is a more advanced example, using the parameters covered so far:

```
- name: Get an entity and set a Cookie
 uri:
 url: https://example.com/entity/1
 method: GET
 headers:
 Cookie: type=TEST
```

## Send Information to an API

Now that you know how to call an API, you can start sending information to any API. There are two mutually exclusive parameters to send this information: **src** and **body**.

The **src** option points to a file that contains the body of the HTTP request that you want to make. Alternatively, you can use the **body** option, which defines the body of the HTTP request in your playbook in YAML syntax.

Depending on the format expected by the receiving service, you might need to use the parameter **body\_format**. The options for this parameter are: **raw**, **json** and **form-urlencoded**. For REST APIs, use **json**, and for traditional form-based pages use **form-urlencoded**.

For example, login into a service could be achieved with the following:

```
- name: Login to a form-based webpage
 uri:
 url: https://example.com/login.php
 method: POST
 body_format: form-urlencoded
 body:
 name: your_username
 password: your_password
 enter: Sign in
```

## Handling API Responses

The first kind of information returned by any HTTP service is the status code of the response. Use the **status\_code** option to tell the **uri** module the status code you expect on success. If the status code in the response is different, the task will fail.

Secondly, you must handle the response itself.

If you want to save the response as a file, specify that with the **dest** parameter.

To use the response in the playbook, use the **return\_content** option to indicate that the body of the response should be added to the result dictionary. Save that in a variable by using **register**.

Here you can see an example of how to use the response of a request:

```
- name: Check the contents of the response
uri:
 url: http://www.example.com
 return_content: yes
register: response
failed_when: "'SUCCESS' not in response.content"
```

The following is another practical example that parses data returned from a GitLab APIv4 call as variables in Ansible. The variable **my\_private\_token** is a Personal Access Token set up in the GitLab interface, and presumably loaded into the play from a file protected with Ansible Vault. The JSON the API call returns is a list of dictionaries, where each dictionary contains information about one user; the key **username** contains the user's username.

```
- name: Use GitLab API to get user data
uri:
 url: https://git.lab.example.com/api/v4/users/
 method: GET
 headers:
 Private-Token: "{{ my_private_token }}"
 return_content: yes
register: gitlab_api_result

- name: Print user names
 debug:
 msg: A valid username is {{ item['username'] }}
 loop: "{{ gitlab_api_result['json'] }}"
```

## HTTP Security Settings

The **uri** module also supports Digest, Basic or WSSE authentication, which can be controlled by using the **url\_username** and **url\_password** parameters. If the remote service supports Basic authentication but this module fails to authenticate, then try forcing Basic authentication using the **force\_basic\_auth** parameter.

A more secure option is to use private keys to establish secure connections to the server. Refer to the PEM certificate chain file by using the **client\_cert** parameter. If the certificate chain file does not contain the key, point the module to the file storing the key using the **client\_key** parameter.

Finally, if you must avoid TLS certificate validation, then set the **validate\_certs** parameter to **false**. This setting reduces the security of your connection.

## Preparing and Parsing Data with Filters

In previous chapters, you have seen how to use filters to transform data. There are several filters that are useful when dealing with HTTP and REST APIs.

URLs support a limited subset of the US-ASCII character set. To ensure that a URL is correctly encoded, use the **urlencode** filter.

```

- name: Using the uri module with urlencode
 hosts: localhost
```

```
vars:
 entity_name: Test spaces
tasks:
 - name: URL call
 uri:
 url: "http://example.com?name={{ entity_name | urlencode }}"
```

The **to\_json** and **from\_json** filters can also be useful when marshaling data to and from an API.

The **xml** module can also be useful, in conjunction with filters, to process data from some APIs.



## References

### URI Module

[https://docs.ansible.com/ansible/2.8/modules/uri\\_module.html](https://docs.ansible.com/ansible/2.8/modules/uri_module.html)

### HTTP Methods Definitions

<https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

### HTTP Status Code Definitions

<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

### Filters

[https://docs.ansible.com/ansible/2.8/user\\_guide/playbooks\\_filters.html](https://docs.ansible.com/ansible/2.8/user_guide/playbooks_filters.html)

## ► Guided Exercise

# Interacting with APIs using Ansible Playbooks

In this exercise, you will write playbooks that interact with the REST API of a Red Hat Ansible Tower server, in order to operate with jobs and templates.

## Outcomes

You should be able to use the Tower REST API to perform operations that are not available in the Ansible plugins.

## Before You Begin

Open a terminal on the **workstation** machine as the **student** user, and run the following command:

```
[student@workstation ~]$ lab api-interaction start
```

- 1. Clone the Git repository `http://git.lab.example.com:8081/git/api-interaction.git` in the `/home/student/git-repos` directory.
  - 1.1. From a terminal, create the directory `/home/student/git-repos` if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- 1.2. Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- 1.3. Clone the repository:

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/api-interaction.git
Cloning into 'api-interaction'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 15 (delta 7), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
[student@workstation git-repos]$ cd api-interaction
```

- 2. Review the contents of the `inventory.yml` file.

```

tower: ①
hosts:
 tower.lab.example.com:
vars:
 template_name: DEV ftpservers setup ②
 copy_template_name: Exact copy of DEV ftpservers setup ③
 tower_fqdn: tower.lab.example.com ④
 tower_user: admin ⑤
 tower_password: redhat ⑥

```

- ① Host group to execute the calls.
- ② Name of the existing template.
- ③ Name of the template to create.
- ④ API URL.
- ⑤ User to login.
- ⑥ Password of the user.

- 3. Modify the **tower\_copy\_template.yml** playbook to add the tasks to copy and launch job templates. For your convenience, the already edited file is available under the **~/DO447/solutions/api-interaction/** directory.

```
[student@workstation api-interaction]$ vi tower_copy_template.yml
```

- 3.1. The first task in the playbook calls the Tower API to copy a template. The URL of that action is `/api/v2/job_templates/<template-name>/copy/`. The task should look like:

```

- name: Copy an existing Ansible Tower Job Template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ template_name }} | urlencode ① }}/copy/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: [201, 202]
 body:
 name: "{{ copy_template_name }}"
 body_format: json
 register: copy ②

```

- ① Notice how the name of the template is encoded to use in a URL. Also notice that URLs for the Ansible Tower API must always end with a final / character.
- ② Register API call result for later use.

- 3.2. The second task of the playbook launches a job in the newly created template. The URL for this actions is `/api/v2/job_templates/<template-name>/launch/`. The task should look like:

```

- name: Start a new Ansible Tower Job using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name }} |"
 | urlencode }}/launch/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: [201, 202]
 body:
 inventory: "{{ copy.json.inventory }}" ②
 body_format: json

```

- ①** Now use the name of the copied template.
- ②** Use the inventory ID returned by the copy call.

3.3. The whole file should look like this:

```

- name: Using the uri module to connect to tower
 hosts: tower
 gather_facts: false

 tasks:
 - name: Copy an existing Ansible Tower Job Template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ template_name }} |"
 | urlencode }}/copy/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: [201, 202]
 body:
 name: "{{ copy_template_name }}"
 body_format: json
 register: copy

 - name: Start a new Ansible Tower Job using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name }} |"
 | urlencode }}/launch/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: [201, 202]

```

```
body:
 inventory: "{{ copy.json.inventory }}"
body_format: json
```

- 3.4. Verify the syntax and execute the playbook using the **ansible-playbook** command:

```
[student@workstation api-interaction]$ ansible-playbook --syntax-check \
> tower_copy_template.yml

playbook: tower_copy_template.yml
[student@workstation api-interaction]$ ansible-playbook tower_copy_template.yml

PLAY [Using the uri module to connect to tower] *****

TASK [Copy an existing Ansible Tower Job Template using the uri module] ****
ok: [tower.lab.example.com]

TASK [Start a new Ansible Tower Job using the uri module] ****
ok: [tower.lab.example.com]

PLAY RECAP *****
tower.lab.example.com : ok=2 changed=0 ...
```



### Note

The **tower\_copy\_template.yml** playbook is not idempotent. In case you have made an error and the template was incorrectly created, delete the template, before running the corrected playbook again.

- 4. Modify the **tower\_add\_survey.yml** playbook to add a survey to the job template and activate it. For your convenience, the already edited file is available under the **~/DO447/solutions/api-interaction/** directory.

```
[student@workstation api-interaction]$ vi tower_add_survey.yml
```

- 4.1. Add a task to the playbook that creates a survey for the new template. The URL for this action is `/api/v2/job_templates/<template-name>/survey_spec/`. The task should look like:

```
- name: Add a survey to the existing Ansible Tower Job Template using the uri
 module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name | urlencode }}/survey_spec/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: 200
```

```

body:
 name: "What is the purpose of this deployment?"
 description: "This purpose will be displayed in the welcoming message"
 spec:
 - type: "text"
 question_name: "What is the purpose of this deployment?"
 question_description: "This purpose will be displayed in the welcoming message"
 variable: "deployment_purpose"
 required: true
 choices: ""
 new_question: true
 min: 1
 max: 40
 default: "Testing"
 body_format: json

```

- 4.2. Then, add a second task to enable the survey in the job template. This is done modifying the survey status in the template. The task should look like this:

```

- name: Enable the survey
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name }}|urlencode }}"
 validate_certs: no
 method: PATCH ①
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: 200
 body:
 survey_enabled: true ②
 body_format: json

```

- ① Use PATCH to modify the state of an object in a REST API.
- ② Only the state of the survey is modified.

- 4.3. The whole file should look like:

```

- name: Using the uri module to add a survey
 hosts: tower
 gather_facts: false

 tasks:
 - name: Add a survey to the existing Ansible Tower Job Template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name }}|urlencode }}/survey_spec/"
 validate_certs: no
 method: POST
 return_content: yes

```

```

force_basic_auth: yes
user: "{{ tower_user }}"
password: "{{ tower_password }}"
status_code: 200
body:
 name: "What is the purpose of this deployment?"
 description: "This purpose will be displayed in the welcoming message"
 spec:
 - type: "text"
 question_name: "What is the purpose of this deployment?"
 question_description: "This purpose will be displayed in the
welcoming message"
 variable: "deployment_purpose"
 required: true
 choices: ""
 new_question: true
 min: 1
 max: 40
 default: "Testing"
 body_format: json

- name: Enable the survey
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name
| urlencode }}/"
 validate_certs: no
 method: PATCH
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: 200
 body:
 survey_enabled: true
 body_format: json

```

- 4.4. Verify the syntax and execute the playbook using the **ansible-playbook** command:

```

[student@workstation api-interaction]$ ansible-playbook --syntax-check \
> tower_add_survey.yml

playbook: tower_add_survey.yml
[student@workstation api-interaction]$ ansible-playbook tower_add_survey.yml

PLAY [Using the uri module to add a survey] ****
TASK [Add a survey to the existing Job Template using the uri module] ****
ok: [tower.lab.example.com]

TASK [Enable the survey] ****
ok: [tower.lab.example.com]

```

```
PLAY RECAP ****
tower.lab.example.com : ok=2 changed=0 ...
```

- 4.5. Add a new task to the **tower\_add\_survey.yml** playbook that executes the job template adding the variable for the survey. Tag the new task so it can be executed separately. The new task should look like this:

```
- name: Start a new Ansible Tower Job using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name }}|urlencode }}/launch/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: [201, 202]
 body:
 inventory: 4
 extra_vars:
 deployment_purpose: "Testing" ①
 body_format: json
 tags: with_variable ②
```

- ① Extra variables needed for the survey.
- ② Tag to execute separately.

- 4.6. Verify the syntax and execute the playbook using the **ansible-playbook** command:

```
[student@workstation api-interaction]$ ansible-playbook --syntax-check \
> tower_add_survey.yml

playbook: tower_add_survey.yml
[student@workstation api-interaction]$ ansible-playbook tower_add_survey.yml \
> --tags with_variable

PLAY [Using the uri module to connect to tower] *****

TASK [Start a new Ansible Tower Job using the uri module] *****
ok: [tower.lab.example.com]

PLAY RECAP ****
tower.lab.example.com : ok=1 changed=0 ...
```

- ▶ 5. Modify the **tower\_template\_cleanup.yml** playbook to delete the copied template. For your convenience, the already edited file is available under the **~/DO447/solutions/api-interaction/** directory.

```
[student@workstation api-interaction]$ vi tower_template_cleanup.yml
```

- 5.1. Add a task to the playbook that deletes the template created at the beginning of the exercise. The task should look like:

```
- name: Delete an existing Ansible Tower Job Template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name | urlencode }}/"
 validate_certs: no
 method: DELETE
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: 204
```

- ① Use the **DELETE** method to delete the template.

- 5.2. The whole file should look like this:

```

- name: Using the uri module to delete a Job Template
 hosts: tower
 gather_facts: false

 tasks:
 - name: Delete an existing Job Template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ copy_template_name | urlencode }}"
 validate_certs: no
 method: DELETE
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: 204
```

- 5.3. Verify the syntax and execute the playbook using the **ansible-playbook** command:

```
[student@workstation api-interaction]$ ansible-playbook --syntax-check \
> tower_template_cleanup.yml

playbook: tower_template_cleanup.yml
[student@workstation api-interaction]$ ansible-playbook tower_template_cleanup.yml
PLAY [Using the uri module to delete a Job Template] ****
TASK [Delete an existing Job Template using the uri module] ****
ok: [tower.lab.example.com]

PLAY RECAP ****
tower.lab.example.com : ok=1 changed=0 ...
```

- 6. Commit the changes back to the repository:

```
[student@workstation api-interaction]$ git add .
[student@workstation api-interaction]$ git commit -m "Tower API interaction"
[master a1fd4d7] Tower API interaction
 3 files changed, 94 insertions(+), 3 deletions(-)
[student@workstation api-interaction]$ git push
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.09 KiB | 1.09 MiB/s, done.
...output omitted...
```

## Finish

On **workstation**, run the **lab api-interaction finish** script to complete this lab.

```
[student@workstation api-interaction]$ lab api-interaction finish
```

This concludes the exercise.

## ▶ Lab

# Communicating with APIs using Ansible

## Performance Checklist

In this lab, you will fix a simple playbook to interact with the REST API of Ansible Tower to ensure that particular resources exist.

## Outcomes

You should be able to use the provided playbooks to create and modify resources in Ansible Tower using the Ansible **uri** module.

## Before You Begin

Log in as the **student** user on the **workstation** machine, and run the **lab api-review start** command.

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

This script initializes the remote Git repository you need for this lab, `http://git.lab.example.com:8081/git/api-review.git`. The Git repository contains playbooks that you must fix in order to use the Ansible Tower REST API.

1. Clone the playbook project repository.
2. Using Ansible **uri** module, create a copy of the existing **New template** Job Template using the Ansible Tower REST API. To create the playbooks, use variables, filters, and any other applicable Ansible best practices from this course.

Use the following names and locations to modify the existing playbooks and inventories. Notice that the names are case sensitive.

- The name of the Job Template to copy is **New template**.
- Name the copied Job Template **Review template**.
- For all operations, use the Ansible Tower user **simon**.

Use the REST API URL `https://TOWER_NAME/api/v2/job_templates/TEMPLATE_NAME | FILTER/copy/`

Start with modifications to the provided **inventory.yml** file. Modify the file content by replacing the variable values with the appropriate values. Then, modify the contents of the **copy\_template.yml** playbook to use these variables.

3. Copy the Job Template using that modified playbook.
4. Modify the provided **template\_cleanup.yml** playbook to delete the original **New template** template. Use the **DELETE** method.
5. Use the modified playbook to remove the original Job Template.
6. Save, commit, and push your changes to the remote repository.

## Evaluation

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

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

## Finish

From the **workstation** machine, run the **lab api-review finish** command to complete this lab.

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

This concludes the lab.

## ► Solution

# Communicating with APIs using Ansible

### Performance Checklist

In this lab, you will fix a simple playbook to interact with the REST API of Ansible Tower to ensure that particular resources exist.

### Outcomes

You should be able to use the provided playbooks to create and modify resources in Ansible Tower using the Ansible **uri** module.

### Before You Begin

Log in as the **student** user on the **workstation** machine, and run the **lab api-review start** command.

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

This script initializes the remote Git repository you need for this lab, `http://git.lab.example.com:8081/git/api-review.git`. The Git repository contains playbooks that you must fix in order to use the Ansible Tower REST API.

1. Clone the playbook project repository.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos
[student@workstation ~]$ cd /home/student/git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/api-review.git
...output omitted...
[student@workstation git-repos]$ cd api-review
[student@workstation api-review]$
```

2. Using Ansible **uri** module, create a copy of the existing **New template** Job Template using the Ansible Tower REST API. To create the playbooks, use variables, filters, and any other applicable Ansible best practices from this course.

Use the following names and locations to modify the existing playbooks and inventories. Notice that the names are case sensitive.

- The name of the Job Template to copy is **New template**.
- Name the copied Job Template **Review template**.
- For all operations, use the Ansible Tower user **simon**.

Use the REST API URL `https://TOWER_NAME/api/v2/job_templates/TEMPLATE_NAME | FILTER/copy/`

Start with modifications to the provided **inventory.yml** file. Modify the file content by replacing the variable values with the appropriate values. Then, modify the contents of the **copy\_template.yml** playbook to use these variables.

#### 2.1. Modify the content of the **inventory.yml** file.

```
[student@workstation api-review]$ vi inventory.yml
```

```
tower:
 hosts:
 tower.lab.example.com:
 vars:
 template_name: New template
 copy_template_name: Review template
 tower_fqdn: tower.lab.example.com
 tower_user: simon
 tower_password: redhat123
```

- 2.2. Modify the provided **copy\_template.yml** playbook to add a task to copy the existing job template. Use the **POST** method to call the proper Ansible Tower API and variables from the inventory file.

```
[student@workstation api-review]$ vi copy_template.yml
```

The first task in the playbook calls the Tower API to copy a template. The URL for this action is `/api/v2/job_templates/<template-name>/copy/`. The task should look like:

```
- name: Copy the existing New template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ template_name | urlencode }}/copy/"
 validate_certs: no
 method: POST
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: [201, 202]
```

```
body:
 name: "{{ copy_template_name }}"
 body_format: json
```

3. Copy the Job Template using that modified playbook.

```
[student@workstation api-review]$ ansible-playbook copy_template.yml

PLAY [Using the uri module to connect to tower] *****

TASK [Copy the existing New template using the uri module] *****
ok: [tower.lab.example.com]

PLAY RECAP *****
tower.lab.example.com : ok=1 changed=0 unreachable=0 failed=0 ...
```

4. Modify the provided **template\_cleanup.yml** playbook to delete the original **New template** template. Use the **DELETE** method.

- 4.1. Edit the provided **template\_cleanup.yml** playbook.

Add a task to the playbook that deletes the original template. The task should look like:

```
- name: Delete an existing Ansible Tower Job Template using the uri module
 uri:
 url: "https://{{ tower_fqdn }}/api/v2/job_templates/{{ template_name | urlencode }}/"
 validate_certs: no
 method: DELETE
 return_content: yes
 force_basic_auth: yes
 user: "{{ tower_user }}"
 password: "{{ tower_password }}"
 status_code: 204
```

5. Use the modified playbook to remove the original Job Template.

```
[student@workstation api-review]$ ansible-playbook template_cleanup.yml

PLAY [Using the uri module to delete a Job Template] *****

TASK [Delete an existing Ansible Tower Job Template using the uri module] *...
ok: [tower.lab.example.com]

PLAY RECAP *****
tower.lab.example.com : ok=1 changed=0 unreachable=0 failed=0 ...
```

6. Save, commit, and push your changes to the remote repository.

```
[student@workstation api-review]$ git status
On branch master
Your branch is up to date with 'origin/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: copy_template.yml
modified: inventory.yml
modified: template_cleanup.yml

no changes added to commit (use "git add" and/or "git commit -a")
[student@workstation api-review]$ git add --all
[student@workstation api-review]$ git commit -m "Updated review playbooks"
...output omitted...
[student@workstation api-review]$ git push
...output omitted...
```

## Evaluation

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

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

## Finish

From the **workstation** machine, run the **lab api-review finish** command to complete this lab.

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

This concludes the lab.

# Summary

---

In this chapter, you learned:

- Red Hat Ansible Tower provides a browsable REST API, which can easily be used to automate operations and to integrate Red Hat Ansible Tower with third-party products.
- 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**.
- Launching a Job Template from the API is done in two steps. First, access the template with the GET method to get information about any parameters or data that you might need to launch the job. Then, access the template with the POST method to actually launch the job.
- You can use an Ansible Playbook to launch a Job Template by using the **uri** module to access the Red Hat Ansible Tower API.



## Chapter 12

# 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 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. The advantage of importing the static inventory is that 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 approach 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 to 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

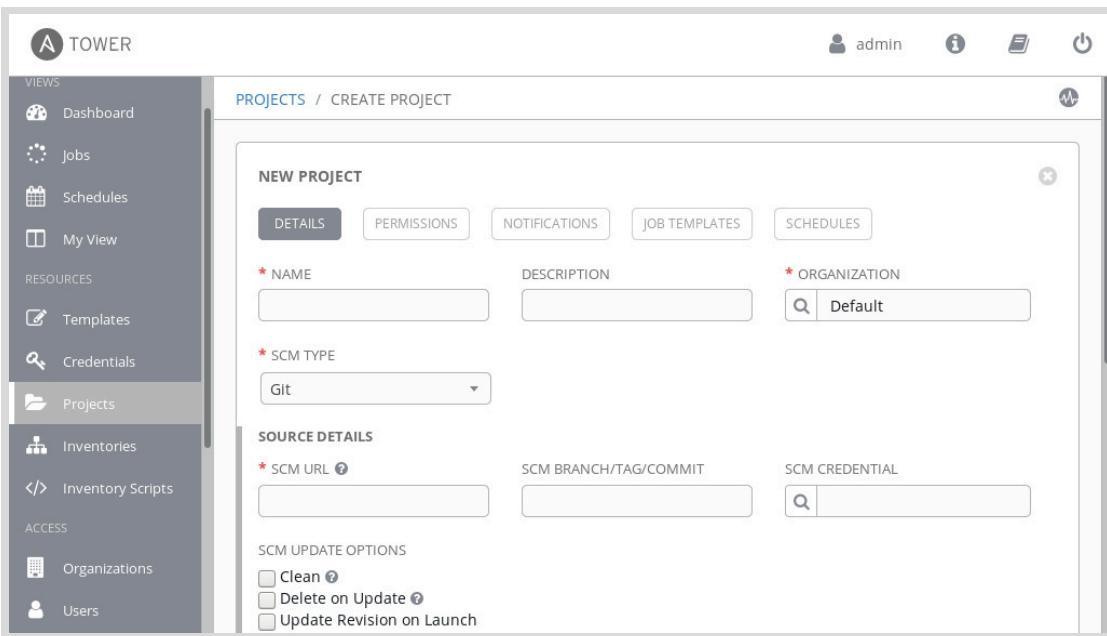
Remember, if the destination inventory is not empty, then by default **awx-manage inventory\_import** combines the existing information in that inventory in Ansible Tower with the data that you are uploading. Although this behavior can be convenient, be aware that it can also make a big mess if done 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 type of SCM supported by Projects in Ansible Tower.

To configure this functionality, start by setting up a Project that points to your Git repository. This is done in exactly the same way as when you are setting up a Project 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 may 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.

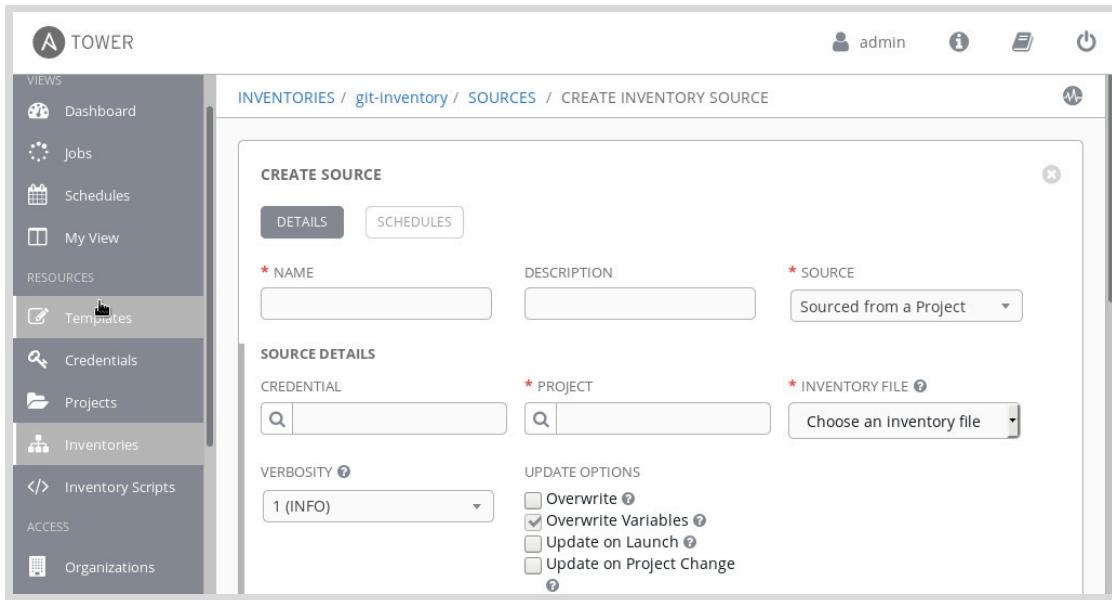


**Figure 12.1: Configuring Git-based projects**

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 12.2: 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

You should be able to:

- 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 advinventory-import start**, to verify that Ansible Tower services are running and all required resources are available.

```
[student@workstation ~]$ lab advinventory-import start
```

- 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"
```

```

2.570 INFO Updating inventory 5: Exercise
3.719 INFO Reading Ansible inventory source: /root/example-inventory
3.723 INFO Using VIRTUAL_ENV: /var/lib/awx/venv/ansible
3.723 INFO Using PATH: /var/lib/awx/venv/ansible/bin:/var/lib/...
3.723 INFO Using PYTHONPATH: /var/lib/awx/venv/ansible/lib/...
4.633 ERROR ansible-inventory 2.8.0
...output omitted...
4.635 ERROR Parsed /root/example-inventory inventory source with ini plugin
4.635 INFO Processing JSON output...
4.636 INFO Loaded 2 groups, 5 hosts
4.967 INFO Inventory import completed for (Exercise - 15) in 2.4s
[root@tower ~]# exit

```

Ignore the error messages as long as the last message indicates that the inventory import has completed.

14. Log in to the Ansible Tower web interface as the **admin** user with **redhat** as password. 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.
  15. 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 `http://git.lab.example.com:8081/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         | <b>MyProjectGit</b>                                            |
| DESCRIPTION  | Project Based on Git                                           |
| ORGANIZATION | <b>Default</b>                                                 |
| SCM TYPE     | <b>Git</b>                                                     |
| SCM URL      | <code>http://git.lab.example.com:8081/git/inventory.git</code> |

- 2.4. Check the **UPDATE REVISION ON LAUNCH** checkbox.
  - 2.5. Click **SAVE** to add the project.
- ▶ 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         | <b>ExerciseGit</b>        |
| DESCRIPTION  | Static Inventory from Git |
| ORGANIZATION | <b>Default</b>            |

- 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           | <b>git-inventory</b>          |
| DESCRIPTION    | Static Inventory from Git     |
| SOURCE         | <b>Sourced from a Project</b> |
| PROJECT        | <b>MyProjectGit</b>           |
| INVENTORY FILE | <b>git-inventory</b>          |



#### Note

Make sure you manually enter **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 `http://git.lab.example.com:8081/git/inventory.git` Git repository in the `/home/student/git-repos` directory, and change to the cloned project directory.

```
[student@workstation ~]$ cd /home/student/git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/inventory.git
[student@workstation git-repos]$ 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 add git-inventory
[student@workstation inventory]$ git commit -m "adding serverf"
[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.

## Finish

From **workstation**, run the **lab advinventory-import finish** script to clean up this exercise.

```
[student@workstation ~]$ lab advinventory-import finish
```

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.

## 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 example looks at the built-in support for OpenStack. The second example briefly examines the built-in support for getting information from a Red Hat Satellite 6 server. The third example investigates how custom dynamic inventory scripts can be used.

## OpenStack Dynamic Inventories

Cloud technologies like OpenStack bring many changes to the server life cycle. 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. The following is an example of how to create credentials for use by an OpenStack dynamic inventory:

The screenshot shows the Ansible Tower interface. The left sidebar has a dark theme with the following navigation items: VIEWS, Dashboard, Jobs, Schedules, My View, RESOURCES, Templates, Credentials (which is selected), Projects, Inventories, Inventory Scripts, ACCESS, and Organizations. The main content area has a light background and displays the 'CREDENTIALS / CREATE CREDENTIAL' page. At the top of this page is a 'NEW CREDENTIAL' section with two tabs: 'DETAILS' (selected) and 'PERMISSIONS'. Below these tabs are fields for 'NAME' (with a placeholder 'Ansible'), 'DESCRIPTION' (empty), and 'ORGANIZATION' (with a placeholder 'SELECT AN ORGANIZATION'). A search bar labeled 'CREDENTIAL TYPE' contains the value 'OpenStack'. Under the 'TYPE DETAILS' heading, there are three sets of fields: 'USERNAME' (empty), 'PASSWORD (API KEY)' (with a 'SHOW' button and empty field), and 'HOST (AUTHENTICATION URL)' (empty). Below these are fields for 'PROJECT (TENANT NAME)' (empty) and 'DOMAIN NAME' (empty).

**Figure 12.3: 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 must choose the appropriate credential type for the product you are using. In this example, the product 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/`

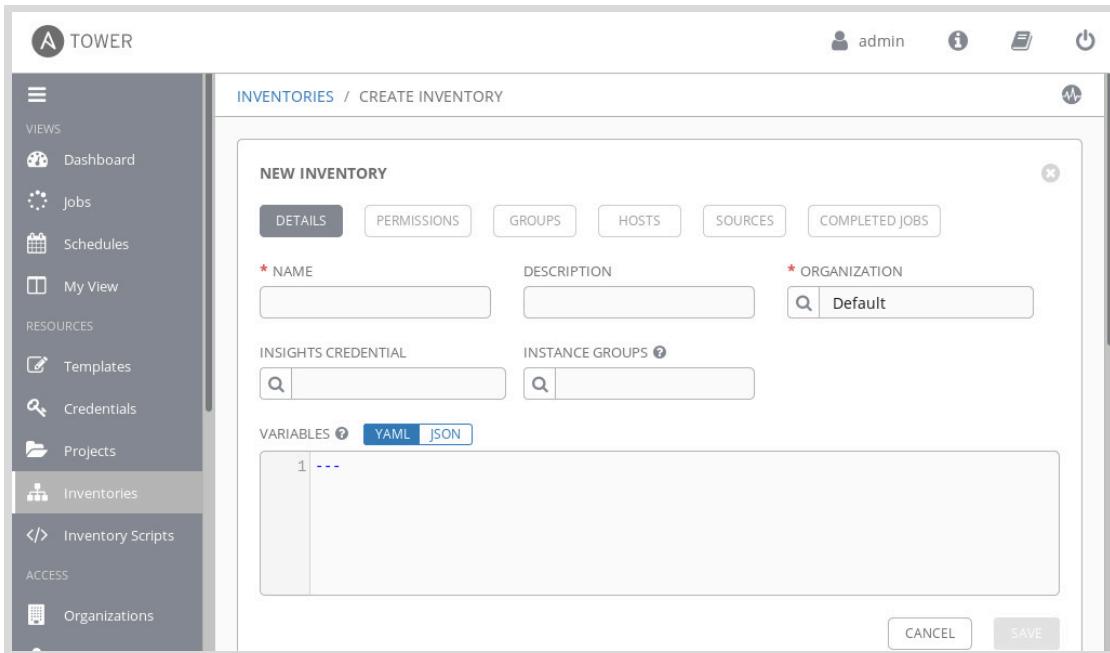
**Project**

The name of the project (tenant) that you want to use

**Domain Name**

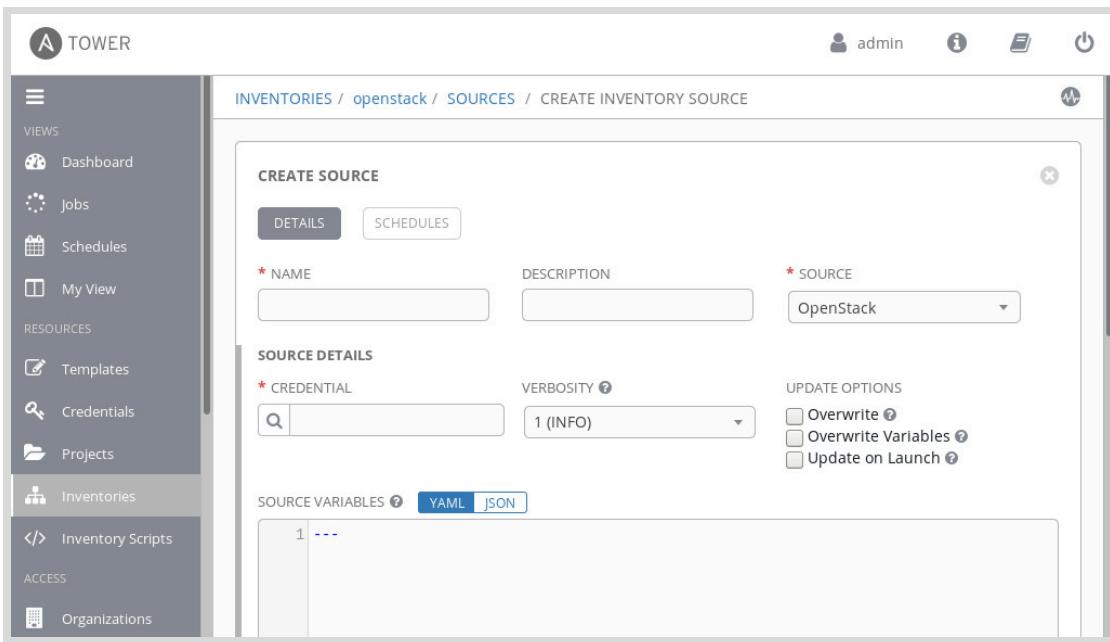
Needed only with Keystone v3, defines administrative boundaries

Those newly created credentials will 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 12.4: Cloud inventory creation**

That new inventory needs a unique **NAME** and you must 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 12.5: 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** from which 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, meaning 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, 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, 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 to display the contents of that group, giving you a list of the hosts associated with the group. This inventory is updated every time you synchronize it with the external source. Synchronization 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. After it is registered, the new server appears in the dynamic inventory generated from Red Hat Satellite information. 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/](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**. Three pieces of additional information are required:

#### 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 is 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).

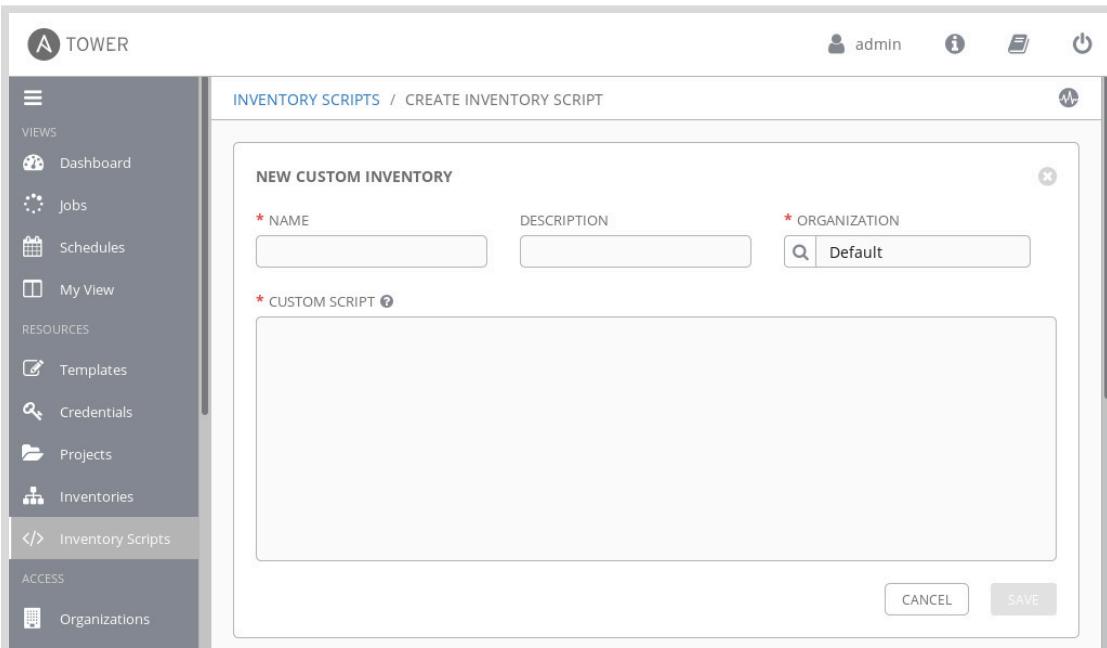
Optionally, if the `--list` option returns a top-level element called `_meta`, it is possible to return all host variables in one script call, which improves script performance. In that case, `--host` calls will not be made.

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.



**Figure 12.6: 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.

After importing and defining 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 existing inventory for the dynamic inventory. Set the **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.

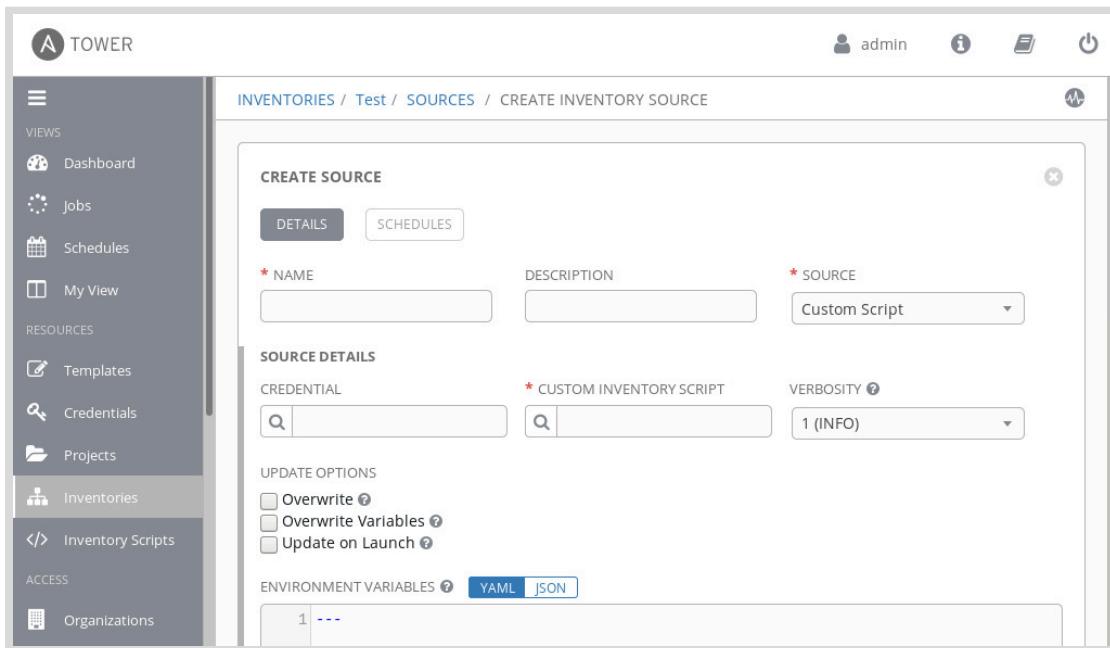


Figure 12.7: Adding new Inventory Group for custom dynamic inventory



## 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 advinventory-dynamic start**. This setup script prepares the IdM server for this exercise.

```
[student@workstation ~]$ lab advinventory-dynamic start
```

- ▶ 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         | <b>ldap-freeipa.py</b>           |
| DESCRIPTION  | Dynamic Inventory for IdM Server |
| ORGANIZATION | <b>Default</b>                   |

- 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         | <b>Dynamic Inventory</b>          |
| DESCRIPTION  | Dynamic Inventory from IdM server |
| ORGANIZATION | <b>Default</b>                    |

- 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                    | <b>Custom Script</b>                |
| DESCRIPTION             | Custom Script for Dynamic Inventory |
| SOURCE                  | <b>Custom Script</b>                |
| CUSTOM INVENTORY SCRIPT | <b>ldap-freeipa.py</b>              |

- 4.4. Under the **UPDATE OPTIONS** section, select 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 then wait until the cloud becomes green and static.
  - 5.2. Click **GROUPS**, and then 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 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 (this setting is 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
```

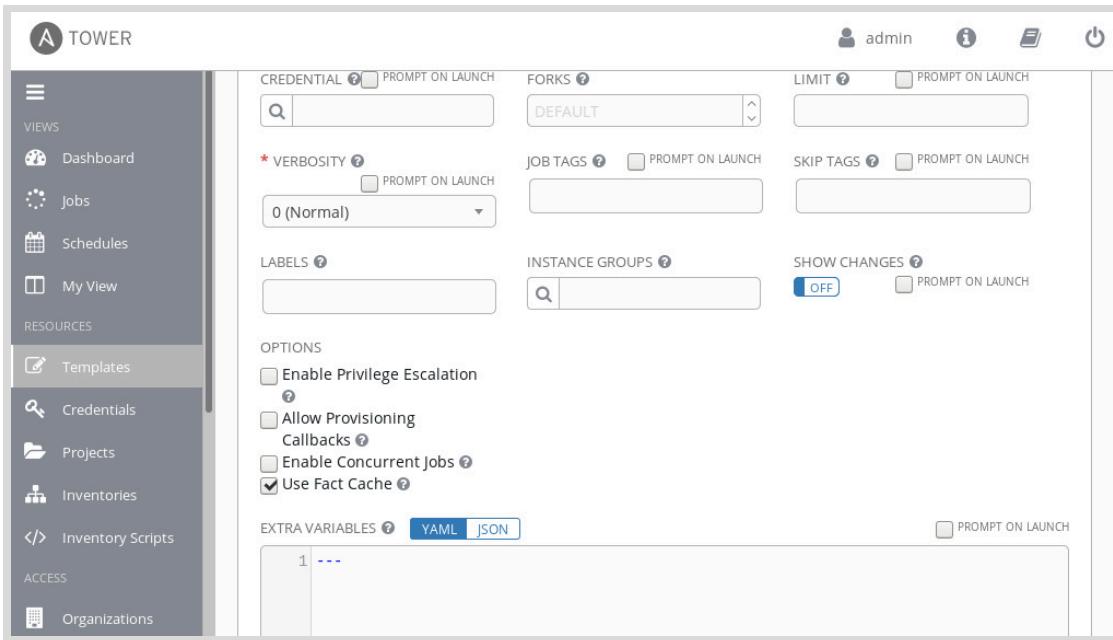


Figure 12.8: Enabling "Use Fact Cache" in a job template

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. On that page, you must specify a name for the smart inventory, assign it to an organization, and specify the smart host filter for the smart inventory.

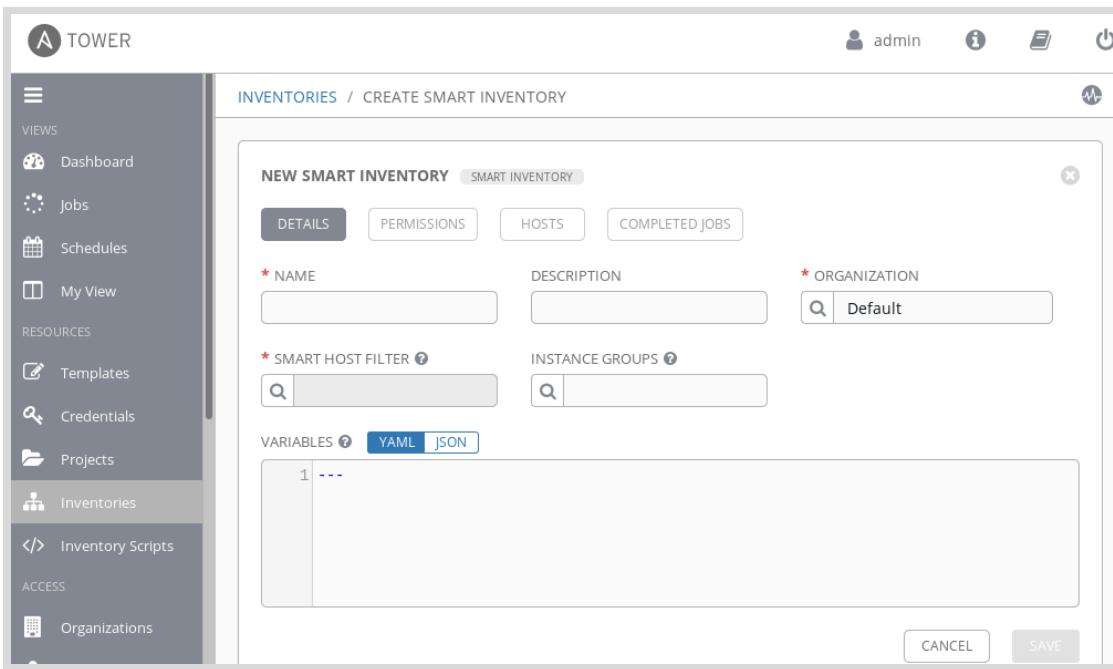


Figure 12.9: New Smart Inventory page

## 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 **DYNAMIC HOSTS** window opens. Enter your host filter or filters in the search field, and then 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**, followed by the name of the Ansible fact in the old format (fact injected as a variable name), a colon, and then the exact value that you want to match. 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

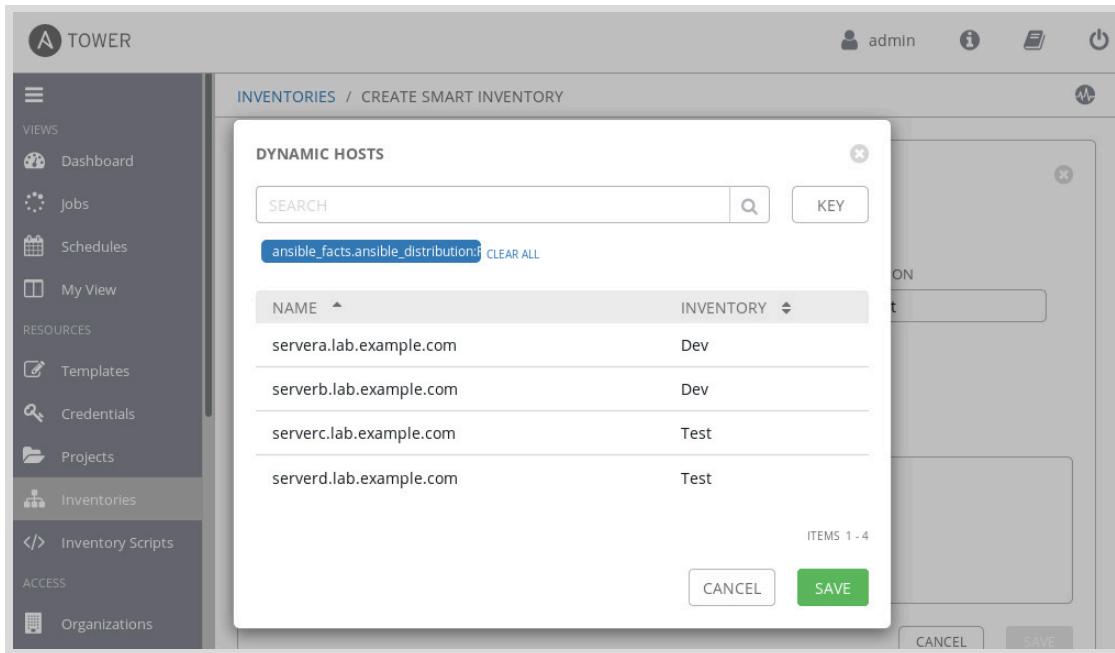
The current syntax for host filters is potentially confusing, because it appears to refer to a fact namespaced under **ansible\_facts**, but that is *not* the case. The **ansible\_facts** string in the host filter indicates that you want to match a fact, not a host name or something else.

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 currently the default setting in Ansible 2.8, but is subject to change.

Host filters for smart inventories do not yet support the modern naming for facts, 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 12.10: 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 advinventory-smart start**, which verifies that Ansible Tower services are running and all the required resources are available.

```
[student@workstation ~]$ lab advinventory-smart start
```

- ▶ 1. Log in to the Ansible Tower web interface running on the **tower** system using the **admin** account and the **redhat** password.
- ▶ 2. 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.
  - 2.1. Click **Inventories** in the left navigation bar.
  - 2.2. Click on the link for the **Dev** host group, and go to **HOSTS**.
  - 2.3. Click on the link for **servera.lab.example.com**, and go to **FACTS**.
  - 2.4. Verify that the facts for **servera.lab.example.com** are available.
  - 2.5. Repeat the same steps to verify that the facts for **serverb.lab.example.com** are also available in Ansible Tower's cache.
- ▶ 3. 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.
  - 3.1. Click **Inventories** in the left navigation bar.
  - 3.2. Click **+**, and select **Smart Inventory** to create a new smart inventory.
  - 3.3. On the next screen, fill in the details as follows:

| Field             | Value                                            |
|-------------------|--------------------------------------------------|
| NAME              | <b>Smart</b>                                     |
| DESCRIPTION       | Smart Inventory                                  |
| ORGANIZATION      | <b>Default</b>                                   |
| SMART HOST FILTER | <b>ansible_facts.ansible_distribution:RedHat</b> |

**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 then 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.

- 3.4. Click **SAVE** to create the smart inventory.
- 3.5. Click **HOSTS**, and then verify that both **servera.lab.example.com** and **serverb.lab.example.com** are available.
- ▶ 4. 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.
  - 4.1. Click **Templates** in the left navigation bar.
  - 4.2. Click on the link for the **TEST webservers setup** job template.
  - 4.3. Verify that you have selected the **Use Fact Cache** checkbox. Click **LAUNCH** to launch a job based on this job template. When the job runs, it retrieves facts for the hosts **serverc.lab.example.com** and **serverd.lab.example.com**.
  - 4.4. Wait until the job's status is **Successful**, and then click **Inventories** in the left navigation bar.
  - 4.5. Click on the link for the **Test** inventory, and click the **HOSTS** button.
  - 4.6. 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.
  - 4.7. 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**.
  - 4.8. Click **Inventories** in the left navigation bar, then click on the link for the **Smart** inventory.
  - 4.9. Go to **HOSTS**, and verify that both **serverc.lab.example.com** and **serverd.lab.example.com** are now available in this smart inventory. These hosts are now available because they meet the smart host filter condition.
- ▶ 5. 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

You should be able to:

- 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 advinventory-review start**. This setup script prepares the IdM server for this exercise.

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

1. From the CLI on **tower.lab.example.com**, import the existing static inventory, available in the **/root/lab-example-inventory** file, into the existing empty **Lab** Ansible Tower inventory. The lab setup script has created that **Lab** inventory for you.
2. Create a new project, named **LabProjectGit** that uses the Git repository located at <http://git@git.lab.example.com:8081/git/inventory.git>. Use the existing **student-git** SCM credentials. Make sure the revision is updated on project launch.
3. Create a new inventory named **LabGit** with the **git-inventory-lab** inventory file available through the **LabProjectGit** project.
4. Add the 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 the **Operations** machine credential, and then 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.

## Evaluation

As the **student** user on **workstation**, run the **lab advinventory-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 advinventory-review grade
```

This concludes the comprehensive review.

## ► 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

You should be able to:

- 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 advinventory-review start**. This setup script prepares the IdM server for this exercise.

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

- From the CLI on **tower.lab.example.com**, import the existing static inventory, available in the **/root/lab-example-inventory** file, into the existing empty **Lab** Ansible Tower inventory. The lab setup script has created that **Lab** inventory for you.

- 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 ~]#
```

- 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...
```

- 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"
 3.082 INFO Updating inventory 5: Lab
 4.972 INFO Reading Ansible inventory source: /root/lab-example-inventory
 4.976 INFO Using VIRTUAL_ENV: /var/lib/awx/venv/ansible
 4.977 INFO Using PATH: /var/lib/awx/venv/ansible/bin:/var/lib/awx/venv/
awx/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
 4.977 INFO Using PYTHONPATH: /var/lib/awx/venv/ansible/lib/python3.6/site-
packages:
 5.776 ERROR ansible-inventory 2.8.0
...output omitted...
 5.778 ERROR Parsed /root/lab-example-inventory inventory source with ini
plugin
 5.779 INFO Processing JSON output...
 5.779 INFO Loaded 1 groups, 2 hosts
 5.973 INFO Inventory import completed for (Lab - 11) in 3.0s
[root@tower ~]# exit
```

Ignore the error messages as long as the last message indicates that the inventory import has completed.

- 1.4. Log into the Ansible Tower web UI as **admin**. Use **redhat** as the 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 imported in a previous step.
- 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**, as well as **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 <http://git@git.lab.example.com:8081/git/inventory.git>. Use the existing **student-git** SCM credentials. Make sure the revision is updated on project launch.
  - 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           | <b>LabProjectGit</b>                                                                                                      |
| DESCRIPTION    | Project Based on Git                                                                                                      |
| ORGANIZATION   | <b>Default</b>                                                                                                            |
| SCM TYPE       | <b>Git</b>                                                                                                                |
| SCM URL        | <a href="http://git@git.lab.example.com:8081/git/inventory.git">http://git@git.lab.example.com:8081/git/inventory.git</a> |
| SCM CREDENTIAL | <b>student-git</b>                                                                                                        |

- 2.4. Select the **UPDATE REVISION ON LAUNCH** checkbox.
- 2.5. Click **SAVE** to add the project.
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 then select **Inventory** to add a new inventory.
  - 3.3. On the next screen, fill in the details as follows:

| Field        | Value                     |
|--------------|---------------------------|
| NAME         | <b>LabGit</b>             |
| DESCRIPTION  | Static Inventory from Git |
| ORGANIZATION | <b>Default</b>            |

- 3.4. Click **SAVE** to add the inventory.
- 3.5. Go to the **SOURCES** section, and then click **+** to add a new source.
- 3.6. On the next screen, fill in the details as follows:

| Field          | Value                         |
|----------------|-------------------------------|
| NAME           | <b>git-inventory-lab</b>      |
| DESCRIPTION    | Static Inventory from Git     |
| SOURCE         | <b>Sourced from a Project</b> |
| PROJECT        | <b>LabProjectGit</b>          |
| INVENTORY FILE | <b>git-inventory-lab</b>      |

**Note**

Make sure that you manually enter **git-inventory-lab** in the **INVENTORY FILE** combo box. The file may not display in the combo box 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 that the **servere.lab.example.com** and **serverf.lab.example.com** hosts are available under the **HOSTS** section.
4. Add the 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         | <b>ldap-idm.py</b>        |
| DESCRIPTION  | Dynamic Inventory for IdM |
| ORGANIZATION | <b>Default</b>            |

- 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         | <b>Lab Dynamic Inventory</b>      |
| DESCRIPTION  | Dynamic Inventory from IdM server |
| ORGANIZATION | <b>Default</b>                    |

- 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                    | <b>Custom Script</b>                    |
| DESCRIPTION             | Custom Script for Lab Dynamic Inventory |
| SOURCE                  | <b>Custom Script</b>                    |
| CUSTOM INVENTORY SCRIPT | <b>ldap-idm.py</b>                      |

- 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 then 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 the **Operations** machine credential, and then 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 then 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 then 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**, then 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 then 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 then select **Smart Inventory** to create a new smart inventory.
  - 8.3. On the next screen, fill in the details as follows:

| Field             | Value                                            |
|-------------------|--------------------------------------------------|
| NAME              | <b>LabSmart</b>                                  |
| DESCRIPTION       | Lab Smart Inventory                              |
| ORGANIZATION      | <b>Default</b>                                   |
| SMART HOST FILTER | <b>ansible_facts.ansible_distribution:RedHat</b> |



### 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, then 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.

## Evaluation

As the **student** user on **workstation**, run the **lab advinventory-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 advinventory-review grade
```

This concludes the comprehensive review.

# 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 that 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.



# Creating a Simple CI/CD Pipeline with Ansible Tower

### Goal

Build and operate a proof-of-concept CI/CD pipeline based on Ansible automation and integrating Red Hat Ansible Tower.

### Objectives

- Integrate Ansible Tower with a web-based Git repository system such as GitLab or GitHub to build a simple pipeline to automatically deploy playbooks when changes are committed.

### Sections

- Integrating CI/CD with Ansible Tower (and Guided Exercise)

# Creating a Simple CI/CD Pipeline with Ansible Tower

---

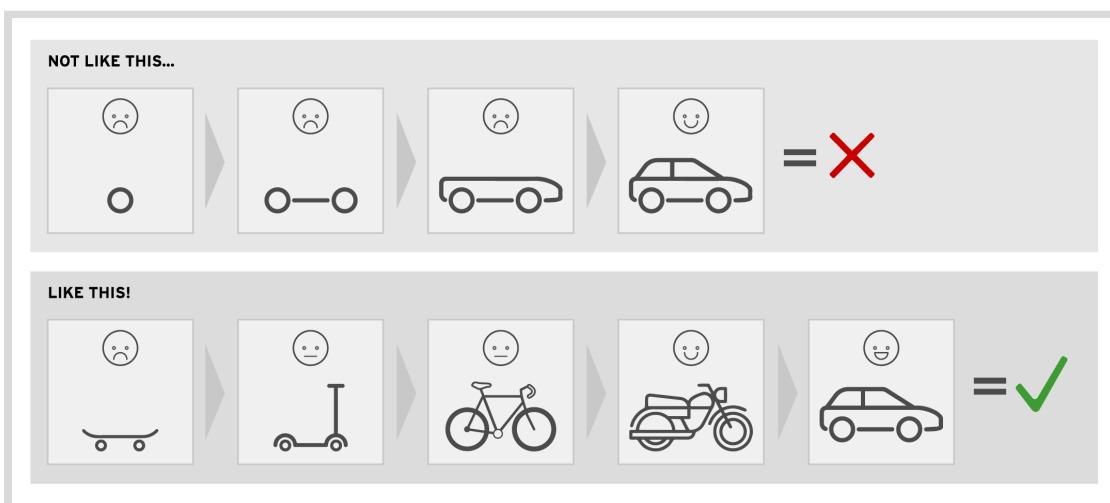
## Objectives

After completing this section, you should be able to create a CI/CD Pipeline in GitLab to call an Ansible Tower Job Template.

## Continuous Integration

Continuous Integration is the DevOps practice of using automation to integrate individual developer contributions into a shared code repository. Continuous Delivery focuses on executing processes more quickly and more frequently, by bringing and executing them together.

This practice encourages committing small changes more often, versus committing large changes infrequently.



**Figure 13.1: Release often to keep your customers happy.**

Automated routines are grouped together in jobs, which are placed in a pipeline. The pipeline defines the order in which the jobs are executed.

There are no rules for the contents of your pipeline, but the basic premise is that pipelines facilitate automated quality control. This automation eliminates the need for human intervention in a reliably repeatable way.

Some examples of what the jobs in your pipeline may do include:

- Syntax checking to ensure that code does not contain any syntax errors that prevents code compilation.
- "Linting" (running a program to analyze your code) to detect behaviors and practices that could potentially be improved.
- Calling the necessary steps to build code to produce a deployable artifact.
- Deploying and installing code to a server.

- "Smoke testing" in the form of cursory, limited tests to ensure that the most important functions work.
- Unit testing to verify that an individual unit of code (function) works as expected.
- Integration testing to verify that the individual units of code (functions) work together as expected.
- Regression testing to verify that new, or recently modified code does not adversely affect existing features (functions).

This practice has a number of benefits:

- Fast feedback about the operational status of the code once it is committed.
- End-to-end testing of code is automated when code is checked in, resulting in a faster feedback loop between developers working on the code.
- Facilitates integrating code from different developers.
- Greater visibility for developers collaborating on code.
- Developers and testers are able to fail faster, within the same sprint, further reducing the feedback loop.
- Reduced debugging sessions at the end of long sprints.

In order to get started with CI/CD pipelines, you need:

- Access to a distributed version control service
- Tests for your code
- Access to a CI/CD service to run your tests



### Note

You can integrate Red Hat Ansible Tower with a variety of CI/CD platforms to create a pipeline for Ansible Playbook projects. In this course, we use a classroom instance of GitLab that can provide both repository management and CI/CD platform services. Other alternatives are possible and might have different advantages, but this configuration provides a good example.

## Building CI/CD Pipelines with GitLab

In the CI/CD pipeline, GitLab uses *runners* to run jobs inside of the pipeline. Runners provide the environment in which your pipeline is executed. A runner can be specific to a project or can be shared across all projects.

Runners *executors* to execute commands inside of your jobs.

Executors can be:

- Containers
- Virtual Machines
- Local shells (jobs execute in a shell on the GitLab machine itself)

- SSH to connect to a different machine to execute jobs

Ideally a runner executes the code defined in the pipeline independently of any connected systems.



Figure 13.2: Pipeline with jobs

## CI/CD and Red Hat Ansible Tower

Use a CI/CD pipeline, integrated with Red Hat Ansible Tower, to perform automated routines with each commit to an Ansible Playbook project. A typical pipeline performs the following steps:

1. Pulls the latest version of the playbook from the **dev** branch.
2. Performs syntax checking to ensure sanity of the code, and linting to ensure that it adheres to a set of best practices rules.
3. Synchronizes the playbook from the **dev** branch to a Project in Ansible Tower.
4. Executes the Job Template against the Dev Inventory in Ansible Tower, using the code in the **dev** branch.
5. Verifies the functionality of a critical component in the form of a unit test for the hosts in the Dev Inventory.
6. Merges the **dev** branch to the **master** branch.
7. Synchronizes the playbook from the **master** branch to a Project in Ansible Tower.
8. Executes the Job Template against the Prod Inventory in Ansible Tower, using code from the **master** branch.
9. Verifies the functionality of a critical component in the form of a unit test for hosts in the Prod Inventory.
10. Sends a notification to inform the developers of the status of the job.

This pipeline automates processes that would otherwise have to be performed manually. Automation allows developers to focus on creating and editing playbooks using their tool of choice. Developers can simply commit their work, knowing that automation with a CI pipeline is integrated with Red Hat Ansible Tower.

## Ansible Lint

Ansible Lint is a command-line tool to detect errors, bugs, suspicious code constructs, and stylistic errors that could potentially be improved.

It is used by the Ansible Galaxy project to lint and calculate quality scores for content contributed to the Galaxy Hub.

**Important**

The **ansible-lint** command is not currently shipped with Red Hat Ansible Automation or officially supported by Red Hat, but is developed by the Ansible community upstream.

This command is included in EPEL for Red Hat Enterprise Linux 7 and current releases of Fedora. Instructions for installation on other operating systems is available at <https://docs.ansible.com/ansible-lint/>.

Ansible Lint uses rules from  `${PYTHON_PATH}/site-packages/ansiblelint/rules/` in the form of Python modules which may be used as is, or edited to suit your own needs. Additional user-defined rules may be created and used with the default rules, as follows:

Consider the following example of a badly written playbook.

```
[student@demo examples]$ cat playbook.yml

- name: A bad play
 hosts: demo.lab.example.com

 tasks:
 - command: systemctl mask iptables.service
 ...
```

Syntactically, the playbook is free of any errors, and the **ansible-playbook playbook.yml --syntax-check** command will state the same.

```
[student@demo examples]$ ansible-playbook playbook.yml --syntax-check

playbook: playbook.yml
```

**ansible-lint** reveals that this is not a well written playbook.

```
[student@demo examples]$ ansible-lint playbook.yml
[301] Commands should not change things if nothing needs doing ①
playbook.yml:6 ②
Task/Handler: command systemctl mask iptables.service ③

[303] systemctl used in place of systemd module
playbook.yml:6
Task/Handler: command systemctl mask iptables.service

[502] All tasks should be named
playbook.yml:6
Task/Handler: command systemctl mask iptables.service
```

- ① The numbers in square brackets tell us which linting rules (tags) were matched, along with a short description of the issue.
- ② The filename and the line numbers where the issues were found.
- ③ The names of the problematic tasks.

Rules in **ansible-lint** are implemented as Python modules, and are stored in  `${PYTHON_PATH}/site-packages/ansiblelint/rules/`.

Remediating these issues is simplified as a result of **ansible-lint**.

```
[student@demo examples]$ cat playbook.yml

- name: A better play
 hosts: demo.lab.example.com

 tasks:
 - name: Mask iptables.service
 systemd:
 name: iptables.service
 masked: true
...
[student@demo examples]$ ansible-lint playbook.yml
[student@demo examples]$ echo $?
0
```



#### Note

The **ansible-lint** command is more capable of verifying the sanity of a playbook when compared to the **ansible-playbook --syntax-check** command. Thus, there is no need to use **ansible-playbook --syntax-check** when using the **ansible-lint** command.

Many organizations implement a Style Guide, which defines the standards for writing playbooks. For example, the way that Booleans are declared can vary: True vs true vs Yes vs yes vs 1. While all these values mean the same thing, consistent use of styles assists in the readability of a playbook, which further aids in troubleshooting playbooks with errors.

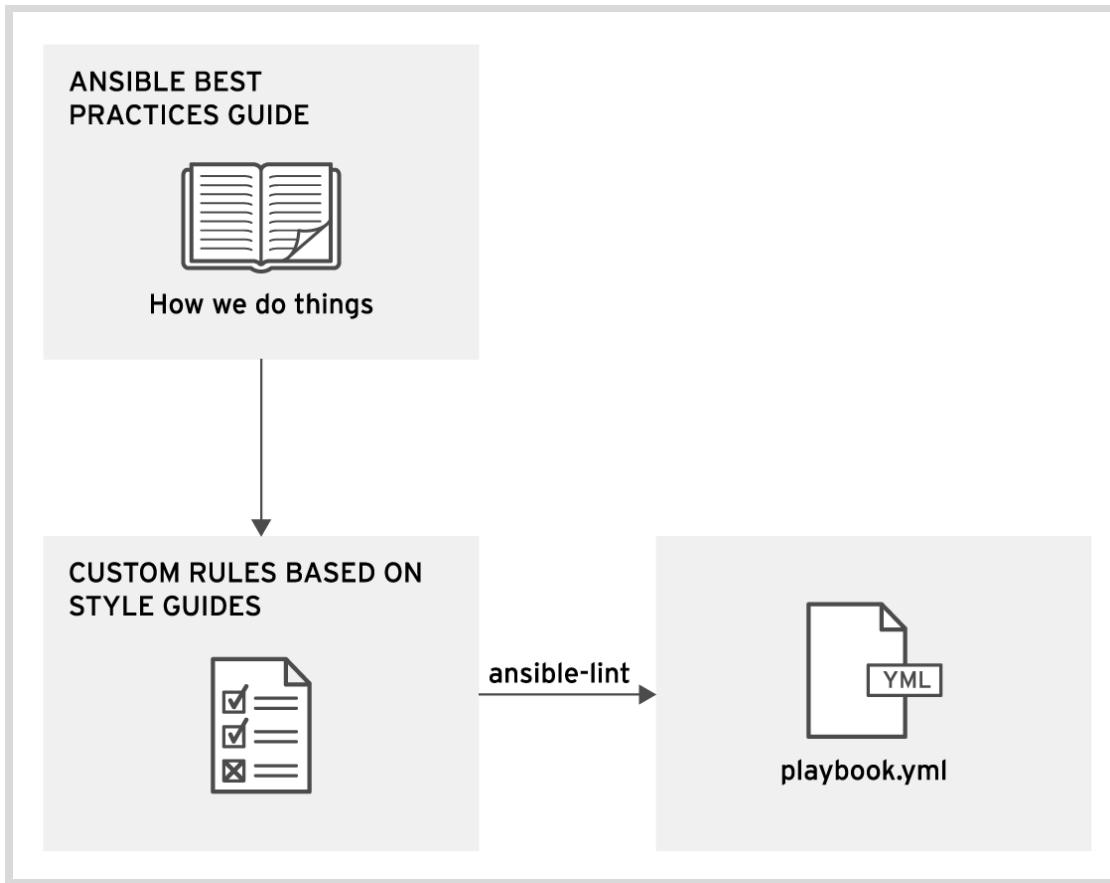


Figure 13.3: Ansible Lint

The **ansible-lint** command implements rules as Python modules, and your own rules may be used in addition to the default ones.

```
[student@demo ~]$ ansible-lint -R ~/ansible-lint/custom_rules/ playbook.yml
```

To override the default rules in  `${PYTHON_PATH}/site-packages/ansiblelint/rules/`, and instead use only your custom rules, use the **ansible-lint** command as follows:

```
[student@demo ~]$ ansible-lint -r ~/ansible-lint/custom_rules/ playbook.yml
```

```
Copyright (c) 2018, Ansible Project

from ansiblelint import AnsibleLintRule
import re

class VariableHasSpacesRule(AnsibleLintRule):
 id = '206'
 shortdesc = 'Variables should have spaces before and after: {{ var_name }}'
 description = 'Variables should have spaces before and after: `{{ var_name }}`'
 severity = 'LOW'
 tags = ['formatting']
 version_added = 'v4.0.0'

 bracket_regex = re.compile(r"\{\{[^{} -]|\[^{}]-\}\}"))

 def match(self, file, line):
 return self.bracket_regex.search(line)
```

**Figure 13.4: Example Ansible Lint rule checking for spaces before and after variable names.**



## References

### Continuous Integration

[https://en.wikipedia.org/wiki/Continuous\\_integration](https://en.wikipedia.org/wiki/Continuous_integration)

### GitLab Continuous Integration & Delivery

<https://about.gitlab.com/product/continuous-integration/>

### Ansible Continuous Delivery

<https://www.ansible.com/use-cases/continuous-delivery>

### Ansible Lint

<https://docs.ansible.com/ansible-lint/>

### lint (software)

[https://en.wikipedia.org/wiki/Lint\\_\(software\)](https://en.wikipedia.org/wiki/Lint_(software))

## ► Guided Exercise

# Integrating a GitLab CI/CD Pipeline with Ansible Tower

In this exercise, you will build and operate a proof-of-concept CI/CD pipeline in GitLab CE, based on Ansible automation and integrating Red Hat Ansible Tower.

## Outcomes

You should be able to trigger a job in Ansible Tower via a CI/CD pipeline in GitLab.

## Before You Begin

Open a terminal on the **workstation** machine as the **student** user, and run the following command:

```
[student@workstation ~]$ lab cicd-tower start
```

- ▶ 1. Clone the `http://git.lab.example.com:8081/git/cicd-tower.git` repository, which contains the Ansible Playbook project you use in this exercise.
  - 1.1. Change to the `/home/student/git-repos` directory. Create the directory if it does not exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos
[student@workstation ~]$ cd /home/student/git-repos
```

- 1.2. Clone the repository.

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/cicd-tower.git
```

- ▶ 2. Review the files in the **master** and **dev** branches of the repository.

- 2.1. Change to the `cicd-tower` directory.

```
[student@workstation git-repos]$ cd cicd-tower
```

- 2.2. Display the current branch.

```
[student@workstation cicd-tower]$ git branch
* master
```

- 2.3. List the files in the master branch.

```
[student@workstation cicd-tower]$ ls -l
total 8
-rw-rw-r-- 1 student student 609 May 20 13:50 playbook.yml
-rw-rw-r-- 1 student student 19 May 20 13:50 README.md
```

2.4. List all the branches.

```
[student@workstation cicd-tower]$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/dev
remotes/origin/master
```

2.5. Checkout the **dev** branch.

```
[student@workstation cicd-tower]$ git checkout dev
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
Switched to a new branch 'dev'
```

2.6. Display the current branch.

```
[student@workstation cicd-tower]$ git branch
* dev
 master
```

2.7. List the files in the **dev** branch.

```
[student@workstation cicd-tower]$ ls -la
total 12
drwxrwxr-x. 3 student student 77 May 20 20:56 .
drwxrwxr-x. 3 student student 24 May 20 20:55 ..
drwxrwxr-x. 8 student student 163 May 20 20:56 .git
-rw-rw-r--. 1 student student 1167 May 20 20:56 .gitlab-ci.yml
-rw-rw-r--. 1 student student 869 May 20 20:56 playbook.yml
-rw-rw-r--. 1 student student 19 May 20 20:55 README.md
```

- 3. Use the **ansible-lint** command to identify any issues with the **playbook.yml** file in the Git repository. You use this playbook to deploy both test and production web servers.

3.1. The **ansible-lint** command performs syntax checks in the same way as the **ansible-playbook --syntax-check** command.

```
[student@workstation cicd-tower]$ ansible-lint playbook.yml
Couldn't parse task at playbook.yml:21 (no action detected in task. This often
indicates a misspelled module name, or incorrect module path.

The error appears to be in <unicode string>: line 21, column 7, but may
be elsewhere in the file depending on the exact syntax problem.

(could not open file to display line)
{ 'firewall': { '_file_': 'playbook.yml',
```

```
'__line__: 22,
'immediate': True,
'permanent': True,
'service': 'http',
'state': 'enabled'},
'name': 'Open the port in the firewall'}
```

The output indicates there is an error with the **Open the port in the firewall** task in **playbook.yml** playbook.

- 3.2. There is no module called **firewall**, the correct module name is **firewalld**. Edit the third task in the **playbook.yml** file and correct the module name.

```
[student@workstation cicd-tower]$ cat playbook.yml
...output omitted...
- name: Open the port in the firewall
 firewalld:
 service: http
 state: enabled
 immediate: true
 permanent: true
...output omitted...
```

- 3.3. The **ansible-lint** command does more than syntax checking. It validates that your playbook is developed according to best practices.

Check the **playbook.yml** file again using the **ansible-lint** command:

```
[student@workstation cicd-tower]$ ansible-lint playbook.yml
[206] Variables should have spaces before and after: {{ var_name }}
playbook.yml:12
name: "{{packages}}"

[502] All tasks should be named
playbook.yml:28
Task/Handler: copy content=Hello world
dest=/var/www/html/index.html __line__=28 __file__=playbook.yml
```

- 4. The **ansible-lint** command identifies two issues, one on line 12 and one on line 28. The numeric code in brackets represents a linting category, followed by a description of the issue. To see all linting rules use the **ansible-lint -L** command.

Using your editor, make the following changes:

- 4.1. On line 12, introduce a space before and after the **packages** variable.
- 4.2. On line 28, provide the task a name of **Install sample web content**.

When you finish correcting these two issues, the content of the **playbook.yml** file is the following:

```
[student@workstation cicd-tower]$ cat playbook.yml

- name: Implement a basic webserver
 hosts: webservers
```

```

vars:
 packages:
 - httpd
 - httpd-devel

tasks:
 - name: Install the webserver software
 yum:
 name: "{{ packages }}"
 state: installed

 - name: Start and persistently enable httpd
 service:
 name: httpd
 enabled: yes
 state: started

 - name: Open the port in the firewall
 firewalld:
 service: http
 state: enabled
 immediate: true
 permanent: true

 - name: Install sample web content
 copy:
 content: "Hello world\n"
 dest: /var/www/html/index.html

 - name: Smoke Test
 uri:
 url: "http://{{ inventory_hostname }}"
 return_content: yes
 status_code: 200
 register: response
 delegate_to: localhost
 become: no
 failed_when: '"Hello World" not in response.content'

...

```

- 5. Verify that the **ansible-lint** command does not identify any other issues. The **ansible-lint** command does not return any output when there are no issues or errors with a playbook.

You can also examine the return code of the **ansible-lint** command to verify that playbook passes all linting tests.

```

[student@workstation cicd-tower]$ ansible-lint playbook.yml
...output omitted...
[student@workstation cicd-tower]$ echo $?
0

```

- 6. If a GitLab project contains a `.gitlab-ci.yml` file, GitLab creates a CI/CD pipeline for the project.

This file defines a sequence of stages, and each stage executes one or more jobs. Stages control the order of job execution. If a particular job fails, subsequent stages and jobs do not execute.

GitLab pipeline jobs execute on an available GitLab Runner instance. You can configure a GitLab Runner to execute your CI/CD pipelines on any machine. Configuring GitLab Runners is beyond the scope of the course.

In the classroom environment, a GitLab Runner instance is configured on the GitLab server. Use this runner to execute simple bash commands to implement CI/CD pipelines.

Review the `.gitlab-ci.yml` file that defines the CI/CD pipeline for this project:

```
[student@workstation cicd-tower]$ cat .gitlab-ci.yml
variables:
 LAUNCH_TOWER_JOB: tower-cli job launch --monitor --insecure
 TOWER_CREDENTIALS: -u admin -p redhat -h tower.lab.example.com
 GIT_REPO: http://git:redhat321@git.lab.example.com:8081/git/cicd-tower.git

stages:
 - lint
 - deploy
 - auto_merge

First stage; all branches
syntax check and linting:
stage: lint
script:
 - if ls *.yml; then true; else echo "No playbooks found!"; exit 1; fi
 - ansible-lint *.yml

#Second stage; only dev branch
launch test job:
stage: deploy
script:
 - tower-cli config verify_ssl false
 - $LAUNCH_TOWER_JOB $TOWER_CREDENTIALS -J "Deploy Test WebServers"
 - echo $?
only:
 - dev

#Second stage; only the master branch
launch prod job:
stage: deploy
script:
 - tower-cli config verify_ssl false
 - $LAUNCH_TOWER_JOB $TOWER_CREDENTIALS -J "Deploy Prod WebServers"
 - echo $?
only:
 - master

#Third stage; only applies to the dev branch
push to master:
stage: auto_merge
```

```

script:
 - git remote set-url origin $GIT_REPO
 - git checkout dev && git pull
 - git checkout master && git pull
 - git merge --no-ff dev
 - git push origin master
only:
 - dev

```

This pipeline configuration file defines a pipeline with three stages: **lint**, **deploy**, and **auto\_merge**.

The **lint** stage contains a single job, with a name of **syntax check and linting**. This job executes the **ansible-lint** command against every YAML file. If no YAML files exist in the repository, then the job fails.

The **deploy** stage triggers the execution of an Ansible Tower job template. For the **dev** branch, the Deploy Test Web Servers job template executes, while the Deploy Prod Web Servers template executes for the **master** branch.

The **auto\_merge** stage only defines a job for the **dev** branch. This job merges the changes in the **dev** branch into the master branch, and pushes to the remote repository.

► 7. Explore the Test resources in Ansible Tower.

7.1. From a new window or tab in Firefox, login to Ansible Tower at <https://tower.lab.example.com> as the **admin** user with the password **redhat**.

7.2. On the left, select **Templates** then select **Deploy Test WebServers**.

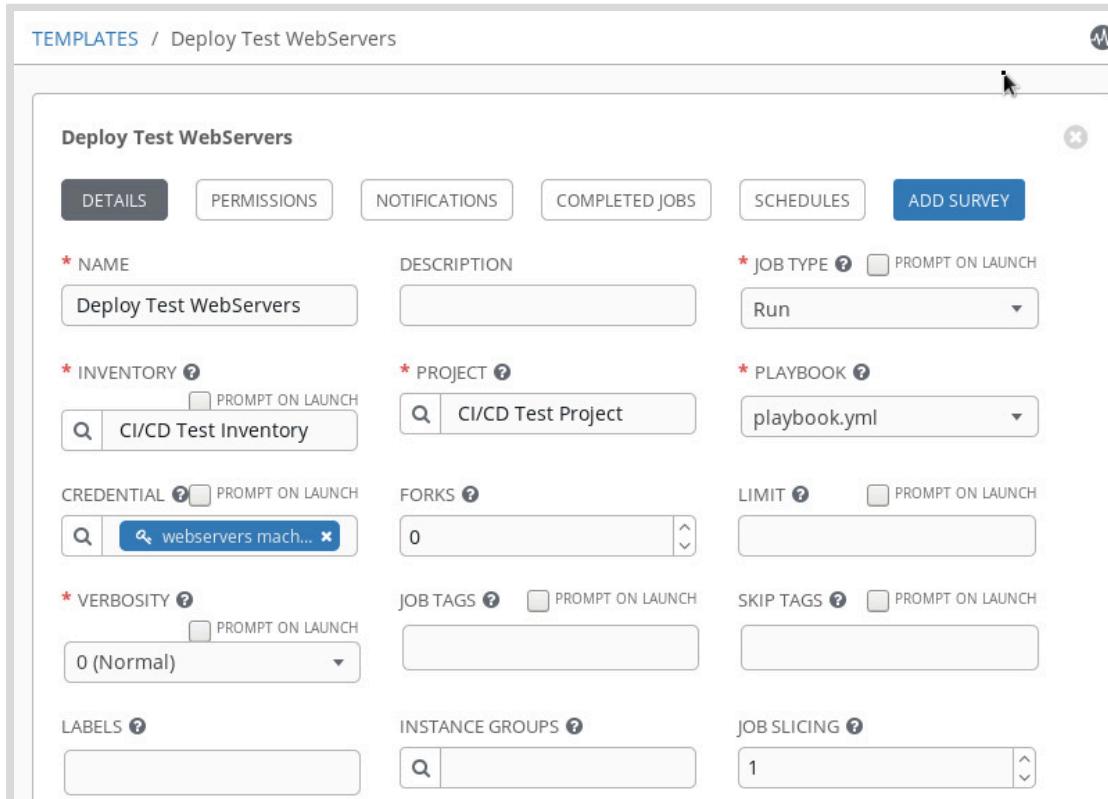


Figure 13.5: Test Web Servers Job Template

The **launch test job** stage in the GitLab pipeline calls the **Deploy Test WebServers** job template in Ansible Tower, which uses the following resources:

- An Inventory called **CI/CD Test Inventory**.
- A Project called **CI/CD Test Project** that uses a playbook called **playbook.yml**.
- A Credential called **webservers machine credential** that is used to execute tasks on the inventory hosts.

- 7.3. On the left, select **Inventories**, then select **CI/CD Test Inventory**, and then select the **HOSTS** tab.

The screenshot shows the 'Inventories / CI/CD Test Inventory / HOSTS' page. At the top, there are tabs: DETAILS, PERMISSIONS, GROUPS, HOSTS (which is selected), SOURCES, and COMPLETED JOBS. Below the tabs is a search bar and a 'RUN COMMANDS' button. The main area displays a table with two rows. Each row represents a host: 'serverc.lab.example.com' and 'serverd.lab.example.com'. Both hosts are listed under the 'HOSTS' column and are assigned to the 'webservers' group in the 'RELATED GROUPS' column. There are edit and delete icons for each host entry. At the bottom right of the table, it says 'ITEMS 1 - 2'.

**Figure 13.6: CI/CD Test Inventory Hosts**

When an Ansible Tower job template uses the **CI/CD Test Inventory**, it can execute jobs against the hosts **serverc.lab.example.com** and **serverd.lab.example.com**, which are grouped in the **webservers** group.

- 7.4. On the left, select **Projects** and then select **CI/CD Test Project**.

The screenshot shows the 'PROJECTS / CI/CD Test Project' page. At the top, there are tabs: DETAILS (selected), PERMISSIONS, NOTIFICATIONS, JOB TEMPLATES, and SCHEDULES. Below the tabs, there are fields for NAME ('CI/CD Test Project'), DESCRIPTION, and ORGANIZATION ('Default'). Under 'SCM TYPE', 'Git' is selected. In 'SOURCE DETAILS', the 'SCM URL' is set to 'http://git.lab.example.com:8081/git/cicd-tower.git', 'SCM BRANCH/TAG/COMMIT' is 'dev', and 'SCM CREDENTIAL' is 'gitlab credential'. Under 'SCM UPDATE OPTIONS', 'UPDATE REVISION ON LAUNCH' is checked. There are also fields for 'CACHE TIMEOUT (SECONDS)' (set to 0) and 'SCM UPDATE OPTIONS' checkboxes for CLEAN, DELETE ON UPDATE, and UPDATA REVISION ON LAUNCH.

**Figure 13.7: CI/CD Test Project**

The **CI/CD Test Project** uses a credential called **gitlab credential** to pull resources from the **dev** branch of the git repository located at **http://git.lab.example.com:8081/git/cicd-tower.git**.

- For use by Sgt Dheeraj sgt.dheeraj.dhirajsharma693@gmail.com Copyright © 2022 Red Hat, Inc.
- 7.5. On the left, select **Credentials**, and then explore the credentials used. In your environment, the **gitlab credential** is used to pull files from the GitLab Project called **cicd-tower**. Similarly, the **webservers machine credential** is used to login to your inventory hosts.
  - 7.6. Similar resources exist for the **Prod** objects. The playbook is first tested against our test inventory, and if the smoke test passes, the playbook is merged from the **dev** branch to the **master** branch. This branch is used by the **CI/CD Prod Project** to execute **playbook.yml** against hosts in the **CI/CD Prod Inventory**, which are **servera.lab.example.com** and **serverb.lab.example.com**. This process protects our production servers from being negatively impacted by errors found in playbooks, by performing tests in your pipeline.
- ▶ 8. Commit your local changes to the **playbook.yml** file to the **dev** branch. Push the committed changes to the remote repository. When you push changes to GitLab, you trigger an execution of the CI/CD pipeline that is defined in the **.gitlab-ci.yml** file.

```
[student@workstation cicd-tower]$ git add .
[student@workstation cicd-tower]$ git commit -m "Implemented CI pipeline"
[student@workstation cicd-tower]$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 353 bytes | 353.00 KiB/s, done.
...output omitted...
```

- ▶ 9. Review the recent pipeline execution that was triggered because of changes you pushed to the **dev** branch in the previous step. The pipeline fails in the second stage. Identify the error that causes the pipeline to fail.
- 9.1. From a new Window or Tab in Firefox, login to your GitLab instance at <http://git.lab.example.com:8081>, as the **git** user with the password **redhat321**.
  - 9.2. Select the **cicd-tower** project.
  - 9.3. From the menu on the left, click **CI/CD** (or the rocket icon, if the menu is collapsed) and then click **Pipelines**. The most recent pipeline execution appears at the top of the resulting page:



Figure 13.8: GitLab CI/CD pipeline status

Note that the pipeline has three stages called in order: **lint**, **deploy**, then **auto\_merge**.

To see what happens at each stage, click on the icon for that stage. At stage 1, you lint the playbook and check syntax. At stage 2, you trigger the **Deploy Test Webservers** job template. At stage 3, you merge the code from the dev branch into the master branch, and then push the merged changes to the remote repository. This triggers another CI/CD pipeline for the master branch.

- 9.4. When you see that your pipeline job completes, the second stage has a status of **failed**.

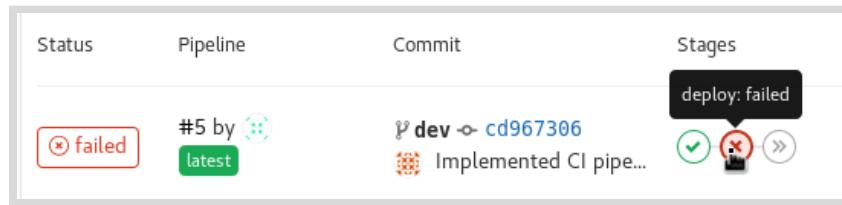


Figure 13.9: GitLab CI/CD pipeline failure

As a result, the third stage does not execute and code is not automatically merged into the **master** branch from the **dev** branch. Click the icon of the failed stage to see the name of the failed job. Click **launch\_test\_job** to open a new Firefox tab that displays output from the job.

The screenshot shows the GitLab Job summary page for a failed test job. The job log on the left shows a smoke test failing with the message: "changed: [serverc.lab.example.com] TASK [Smoke Test] \*\*\*\* fatal: [serverc.lab.example.com]: FAILED! => {"accept\_ranges": "bytes", "changed": false, "connection": "close", "content": "Hello world\\n", "content\_length": "12", "content\_type": "text/html; charset=UTF-8", "cookies": {}, "cookies\_string": "", "date": "Mon, 20 May 2019 18:19:10 GMT", "elapsed": 0, "etag": "\\\"c-58955c4b6abdd\\\"", "failed\_when\_result": true, "last\_modified": "Mon, 20 May 2019 18:19:09 GMT", "msg": "OK (12 bytes)", "redirected": false, "server": "Apache/2.4.37 (Red Hat Enterprise Linux)", "status": 200, "url": "http://serverc.lab.example.com"}". The right panel provides details about the job: Duration: 46 seconds, Timeout: 1h (from project), Runner: shell-runner (#3), Commit: 60cc2eb6, and Updates ansible-lint recommendations. A 'New issue' button is also present.

Figure 13.10: GitLab Job summary page

The test environment web servers fail the smoke test. According to the smoke test, the content on the test server is not correctly capitalized. The message on the web server should be **Hello World**, not **Hello world**. This means that the Ansible Tower Job Template called **Deploy Prod WebServers** was never called from the pipeline.

```
[student@workstation ~]$ curl http://servera
This is a test message RedHat 8.0

Current Host: servera

Server list:

servera.lab.example.com

serverb.lab.example.com

[student@workstation ~]$ curl http://serverb
This is a test message RedHat 8.0

Current Host: serverb

Server list:

servera.lab.example.com

serverb.lab.example.com

[student@workstation ~]$ curl http://serverc
Hello world
[student@workstation ~]$ curl http://serverd
Hello world
```

- 10. Return to the web browser session where you are logged in to **Ansible Tower**. From the menu on the left, select **Jobs** to verify that the **Deploy Test WebServers** job template executed with an error, and that the **Deploy Prod WebServers** job template did not execute.

The screenshot shows the Ansible Tower 'JOBS' page with a search bar and filter options ('Compact', 'Expanded', 'Finish Time (Descending)'). Three job entries are listed:

- 7 - Deploy Test WebServers** (Playbook Run) - Status: Failed (red circle)
- 8 - CI/CD Test Project** (SCM Update) - Status: Failed (red circle)
- 6 - CI/CD Test Project** (SCM Update) - Status: Failed (red circle)

Figure 13.11: Ansible Tower Jobs after the pipeline failed

- 11. Correct the content that the **playbook.yml** playbook deploys to each web server. Commit and push the changes. Verify that your code changes to the **dev** branch are integrated into the **master** branch, and that production servers are updated with these changes.
- 11.1. From your terminal window, return to the **/home/student/git-repos/cicd-tower** directory. Correct the fourth task in the **playbook.yml** playbook with the correctly capitalized content. When you finish, the content of the fourth task is:

```
- name: Install sample web content
 copy:
 content: "Hello World\n"
 dest: /var/www/html/index.html
```

- 11.2. Commit your changes to the **dev** branch, and push to the remote repository:

```
[student@workstation cicd-tower]$ git branch
* dev
 master
[student@workstation cicd-tower]$ git add .
[student@workstation cicd-tower]$ git commit \
> -m "Fixes content error."
[student@workstation cicd-tower]$ git push
```

- 12. Verify that the recent push to the repository triggers two pipeline executions, resulting in an update to production servers.
- 12.1. Go back to the web browser where you have a GitLab session, and return to the **Pipelines** summary page for the **cicd-tower** project.

| Status | Pipeline      | Commit                                   | Stages |
|--------|---------------|------------------------------------------|--------|
| passed | #7 by  latest | master -> ee2b7d1d<br>Merge branch 'dev' |        |
| passed | #6 by  latest | dev -> 215350b1<br>Fixes content error.  |        |

**Figure 13.12: GitLab CI/CD pipeline passes all stages**

A new pipeline execution for the **dev** branch is visible. After the pipeline finishes, it has a status of **passed**.

Because the 3rd stage of this pipeline pushes a new merge commit to the **master** branch, a new pipeline execution starts.

The pipeline for the **master** branch only defines two stages: **lint** and **deploy**.

#### 12.2. Verify that the production web servers now respond with the correct content:

```
[student@workstation cicd-tower]$ curl servera
Hello World
[student@workstation cicd-tower]$ curl serverb
Hello World
```

#### 12.3. Return to the web browser session where you are logged into **Ansible Tower** and from the menu on the left, select **Jobs**, and verify that the **Deploy Prod WebServers** successfully.

| JOBS                        |                                  |
|-----------------------------|----------------------------------|
| JOBS (12)                   |                                  |
| SEARCH                      | <input type="button" value="Q"/> |
| KEY                         |                                  |
| 13 - Deploy Prod WebServers | Playbook Run                     |
| 14 - CI/CD Prod Project     | SCM Update                       |
| 10 - Deploy Test WebServers | Playbook Run                     |
| 11 - CI/CD Test Project     | SCM Update                       |
| 7 - Deploy Test WebServers  | Playbook Run                     |
| 8 - CI/CD Test Project      | SCM Update                       |
| 6 - CI/CD Test Project      | SCM Update                       |

**Figure 13.13: Deploy Prod Web Servers Job executes successfully**

## Finish

On the **workstation** machine, run the **lab cicd-tower finish** script to complete this lab.

```
[student@workstation $ lab cicd-tower finish
```

This concludes the guided exercise.

# Summary

---

In this chapter, you learned:

- Pipelines further automation efforts by allowing commands to be executed on merge by the developer.
- To create a pipeline, additional tooling is required such as GitLab or Jenkins.
- Using GitLab, pipelines execute in runners which provide the environment for the commands to be executed.
- Pipelines consist of jobs that represent the various stages at which jobs execute.
- Pipeline jobs consist of commands that execute on the runner.
- Developers merge code to the shared repository for pipelines to execute.
- Pipelines can trigger Job Templates in Ansible Tower.

## Chapter 14

# 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.

## 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.service - PostgreSQL database server
 Loaded: loaded (/usr/lib/systemd/system/postgresql.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)
```

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.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.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 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
```

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 Do Not Appear in the List of Job Templates

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 State

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 in the **/var/** directory.
- 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'
```

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 will be able to:

- Start, stop, and restart Ansible Tower services.
- Use the log files to troubleshoot Ansible Tower.
- Use the **awx-manage** utility to create a new superuser and reset a superuser password.

## Before You Begin

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 start**, which prepares the **tower.lab.example.com** server for this exercise.

```
[student@workstation ~]$ lab admin-troubleshoot start
```

- 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
```

- 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 the **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 with the **status** parameter.

```
[root@tower ~]# ansible-tower-service status
Showing Tower Status
● postgresql.service - PostgreSQL database server
 Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; vendor
 preset: disabled)
 Active: inactive (dead) since Wed 2018-11-21 04:20:43 EDT; 24min ago
 ...output omitted...
● rabbitmq-server.service - RabbitMQ broker
 Loaded: loaded (/usr/lib/systemd/system/rabbitmq-server.service; enabled;
 vendor preset: disabled)
 Active: active (running) since Wed 2018-11-21 04:20:47 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)
```

Notice that the **postgresql** service has the **inactive (dead)** status. The Ansible Tower infrastructure requires this service to be running. Take note of the date and time associated with the status: **Wed 2018-11-21 04:20:43 EDT** in the previous output. You use that information in a later step.

- 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.service
Redirecting to /bin/systemctl stop rabbitmq-server.service
Redirecting to /bin/systemctl stop nginx.service
Redirecting to /bin/systemctl stop supervisord.service
Redirecting to /bin/systemctl start postgresql.service
Redirecting to /bin/systemctl start rabbitmq-server.service
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 controls PostgreSQL as a **postgresql.service** unit through **systemctl**, so you can investigate further using that tool:

```
[root@tower ~]# systemctl status postgresql -l

● postgresql.service - PostgreSQL database server
 Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; vendor
 preset: disabled)
 Active: active (running) since Tue 2019-05-14 09:13:25 UTC; 1min 1s ago
 Process: 4574 ExecStartPre=/usr/libexec/postgresql-check-db-dir postgresql
 (code=exited, status=0/SUCCESS)
 Main PID: 4577 (postmaster)
 Tasks: 14 (limit: 23895)
 Memory: 48.9M
 CGroup: /system.slice/postgresql.service
 ├─4577 /usr/bin/postmaster -D /var/lib/pgsql/data
 ├─4578 postgres: logger process
 ├─4580 postgres: checkpointer process
 ├─4581 postgres: writer process
 ├─4582 postgres: wal writer process
 ├─4583 postgres: autovacuum launcher process
 ├─4584 postgres: stats collector process
 ├─5071 postgres: awx awx 127.0.0.1(46392) idle
 ├─5072 postgres: awx awx 127.0.0.1(46394) idle
 ├─5073 postgres: awx awx 127.0.0.1(46398) idle
 ├─5074 postgres: awx awx 127.0.0.1(46400) idle
 ├─5085 postgres: awx awx 127.0.0.1(46406) idle
 ├─5092 postgres: awx awx 127.0.0.1(46420) idle
 └─5097 postgres: awx awx 127.0.0.1(46424) idle

May 14 09:13:25 tower.lab.example.com systemd[1]: Starting PostgreSQL database
server...
May 14 09:13:25 tower.lab.example.com postmaster[4577]: < 2019-05-14 09:13:25.917
UTC >LOG: redirecting log output to logging collector process
May 14 09:13:25 tower.lab.example.com postmaster[4577]: < 2019-05-14 09:13:25.917
UTC >HINT: Future log output will appear in directory "pg_log".
May 14 09:13:25 tower.lab.example.com systemd[1]: Started PostgreSQL database
server.
```

The PostgreSQL server appears to be logging to a **pg\_log** directory. Use the **find** command to locate that directory:

```
[root@tower ~]# find / -name pg_log
/var/lib/pgsql/data/pg_log
```

- 4.2. Change to the **/var/lib/pgsql/data/pg\_log** directory and inspect the log files.

```
[root@tower ~]# cd /var/lib/pgsql/data/pg_log
[root@tower pg_log]# ls -l
...output omitted...
-rw----- 1 postgres postgres 92745 Mar 20 23:55 postgresql-Tue.log
-rw----- 1 postgres postgres 68055 Mar 21 04:55 postgresql-Wed.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 you 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 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 **supervisord.log** file.

```
[root@tower pg_log]# cd /var/log/supervisor
[root@tower supervisor]# tail -n 50 supervisord.log
...output omitted...
2019-05-14 09:13:20,735 INFO exited: exit-event-listener (terminated by SIGTERM;
not expected)
2019-05-14 09:13:20,735 WARN received SIGTERM indicating exit request
```

```

2019-05-14 09:13:20,737 INFO waiting for awx-dispatcher, awx-callback-receiver,
awx-channels-worker, awx-uwsgi, awx-daphne to die
2019-05-14 09:13:20,763 INFO stopped: awx-channels-worker (terminated by SIGTERM)
2019-05-14 09:13:21,201 INFO stopped: awx-callback-receiver (exit status 0)
2019-05-14 09:13:21,220 INFO stopped: awx-dispatcher (exit status 0)
2019-05-14 09:13:22,747 INFO stopped: awx-uwsgi (exit status 0)
2019-05-14 09:13:23,749 INFO waiting for awx-daphne to die
2019-05-14 09:13:25,751 WARN killing 'awx-daphne' (1553) with SIGKILL
2019-05-14 09:13:25,757 INFO stopped: awx-daphne (terminated by SIGKILL)
2019-05-14 09:13:45,057 CRIT Supervisor is running as root. Privileges were not
dropped because no user is specified in the config file. If you intend to run as
root, you can set user=root in the config file to avoid this message.
2019-05-14 09:13:45,057 INFO Included extra file "/etc/supervisord.d/tower.ini"
during parsing
2019-05-14 09:13:45,057 INFO Increased RLIMIT_NOFILE limit to 4096
2019-05-14 09:13:45,070 INFO RPC interface 'supervisor' initialized
2019-05-14 09:13:45,071 CRIT Server 'unix_http_server' running without any HTTP
authentication checking
2019-05-14 09:13:45,072 INFO daemonizing the supervisord process
2019-05-14 09:13:45,073 INFO supervisord started with pid 5014
2019-05-14 09:13:46,077 INFO spawned: 'exit-event-listener' with pid 5015
2019-05-14 09:13:46,081 INFO spawned: 'awx-dispatcher' with pid 5016
2019-05-14 09:13:46,084 INFO spawned: 'awx-callback-receiver' with pid 5017
2019-05-14 09:13:46,087 INFO spawned: 'awx-channels-worker' with pid 5018
2019-05-14 09:13:46,091 INFO spawned: 'awx-uwsgi' with pid 5019
2019-05-14 09:13:46,094 INFO spawned: 'awx-daphne' with pid 5020
...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 9125, uptime 0:05:03
tower-processes:awx-callback-receiver RUNNING pid 9127, uptime 0:05:03
tower-processes:awx-channels-worker RUNNING pid 9128, uptime 0:05:03
tower-processes:awx-daphne RUNNING pid 9130, uptime 0:05:03
tower-processes:awx-dispatcher RUNNING pid 9126, uptime 0:05:03
tower-processes:awx-uwsgi RUNNING pid 9129, uptime 0:05:03
```

- ▶ 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]# grep job_templates /var/log/tower/tower.log
...output omitted...
2019-05-14 09:06:19,580 WARNING awx.api.generics status 404 received by user
admin attempting to access /api/v1/job_templates/55555/launch/ from 172.25.250.9
2019-05-14 09:06:19,591 WARNING django.request Not Found: /api/v1/
job_templates/55555/launch/
...output omitted...
```

The log messages here indicate that something is trying to launch a nonexistent Job Template.

- ▶ 8. This attempt by **admin** to use the API to launch a nonexistent Job Template with ID 55555 from 172.25.250.9 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]# grep 55555 /var/log/nginx/access.log
...output omitted...
172.25.250.9 - admin [14/May/2019:09:06:19 +0000] "POST /api/v1/
job_templates/55555/launch/ HTTP/1.1" 404 23 "-" "curl/7.61.1" "-"
...output omitted...
```

This concludes the first part of the exercise.

- ▶ 9. In the second part of this exercise, you use the **awx-manage** command to create an additional Ansible Tower system administrator account and change its password.

First, create a new system administrator user with the **awx-manage** command and the **createsuperuser** subcommand. Use the following information:

| Field         | Value                         |
|---------------|-------------------------------|
| User name     | <b>admin2</b>                 |
| Email address | <b>admin2@lab.example.com</b> |
| Password      | <b>redhat</b>                 |

```
[root@tower supervisor]# awx-manage createsuperuser
Username (leave blank to use 'root'): admin2
Email address: admin2@lab.example.com
Password: redhat
Password (again): redhat
Superuser created successfully.
```

- 10. 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 supervisor]# awx-manage changepassword admin2
Changing password for user 'admin2'
Password: redhat2
Password (again): redhat2
Password changed successfully for user 'admin2'
```

- 11. Log out of the **tower** system.

```
[root@tower supervisor]# exit
```

- 12. To confirm the change, open Firefox on **workstation** and log in to Ansible Tower as the **admin2** user with the new **redhat2** password.

## Finish

On **workstation**, run the **lab admin-troubleshoot finish** script to clean up this exercise.

```
[student@workstation ~]$ lab admin-troubleshoot finish
```

This concludes the guided exercise.

# 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.

In each case, you need to obtain a TLS certificate for the server, correctly signed by a CA, and a matching private key, both in PEM format. Use these to replace the existing self-signed certificate and key:

- 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 will be able to:

- Replace the default TLS/SSL certificate used by Ansible Tower with a provided certificate appropriate for the server's host name.

## Before You Begin

- 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 start**, which verifies that the Ansible Tower services are running and all the required resources are available.

```
[student@workstation ~]$ lab admin-cert start
```

- 1. Open a terminal on **workstation** and use **ssh** to log in to the **tower** server as **root**. In the following steps, you will replace the Ansible Tower TLS/SSL certificate with a provided certificate appropriate for the server's host name.

```
[student@workstation ~]$ ssh root@tower
```

- 2. Add a new rule to the SELinux policy to set the type to **cert\_t** on the **/etc/tower/tower.cert** and **/etc/tower/tower.key** files. 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/
```

- 3. Back up then remove the existing **/etc/tower/tower.cert** and **/etc/tower/tower.key** files.

```
[root@tower ~]# cp /etc/tower/tower.* /root
[root@tower ~]# rm /etc/tower/tower.*
rm: remove regular file '/etc/tower/tower.cert'? y
rm: remove regular file '/etc/tower/tower.key'? y
```

- 4. 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.

- ▶ 5. 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.service
Redirecting to /bin/systemctl stop rabbitmq-server.service
Redirecting to /bin/systemctl stop nginx.service
Redirecting to /bin/systemctl stop supervisord.service
Redirecting to /bin/systemctl start postgresql.service
Redirecting to /bin/systemctl start rabbitmq-server.service
Redirecting to /bin/systemctl start nginx.service
Redirecting to /bin/systemctl start supervisord.service
[root@tower ~]# exit
```

- ▶ 6. 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.

## Finish

On **workstation**, run the **lab admin-cert finish** script to clean up this exercise.

```
[student@workstation ~]$ lab admin-cert finish
```

This concludes the guided exercise.

# 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.

## 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.

- 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]#
```

- 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
```

- 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 procedure demonstrates how to restore the Ansible Tower infrastructure from an 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",
"SendsSIGHUP": "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",
"SystemCallErrorNumber": "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/backup\\_restore.html](https://docs.ansible.com/ansible-tower/latest/html/administration/backup_restore.html)

## ► 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 will 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.

## Before You Begin

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 start**, which verifies that the Ansible Tower services are running and all the required resources are available.

```
[student@workstation ~]$ lab admin-recovery start
```

- 1. Open a terminal on **workstation** and use **ssh** to log in to the **tower** server as **root**. In the following steps you will create a backup of Ansible Tower from the CLI.

```
[student@workstation ~]$ ssh root@tower
```

- 2. 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-3.5.0-1.el8 original-ks.cfg
```

- 3. 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...)
```

- 4. Change to the **/root/ansible-tower-setup-bundle-3.5.0-1.el8** directory.

```
[root@tower ~]# cd /root/ansible-tower-setup-bundle-3.5.0-1.el8
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]#
```

- 5. Use the **setup.sh -b** command to back up the Ansible Tower configuration and database.

```
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]# ./setup.sh -b
...output omitted...
RUNNING HANDLER [backup : Remove the backup directory.] *****
changed: [localhost] => {"changed": true, "path": "/var/backups/tower/2019-05-14-11:09:04/", "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-3.5.0-1.el8/2019-05-14-11:09:04", "state": "absent"}

PLAY RECAP *****
localhost : ok=37 changed=29 unreachable=0 failed=0 skipped=10 ...

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2019-05-14-11:08:58.log
```

- 6. Verify that the backup was created in the current working directory by issuing the **ls** command.

```
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]# ls -1
backup.yml
bundle
group_vars
install.yml
inventory
inventory.1588.2019-06-18@04:56:26~
licenses
README.md
```

```
restore.yml
roles
setup.sh
tower-backup-2019-06-20-07:04:57.tar.gz
tower-backup-latest.tar.gz
```

- ▶ 7. Change the Ansible Tower **admin** superuser password to **redhat2** using the **awx-manage** command.

```
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]# awx-manage changepassword \
> admin
Changing password for user 'admin'
Password: redhat2
Password (again): redhat2
Password changed successfully for user 'admin'
```

- ▶ 8. 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.
- ▶ 9. Go back to **tower** server and restore the backup from the file created in previous steps. Use the **setup.sh** command with the **-r** 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-3.5.0-1.el8]# ./setup.sh -r
...output omitted...
PLAY RECAP ****
localhost : ok=43 changed=24 unreachable=0 failed=0

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2019-05-14-11:11:28.log
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]# exit
```

- ▶ 10. 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.

## Finish

On **workstation**, run the **lab admin-recovery finish** script to clean up this exercise.

```
[student@workstation ~]$ lab admin-recovery finish
```

This concludes the guided exercise.

## ► 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**.
- Use the **ansible-tower-service** command to manually stop, start, or restart Red Hat Ansible Tower services.
- Log files for Ansible Tower are located in **/var/log/tower** and **/var/log/supervisor**.
- 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**.



## Chapter 15

# Comprehensive Review

### Goal

Demonstrate skills learned in this course by configuring and operating a new organization in Ansible Tower using a provided specification, Ansible projects, and hosts to be provisioned and managed.

### Objectives

- Review tasks from *Advanced Automation: Red Hat Ansible Best Practices*

### Sections

- Comprehensive Review

### Lab

- Lab: Writing YAML Inventory Files
- Lab: Privilege Escalation, Lookups, and Rolling Updates
- Lab: Restoring Ansible Tower from Backup
- Lab: Adding Users and Teams
- Lab: Creating a Custom Dynamic Inventory
- Lab: Configuring Job Templates
- Lab: Configuring Workflow Job Templates, Surveys, and Notifications
- Lab: Testing the Prepared Environment

# Comprehensive Review

---

## Objectives

After completing this section, you should have reviewed and refreshed the knowledge and skills learned in *Advanced Automation: Red Hat Ansible Best Practices*.

## Reviewing Advanced Automation: Red Hat Ansible Best Practices

Before beginning the comprehensive review for this course, you should be comfortable with the topics covered in each chapter.

You can refer to earlier sections in the textbook for extra study.

### **Chapter 1, Developing with Recommended Practices**

Demonstrate and implement recommended practices for effective and efficient Ansible automation.

- Demonstrate and describe common recommended practices for developing and maintaining effective Ansible automation solutions.
- Create and manage Ansible Playbooks in a Git repository using recommended practices.

### **Chapter 2, Managing Inventories**

Manage inventories using advanced features of Ansible.

- Write static inventory files in YAML format instead of the older INI-like format.
- Structure host and group variables using multiple files per host or group, and use special variables to override the host, port, or remote user Ansible uses for a specific host.

### **Chapter 3, Managing Task Execution**

Control and optimize the execution of tasks by Ansible Playbooks.

- Control automatic privilege escalation at the play, role, task, or block level.
- Configure tasks that can run before roles or after normal handlers, and notify multiple handlers at once.
- Label tasks with tags, and run only tasks labeled with specific tags or start playbook execution at a specific task.
- Optimize your playbook to run more efficiently, and use callback plugins to profile and analyze which tasks consume the most time.

### **Chapter 4, Transforming Data with Filters and Plugins**

Populate, manipulate, and manage data in variables using filters and plugins.

- Format, parse, and define the values of variables using filters.
- Populate variables with data from external sources using lookup plugins.
- Use filters and lookup plug-ins to implement iteration over complex data structures.
- Inspect, validate, and manipulate variables containing networking information with filters.

## **Chapter 5, Coordinating Rolling Updates**

Use advanced features of Ansible to manage rolling updates in order to minimize downtime, and to ensure maintainability and simplicity of Ansible Playbooks.

- Run a task for a managed host on a different host, and control whether facts gathered by that task are delegated to the managed host or to the other host.
- Tune behavior of the serial directive when batching hosts for execution, abort the play if too many hosts fail, and create tasks that run only once for each batch or for all hosts in the inventory.

## **Chapter 6, Installing and Accessing Ansible Tower**

Explain what Red Hat Ansible Tower is and demonstrate a basic ability to navigate and use its web user interface.

- Describe the architecture, use cases, and installation requirements of Red Hat 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.

## **Chapter 7, Managing Access with Users and Teams**

Create user accounts and organize them into teams in Red Hat Ansible Tower. Then, assign the users and teams permissions to administer and access resources in the Ansible Tower service.

- Create new users in the web UI, and explain the different types of user in Ansible Tower.
- Create new teams in the web UI, assign users to them, and explain the different roles that can be assigned to users.

## **Chapter 8, Managing Inventories and Credentials**

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.

- 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.

## **Chapter 9, Managing Projects and Launching Ansible Jobs**

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.

- 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.

## **Chapter 10, Constructing Advanced Job Workflows**

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.

- 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.

## **Chapter 11, Communicating with APIs using Ansible**

Interact with REST APIs in Ansible Playbooks, and control Red Hat Ansible Tower using its REST API.

- Control Ansible Tower by accessing its API with **curl**, and in Ansible Playbooks.
- Write playbooks that interact with a REST API to get information from a web service, and to trigger events.

## **Chapter 12, Managing Advanced Inventories**

Manage inventories that are loaded from external files or generated dynamically from scripts or the Ansible Tower smart inventory feature.

- 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.

## **Chapter 13, Creating a Simple CI/CD Pipeline with Ansible Tower**

Build and operate a proof-of-concept CI/CD pipeline based on Ansible automation and integrating Red Hat Ansible Tower.

- Integrate Ansible Tower with a web-based Git repository system such as GitLab or GitHub to build a simple pipeline to automatically deploy playbooks when changes are committed.

## **Chapter 14, Performing Maintenance and Routine Administration of Ansible Tower**

Perform routine maintenance and administration of Ansible Tower.

- 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.

## ▶ Lab

# Refactoring Inventories and Projects

In this review, you will refactor a plain project to the recommended structure. You will also convert a Red Hat Ansible Engine inventory file from INI format to YAML format.

## Outcomes

You should be able to:

- Refactor a plain project to the recommended structure.
- Convert an inventory file from the INI format to the YAML format.

## Before You Begin

Set up your computers for this exercise by logging in to the **workstation** machine as the **student** user, and running the following command:

```
[student@workstation ~]$ lab review-cr1 start
```

## Instructions

The **project-comp-review** Git repository contains a **site.yml** playbook that configures a front end load balancer and a pool of back-end web servers.

As the **student** user on the **workstation** machine, change to the **~/git-repos** directory. Clone the Git repository at <http://git.lab.example.com:8081/git/project-comp-review.git> into that directory.

Review the project playbooks and the inventory file, and refactor the project according the following specifications:

- Modify the **inventory** file by converting it to YAML syntax. The resulting file should contain two host groups.
- In the **inventory** file, rename the host group **mylocal\_servers** to **web\_servers**.
- Modify the playbooks **deploy\_apache.yml** and **deploy\_webapp.yml** to use the host group **web\_servers** instead of **mylocal\_server** for all plays.
- In the **webapp** role, change the variable **msg** to **webapp\_message**. Update any references to this variable in the project files.
- In the **webapp** role, change the variable **vers** to **webapp\_version**. Update any references to this variable in the project files.
- Verify that the playbook **site.yml** executes with no errors.
- After successful execution, commit your changes to Git with a message of **Refactoring of the review project** and push them to the remote repository.

## Evaluation

Grade your work by running the **lab review-cr1 grade** command from your **workstation** machine. Correct any reported failures in your local Git repository, and then commit and push the changes. After changes are pushed to the remote repository, rerun the script. Repeat this process until you receive a passing score for all criteria.

```
[student@workstation ~]$ lab review-cr1 grade
```

This concludes the lab.

## ► Solution

# Refactoring Inventories and Projects

In this review, you will refactor a plain project to the recommended structure. You will also convert a Red Hat Ansible Engine inventory file from INI format to YAML format.

## Outcomes

You should be able to:

- Refactor a plain project to the recommended structure.
- Convert an inventory file from the INI format to the YAML format.

## Before You Begin

Set up your computers for this exercise by logging in to the **workstation** machine as the **student** user, and running the following command:

```
[student@workstation ~]$ lab review-cr1 start
```

## Instructions

The **project-comp-review** Git repository contains a **site.yml** playbook that configures a front end load balancer and a pool of back-end web servers.

As the **student** user on the **workstation** machine, change to the **~/git-repos** directory. Clone the Git repository at <http://git.lab.example.com:8081/git/project-comp-review.git> into that directory.

Review the project playbooks and the inventory file, and refactor the project according the following specifications:

- Modify the **inventory** file by converting it to YAML syntax. The resulting file should contain two host groups.
- In the **inventory** file, rename the host group **my\_local\_servers** to **web\_servers**.
- Modify the playbooks **deploy\_apache.yml** and **deploy\_webapp.yml** to use the host group **web\_servers** instead of **my\_local\_server** for all plays.
- In the **webapp** role, change the variable **msg** to **webapp\_message**. Update any references to this variable in the project files.
- In the **webapp** role, change the variable **vers** to **webapp\_version**. Update any references to this variable in the project files.
- Verify that the playbook **site.yml** executes with no errors.
- After successful execution, commit your changes to Git with a message of **Refactoring of the review project** and push them to the remote repository.

1. Clone the Git repository.

1. On the **workstation** machine, open a terminal. Create the **git-repos** directory in your home directory.

```
[student@workstation ~]$ mkdir -p ~/git-repos
```

2. Change to the **git-repos** directory and use the **git** command to clone the Git repository located at <http://git.lab.example.com:8081/git/project-comp-review.git>.

```
[student@workstation ~]$ cd git-repos
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/project-comp-review.git
Cloning into 'project-comp-review'...
remote: Enumerating objects: 53, done.
remote: Counting objects: 100% (53/53), done.
remote: Compressing objects: 100% (35/35), done.
remote: Total 53 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (53/53), done.
```

3. Review the files in the project. The project contains the main inventory, **inventory**, a **site.yml** playbook that invokes a set of three playbooks, and a set of roles. The playbooks invoke roles that configure the load balancer, the web servers, and the web application.

```
[student@workstation git-repos]$ tree project-comp-review/
project-comp-review/
├── ansible.cfg
├── deploy_apache.yml
├── deploy_haproxy.yml
├── deploy_webapp.yml
├── group_vars
│ └── lb_servers.yml
├── inventory
└── roles
... output omitted ...
```

2. Convert the inventory to YAML. Rename the host group from **mylocal\_servers** to **web\_servers**.

- 2.1. Open the **~/git-repos/project-comp-review/inventory** file in a text editor. The inventory contains two host groups, **lb\_servers** and **mylocal\_servers**.

The host group **lb\_servers** contains **servera.lab.example.com**. The host group **mylocal\_servers** (to be renamed **web\_servers**) contains **serverb.lab.example.com** and **serverc.lab.example.com**.

Rename the **inventory** file to **inventory.yml** and convert it from INI to YAML format. In that file, rename the host group **mylocal\_servers** to **web\_servers**.

```
[student@workstation git-repos]$ cd project-comp-review/
[student@workstation project-comp-review]$ mv inventory inventory.yml
[student@workstation project-comp-review]$ vim inventory.yml
```

The resulting file should contain two host groups. When updated, the inventory should match the following content:

**Note**

Do not forget to add the **hosts** entry and colons to each line.

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

web_servers:
 hosts:
 server[b:c].lab.example.com:
```

3. In the playbooks, update references to the **mylocal\_servers** host group to use **web\_servers**.

- 3.1. Both the **deploy\_apache.yml** playbook, and the **deploy\_webapp.yml** playbook mention the **mylocal\_servers** host group.

Use a text editor to edit **deploy\_apache.yml**. Locate the **hosts** entry that point to the host group. Update the value of the variable with **web\_servers**.

The file should read as follows:

```
- name: Ensure Apache is deployed
 hosts: web_servers
 force_handlers: True
 gather_facts: no
... output omitted ...
```

- 3.2. Repeat the same operation for **deploy\_webapp.yml**. Update the **hosts** variable with a value of **web\_servers**. After updating the file, it should read as follows:

```
- name: Ensure the web application is deployed
 hosts: web_servers
 gather_facts: no
... output omitted ...
```

4. In the role **webapp**, rename the role variable **msg** to **webapp\_message**. Rename the role variable **vers** to **webapp\_version**.

- 4.1. The **webapp** role contains two variables, **msg** and **vers**. Rename **msg** variable to **webapp\_message** and rename the **vers** variable to **webapp\_version**.

Start by editing the role main playbook located in the **roles** directory. The path relative to the **project-comp-review** is **roles/webapp/tasks/main.yml**.

Rename both the **msg** and **vers** variables in the **content** line. After update, the file should read as the following:

```

tasks file for webapp

- name: Ensure placeholder content is deployed
 copy:
 content: "{{ webapp_message }} (version {{ webapp_version }})\n"
 dest: /var/www/html/index.html
```

- 4.2. These two variables are defined in the role's default variables file declared in the **defaults** directory. Use your favorite editor to edit the **main.yml** playbook, in the **defaults** directory.

Rename the **vers** variable to **webapp\_version** and the **msg** variable to **webapp\_message**. After updating the file it should contain the following:

```
webapp_version: v1.0
webapp_message: "This is {{ inventory_hostname }}."
```

5. Run the **site.yml** playbook.

- 5.1. Make sure you are in the **~/git-repos/project-comp-review** directory and verify that the **site.yml** playbook executes with no errors.

Use the **ansible-playbook** command against the **site.yml** playbook. Use the **-i inventory.yml** option to specify the local inventory.

```
[student@workstation project-comp-review]$ ansible-playbook -i inventory.yml \
> site.yml
PLAY [Ensure HAProxy is deployed] ****
TASK [firewall : Ensure Firewall Sources Configuration] ****
ok: [servera.lab.example.com] => (item={'port': '80/tcp'})

TASK [haproxy : Ensure haproxy packages are present] ****
ok: [servera.lab.example.com]

TASK [haproxy : Ensure haproxy is started and enabled] ****
ok: [servera.lab.example.com]
... output omitted ...

PLAY RECAP ****
servera.lab.example.com : ok=4 changed=0 unreachable=0 ...
serverb.lab.example.com : ok=5 changed=1 unreachable=0 ...
serverc.lab.example.com : ok=5 changed=1 unreachable=0 ...
```

6. Commit your changes and push them to the remote repository.

- 6.1. Use **git status** to review the list of changes.

```
[student@workstation project-comp-review]$ git status
On branch master
Your branch is up to date with 'origin/master'.
```

```
Changes not staged for commit:
(use "git add/rm <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified: deploy_apache.yml
modified: deploy_webapp.yml
deleted: inventory
modified: roles/webapp/defaults/main.yml
modified: roles/webapp/tasks/main.yml

Untracked files:
(use "git add <file>..." to include in what will be committed)

inventory.yml

no changes added to commit (use "git add" and/or "git commit -a")
```

6.2. Add all the files to the staging area.

```
[student@workstation project-comp-review]$ git add --all
```

6.3. Run the **git commit -m** command to add your changes to the index and to commit your changes. Use a commit message of **Refactoring of the review project**.

```
[student@workstation project-comp-review]$ git commit -m \
> "Refactoring of the review project"
[master dc81638] Refactoring of the review project
 5 files changed, 11 insertions(+), 11 deletions(-)
```

6.4. Push your changes to the remote repository.

```
[student@workstation project-comp-review]$ git push
Enumerating objects: 21, done.
Counting objects: 100% (21/21), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (11/11), 928 bytes | 464.00 KiB/s, done.
Total 11 (delta 5), reused 0 (delta 0)
To http://git.lab.example.com:8081/git/project-comp-review.git
 3e3fcdd..dc81638 master -> master
```

## Evaluation

Grade your work by running the **lab review-cr1 grade** command from your **workstation** machine. Correct any reported failures in your local Git repository, and then commit and push the changes. After changes are pushed to the remote repository, rerun the script. Repeat this process until you receive a passing score for all criteria.

```
[student@workstation ~]$ lab review-cr1 grade
```

This concludes the lab.

## ▶ Lab

# Privilege Escalation, Lookups, and Rolling Updates

In this review, you will modify roles to use privilege escalation on only those tasks that need it, configure a play to log execution using a lookup, and configure the play to update one host and abort the rest of the play if that host fails.

## Outcomes

You should be able to:

- Identify and apply existing lookup plug-ins and filters to solve problems.
- Limit privilege escalation to only the tasks that need it.
- Process a play in batches and abort the play if there are too many failures.

## Before You Begin

Set up your computers for this exercise by logging in to the **workstation** machine as the **student** user, and run the following command:

```
[student@workstation ~]$ lab review-cr2 start
```

## Instructions

The **review-cr2** Git repository that you use in this activity contains a **site.yml** playbook that configures a front-end load balancer and a pool of back-end web servers.

- Clone the git repository at <http://git.lab.example.com:8081/git/review-cr2.git> in the **/home/student/git-repos** folder. Before you grade your work, be sure to commit and push all changes back to the remote repository.
- Refactor the second task of the **webapp** role to generate an HTML file for each location in the **~/git-repos/review-cr2/locations.yml** file. Use the **webapp** role's **location.html.j2** template to generate each HTML location file. Write all HTML files to the **/var/www/html** directory. The name of the location file must match the name of the location, with a **.html** extension.
- Apply fine-grained privilege escalation for tasks as a good practice to avoid potential authorization issues. Remove privilege escalation from the Ansible configuration to prevent all tasks from executing as a privileged user by default. Then, enable privilege escalation for tasks and handlers that need it:
  - The block of tasks for the **apache** role.
  - The only handler and the only task in the **firewall** role.
  - Both handlers and task blocks in the **haproxy** role.
  - The block of tasks in the **webapp** role. Make sure this block contains all tasks in the file.

- Introduce logging tasks to register the beginning and the end of the web application deployment.

Before deploying or updating the web application, add a line to the end of the `/tmp/times.txt` file on the Ansible controller (**workstation**). The line must contain the date and time for the deployment, obtained from the `date` command. Use the `lineinfile` module to create and update the log file, and the `pipe lookup` plug-in to retrieve the current date and time. The line must contain the literal string **Deploying version**, and the version of the web app. The line must contain the host name of the server where the deployment occurs, as defined in the inventory file.

After deploying or updating the web application, add a line to the end of the `/tmp/times.txt` file on the **workstation** machine. The line must include the same contents as described in the previous instruction, but the literal string must be **Deployment complete**.

The following lines are an example of the content to be logged:

```
Thu Jun 13 22:10:24 UTC 2019 Deploying version v1.0 to serverc.lab.example.com
Thu Jun 13 22:10:44 UTC 2019 Deployment complete v1.0 to serverc.lab.example.com
```

- Change the `deploy_webapp.yml` playbook to enable batch updates to managed hosts. The first batch must only update a single host. If the update succeeds, the second batch must update the rest of the hosts.

If a single host fails to update, the playbook must stop executing immediately.

## Evaluation

Prior to grading your work, be sure to commit and push all changes to the remote repository. The grading script executes checks against the content of the remote repository, not the files on the **workstation** machine.

Grade your work by running the `lab review-cr2 grade` command from your **workstation** machine.

```
[student@workstation ~]$ lab review-cr2 grade
```

Correct any reported failures in your local Git repository, and then commit and push the changes. Rerun the script after pushing the changes to the remote repository. Repeat this process until you receive a passing score for all criteria.

## Finish

From the **workstation** machine, run the `lab review-cr2 finish` command to complete this lab.

```
[student@workstation ~]$ lab review-cr2 finish
```

This concludes the lab.

## ► Solution

# Privilege Escalation, Lookups, and Rolling Updates

In this review, you will modify roles to use privilege escalation on only those tasks that need it, configure a play to log execution using a lookup, and configure the play to update one host and abort the rest of the play if that host fails.

## Outcomes

You should be able to:

- Identify and apply existing lookup plug-ins and filters to solve problems.
- Limit privilege escalation to only the tasks that need it.
- Process a play in batches and abort the play if there are too many failures.

## Before You Begin

Set up your computers for this exercise by logging in to the **workstation** machine as the **student** user, and run the following command:

```
[student@workstation ~]$ lab review-cr2 start
```

## Instructions

The **review-cr2** Git repository that you use in this activity contains a **site.yml** playbook that configures a front-end load balancer and a pool of back-end web servers.

- Clone the git repository at <http://git.lab.example.com:8081/git/review-cr2.git> in the **/home/student/git-repos** folder. Before you grade your work, be sure to commit and push all changes back to the remote repository.
- Refactor the second task of the **webapp** role to generate an HTML file for each location in the **~/git-repos/review-cr2/locations.yml** file. Use the **webapp** role's **location.html.j2** template to generate each HTML location file. Write all HTML files to the **/var/www/html** directory. The name of the location file must match the name of the location, with a **.html** extension.
- Apply fine-grained privilege escalation for tasks as a good practice to avoid potential authorization issues. Remove privilege escalation from the Ansible configuration to prevent all tasks from executing as a privileged user by default. Then, enable privilege escalation for tasks and handlers that need it:
  - The block of tasks for the **apache** role.
  - The only handler and the only task in the **firewall** role.
  - Both handlers and task blocks in the **haproxy** role.
  - The block of tasks in the **webapp** role. Make sure this block contains all tasks in the file.

- Introduce logging tasks to register the beginning and the end of the web application deployment.

Before deploying or updating the web application, add a line to the end of the `/tmp/times.txt` file on the Ansible controller (**workstation**). The line must contain the date and time for the deployment, obtained from the `date` command. Use the `lineinfile` module to create and update the log file, and the `pipe lookup` plug-in to retrieve the current date and time. The line must contain the literal string **Deploying version**, and the version of the web app. The line must contain the host name of the server where the deployment occurs, as defined in the inventory file.

After deploying or updating the web application, add a line to the end of the `/tmp/times.txt` file on the **workstation** machine. The line must include the same contents as described in the previous instruction, but the literal string must be **Deployment complete**.

The following lines are an example of the content to be logged:

```
Thu Jun 13 22:10:24 UTC 2019 Deploying version v1.0 to serverc.lab.example.com
Thu Jun 13 22:10:44 UTC 2019 Deployment complete v1.0 to serverc.lab.example.com
```

- Change the `deploy_webapp.yml` playbook to enable batch updates to managed hosts. The first batch must only update a single host. If the update succeeds, the second batch must update the rest of the hosts.

If a single host fails to update, the playbook must stop executing immediately.

- Clone the Git repository in `http://git.lab.example.com:8081/git/review-cr2.git` to the **/home/student/git-repos** directory.

- From a terminal, create the directory **/home/student/git-repos** if it does not already exist.

```
[student@workstation ~]$ mkdir -p /home/student/git-repos/
```

- Change to this directory:

```
[student@workstation ~]$ cd git-repos/
```

- Clone the **review-cr2** repository and change to this directory.

```
[student@workstation git-repos]$ git clone \
> http://git.lab.example.com:8081/git/review-cr2.git
Cloning into 'review-cr2'...
remote: Enumerating objects: 62, done.
remote: Counting objects: 100% (62/62), done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 62 (delta 7), reused 0 (delta 0)
Unpacking objects: 100% (62/62), done.
[student@workstation git-repos]$ cd review-cr2
```

2. Update the `~/git-repos/review-cr2/roles/webapp/tasks/main.yml` to generate web pages based on the locations file. Modify the second task to parse the contents of the `~/git-repos/review-cr2/locations.yml` file. This file contains a list of locations in YAML format. In the second task, iterate that list and apply each element contents to the `~/git-repos/review-cr2/roles/webapp/templates/location.html.j2` template.

- 2.1. Use your preferred editor to open the `~/git-repos/review-cr2/roles/webapp/tasks/main.yml` file.

Add a `loop` keyword that iterates through all locations. You can use `file lookup` to retrieve the contents of the `locations.yml` file. Use the `from_yaml` filter to parse the retrieved contents into an iterable form.

Observe that the second task creates a single `/var/www/html/location.html` based on the `location.html.j2` template, instead of multiple files. Use the `{{ item.name }}` variable to create multiple files, based on the name of the location.

The resulting task should display as follows:

```
- name: Ensure location files are created
 template:
 src: location.html.j2
 dest: "/var/www/html/{{ item.name }}.html"
 loop: "{{ lookup('file', 'locations.yml') | from_yaml }}"
```

3. Apply fine-grained control privilege escalation in tasks as a good practice to avoid potential authorization issues.

To remove privilege escalation from the Ansible configuration, edit the `~/git-repos/review-cr2/ansible.cfg` file, and remove `become=true` from the `[privilege_escalation]` section.

Add the `become: true` keyword to the appropriate task and handler files. Be careful to indent the keyword correctly, so it applies to an entire block and not to an individual task.

- 3.1. Update the `apache` role to use fine-grained privilege escalation control.

Add escalation to the `block` keyword of the `~/git-repos/review-cr2/roles/apache/tasks/main.yml` file:

```

tasks file for apache
- block:
 - name: Ensure httpd packages are installed
 ...output omitted...
 - name: Ensure SELinux allows httpd connections to a remote database
 ...output omitted...
 - name: Ensure httpd service is started and enabled
 ...output omitted...
 become: true
```

- 3.2. Update the `firewall` role to use fine-grained privilege escalation control.

Add the escalation to the only task in the tasks file `~/git-repos/review-cr2/roles/firewall/tasks/main.yml`.

```

tasks file for firewall

- name: Ensure Firewall Sources Configuration
...output omitted...
become: true
```

Add the escalation to the only handler the handlers file **~/git-repos/review-cr2/roles/firewall/handlers/main.yml**:

```

handlers file for firewall

- name: reload firewalld
...output omitted...
become: true
```

3.3. Update the **haproxy** role to use fine-grained privilege escalation control.

Add the escalation to each of the handlers in the **~/git-repos/review-cr2/roles/haproxy/handlers/main.yml** file:

```

handlers file for haproxy
- name: restart haproxy
...output omitted...
become: true

- name: reload haproxy
...output omitted...
become: true
```

Add the escalation to the **block** keyword if the tasks file **~/git-repos/review-cr2/roles/haproxy/tasks/main.yml**:

```

tasks file for haproxy
- block:
 - name: Ensure haproxy packages are present
 ...output omitted...
 - name: Ensure haproxy is started and enabled
 ...output omitted...
 - name: Ensure haproxy configuration is set
 ...output omitted...
become: true
```

3.4. Update the **webapp** role to use fine-grained privilege escalation control.

Add the escalation to the **block** keyword of the tasks file **~/git-repos/review-cr2/roles/webapp/tasks/main.yml**:

```

tasks file for webapp
- block:
 - name: Ensure stub web content is deployed
 ...output omitted...
 - name: Ensure location files are created
 ...output omitted...
become: true

```

4. Introduce logging tasks to register the start and end of the deployment on your control node.

- 4.1. Use a text editor to edit the file `~/git-repos/review-cr2/deploy_webapp.yml`.

Add a **pre\_tasks** keyword with a single task. That task executes the **lineinfile** module to append the expected record to the **/tmp/times.txt** file.

Note that the file, and hence the task, must apply to the controller. Add the **delegate\_to: localhost** keyword to ensure local application.

```

pre_tasks:
 - name: Ensure logging the deployment start
 lineinfile:
 path: /tmp/times.txt
 create: yes
 line: "{{ lookup('pipe', 'date') }}: Deploying version
{{ webapp_version }} to {{ inventory_hostname }}"
delegate_to: localhost

```

The exact format of the record may vary. However, each line must contain the text **Deploying version** and the inventory host name.

- 4.2. In the same file, add a **post\_tasks** keyword with a single task that writes the deployment completion record. Like the previous **pre\_tasks** task, the **post\_tasks** task must also run on the controller.

```

post_tasks:
 - name: Ensure logging the deployment completion
 lineinfile:
 path: /tmp/times.txt
 create: yes
 line: "{{ lookup('pipe', 'date') }}: Deployment complete
{{ webapp_version }} to {{ inventory_hostname }}"
delegate_to: localhost

```

Make sure the record contains the **Deployment complete** string and the inventory host name.

5. Change the **deploy\_webapp.yml** playbook to enable batch updates to managed hosts. The first batch must only update a single host. If the update succeeds, the second batch must update the rest of the hosts.

- 5.1. Edit the `~/git-repos/review-cr2/deploy_webapp.yml` file to introduce a **serial** keyword. The first batch must only apply to one host, while the second batch applies changes to all the remaining hosts:

```
- name: Ensure Web App is deployed
hosts: web_servers
gather_facts: false
serial:
 - 1
 - 100%
...output omitted...
```

- 5.2. If a single host in any batch fails to update, the playbook must halt immediately. Use the **max\_fail\_percentage** keyword to indicate Ansible that behavior.

```
- name: Ensure Web App is deployed
hosts: web_servers
gather_facts: false
serial:
 - 1
 - 100%
max_fail_percentage: 0
```

- 6.** Verify the playbook runs correctly:

- 6.1. Execute the playbook and validate it succeeds:

```
[student@workstation review-cr2]$ ansible-playbook site.yml
...output omitted...

PLAY RECAP ****
servera...: ok=6 changed=6 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
serverb...: ok=9 changed=6 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
serverc...: ok=9 changed=6 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
serverd...: ok=9 changed=6 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

- 6.2. Verify the location contents are available in the web application:

```
[student@workstation review-cr2]$ curl \
> -s http://servera.lab.example.com/Sydney.html
Hello from Sydney.

The office address is:
Level 35
100 Miller Street
North Sydney NSW 2060
Australia
```

- 6.3. Check the log file for the expected records:

```
[student@workstation review-cr2]$ tail -6 /tmp/times.txt
Fri Jun 14 13:12:13 UTC 2019: Deploying version v1.0 to serverb.lab.example.com
Fri Jun 14 13:12:31 UTC 2019: Deployment complete v1.0 to serverb.lab.example.com
Fri Jun 14 13:12:31 UTC 2019: Deploying version v1.0 to serverc.lab.example.com
Fri Jun 14 13:12:31 UTC 2019: Deploying version v1.0 to serverd.lab.example.com
Fri Jun 14 13:12:49 UTC 2019: Deployment complete v1.0 to serverc.lab.example.com
Fri Jun 14 13:12:49 UTC 2019: Deployment complete v1.0 to serverd.lab.example.com
```

Note **serverb** deployment starts and completes in the first batch. The rest of the servers may start and complete in any order.

#### 6.4. Commit and push your changes to the Git repository:

```
[student@workstation review-cr2]$ git add .
[student@workstation review-cr2]$ git commit -m "Lab contents"
[master 7a3d4db] Lab contents
 8 files changed, 23 insertions(+), 11 deletions(-)
[student@workstation review-cr2]$ git push
...output omitted...
To http://git.lab.example.com:8081/git/review-cr2.git
 4633518..7a3d4db master -> master
```

## Evaluation

Prior to grading your work, be sure to commit and push all changes to the remote repository. The grading script executes checks against the content of the remote repository, not the files on the **workstation** machine.

Grade your work by running the **lab review-cr2 grade** command from your **workstation** machine.

```
[student@workstation ~]$ lab review-cr2 grade
```

Correct any reported failures in your local Git repository, and then commit and push the changes. Rerun the script after pushing the changes to the remote repository. Repeat this process until you receive a passing score for all criteria.

## Finish

From the **workstation** machine, run the **lab review-cr2 finish** command to complete this lab.

```
[student@workstation ~]$ lab review-cr2 finish
```

This concludes the lab.

## ► 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 the **workstation** machine as the **student** user, and run the following command:

```
[student@workstation ~]$ lab review-cr3 start
```

## Instructions

On **tower.lab.example.com**, use the backup 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 Tower as the **admin** user with a password of **redhat**, and verify that the configuration is restored. One indicator of success is that there should only be one Job Template, **Demo Job Template**, present after the restoration.

This concludes the lab.

## ► 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 the **workstation** machine as the **student** user, and run the following command:

```
[student@workstation ~]$ lab review-cr3 start
```

## Instructions

On **tower.lab.example.com**, use the backup 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-3.5.0-1.el8**.

```
[root@tower ~]# cd /root/ansible-tower-setup-bundle-3.5.0-1.el8
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]#
```

- 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-3.5.0-1.el8]# 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

2019-06-03 09:21:58 (50.6 MB/s) - 'tower-backup-latest.tar.gz' saved [54068/54068]
```

1.4. Using the **setup.sh** command with the **-r** option, restore Ansible Tower from backup.

```
[root@tower ansible-tower-setup-bundle-3.5.0-1.el8]# ./setup.sh -r
...output omitted...
PLAY RECAP

localhost : ok=43 changed=24 unreachable=0 failed=0
skipped=14

The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2019-06-03-04:26:17.log
```

## Evaluation

Log into Tower as the **admin** user with a password of **redhat**, and verify that the configuration is restored. One indicator of success is that there should only be one Job Template, **Demo Job Template**, present after the restoration.

This concludes the lab.

## ▶ 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      | <b>anny@lab.example.com</b> |
| USERNAME   | <b>anny</b>                 |
| PASSWORD   | <b>redhat123</b>            |

- Add a second **Normal User** to the **Default** organization. Use the following information:

| Field      | Value                         |
|------------|-------------------------------|
| FIRST NAME | Robert                        |
| LAST NAME  | Farnham                       |
| EMAIL      | <b>robert@lab.example.com</b> |
| USERNAME   | <b>robert</b>                 |
| PASSWORD   | <b>redhat123</b>              |

- Within the **Default** organization, create a Team called **Devops**. Give the team a description of **Devops Team**.

- Add **anny** and **robert** as Members of the **Devops** Team.

## Evaluation

Use the Tower interface to verify that the users are correctly configured.



### Note

There is a grading script that you will run at the end of all the comprehensive review labs. That script evaluates your work in this lab.

This concludes the lab.

## ► 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      | <b>anny@lab.example.com</b> |
| USERNAME   | <b>anny</b>                 |
| PASSWORD   | <b>redhat123</b>            |

- Add a second **Normal User** to the **Default** organization. Use the following information:

| Field      | Value                         |
|------------|-------------------------------|
| FIRST NAME | Robert                        |
| LAST NAME  | Farnham                       |
| EMAIL      | <b>robert@lab.example.com</b> |
| USERNAME   | <b>robert</b>                 |
| PASSWORD   | <b>redhat123</b>              |

- Within the **Default** organization, create a Team called **Devops**. Give the team a description of **Devops Team**.

- Add **anny** and **robert** as Members of the **Devops** Team.

1. Log in to the Tower web interface as the **admin** user with a password of **redhat**, 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            | <b>anny@lab.example.com</b> |
| USERNAME         | <b>anny</b>                 |
| ORGANIZATION     | <b>Default</b>              |
| PASSWORD         | <b>redhat123</b>            |
| CONFIRM PASSWORD | <b>redhat123</b>            |
| USER TYPE        | <b>Normal User</b>          |

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            | <b>robert@lab.example.com</b> |
| USERNAME         | <b>robert</b>                 |
| ORGANIZATION     | <b>Default</b>                |
| PASSWORD         | <b>redhat123</b>              |
| CONFIRM PASSWORD | <b>redhat123</b>              |
| USER TYPE        | <b>Normal User</b>            |

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.
  - 3.6. Click **SAVE** to save the changes.
  - 3.7. Verify on the next screen that **anny** and **robert** are displayed in the list as members of the team.

## Evaluation

Use the Tower interface to verify that the users are correctly configured.



### Note

There is a grading script that you will run at the end of all the comprehensive review labs. That script evaluates your work in this lab.

This concludes the lab.

## ▶ 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 a custom inventory script to Tower with a name of **ldap-freeipa.py**. The Python script can be downloaded from <http://materials.example.com/classroom/ansible/ipa-setup/ldap-freeipa.py>. Using the **Default** organization, give the inventory script a description of **Dynamic Inventory for IdM Server**.
- Create a new Inventory called **Dynamic Inventory**. Using **Default** organization, give the inventory a description of **Dynamic Inventory from IdM Server**. For **Source**, choose the **ldap-freeipa.py** custom inventory script. Also configure the **OVERWRITE** update option for the inventory.
- Update the dynamic inventory. When finished, review every group synchronized from the IdM Server to ensure that all groups contain hosts.
- Create a new machine credential called **Devops**. Use the following information:

| Field                         | Value             |
|-------------------------------|-------------------|
| NAME                          | <b>Devops</b>     |
| DESCRIPTION                   | Devops Credential |
| ORGANIZATION                  | <b>Default</b>    |
| CREDENTIAL TYPE               | Machine           |
| USERNAME                      | <b>devops</b>     |
| PASSWORD                      | <b>redhat</b>     |
| PRIVILEGE ESCALATION METHOD   | <b>sudo</b>       |
| PRIVILEGE ESCALATION USERNAME | <b>root</b>       |

| Field                         | Value         |
|-------------------------------|---------------|
| PRIVILEGE ESCALATION PASSWORD | <b>redhat</b> |

- Grant the **Admin** role on the **Devops** Credential to the **Devops** team.

## Evaluation

Use the Tower interface to verify that the dynamic inventory script is correctly configured.



### Note

There is a grading script that you will run at the end of all the comprehensive review labs. That script evaluates your work in this lab.

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 a custom inventory script to Tower with a name of **ldap-freeipa.py**. The Python script can be downloaded from <http://materials.example.com/classroom/ansible/ipa-setup/ldap-freeipa.py>. Using the **Default** organization, give the inventory script a description of **Dynamic Inventory for IdM Server**.
- Create a new Inventory called **Dynamic Inventory**. Using **Default** organization, give the inventory a description of **Dynamic Inventory from IdM Server**. For **Source**, choose the **ldap-freeipa.py** custom inventory script. Also configure the **OVERWRITE** update option for the inventory.
- Update the dynamic inventory. When finished, review every group synchronized from the IdM Server to ensure that all groups contain hosts.
- Create a new machine credential called **Devops**. Use the following information:

| Field                         | Value             |
|-------------------------------|-------------------|
| NAME                          | <b>Devops</b>     |
| DESCRIPTION                   | Devops Credential |
| ORGANIZATION                  | <b>Default</b>    |
| CREDENTIAL TYPE               | Machine           |
| USERNAME                      | <b>devops</b>     |
| PASSWORD                      | <b>redhat</b>     |
| PRIVILEGE ESCALATION METHOD   | <b>sudo</b>       |
| PRIVILEGE ESCALATION USERNAME | <b>root</b>       |

| Field                         | Value         |
|-------------------------------|---------------|
| PRIVILEGE ESCALATION PASSWORD | <b>redhat</b> |

- 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 the **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 IdM Server |
| ORGANIZATION | <b>Default</b>                   |

- 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 IdM server |
| ORGANIZATION | <b>Default</b>                    |

- 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, select 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 then click the **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                          | <b>Devops</b>     |
| DESCRIPTION                   | Devops Credential |
| ORGANIZATION                  | <b>Default</b>    |
| CREDENTIAL TYPE               | Machine           |
| USERNAME                      | <b>devops</b>     |
| PASSWORD                      | <b>redhat</b>     |
| PRIVILEGE ESCALATION METHOD   | <b>sudo</b>       |
| PRIVILEGE ESCALATION USERNAME | <b>root</b>       |
| PRIVILEGE ESCALATION PASSWORD | <b>redhat</b>     |

- 5.4. Leave the other fields untouched and click **SAVE** to create the new Credential.
- 6.** To grant the Admin role to the 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. The team displays in the second section, beneath the first one.
- 6.7. In the second section, 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 that the dynamic inventory script is correctly configured.



### Note

There is a grading script that you will run at the end of all the comprehensive review labs. That script evaluates your work in this lab.

This concludes the comprehensive review.

## ▶ 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 to download the Ansible materials for the Project that the new Job Templates will use, based on the following information:

| Field           | Value                                                                                                                         |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------|
| NAME            | <b>student-git</b>                                                                                                            |
| DESCRIPTION     | Student Git Credential                                                                                                        |
| ORGANIZATION    | <b>Default</b>                                                                                                                |
| CREDENTIAL TYPE | Source Control                                                                                                                |
| USERNAME        | <b>git</b>                                                                                                                    |
| SCM PRIVATE KEY | Copy the contents of the <b>/home/student/.ssh/lab_rsa</b> private key file on the <b>workstation</b> machine into this field |

- Create a new Project based on the following information:

| Field       | Value                 |
|-------------|-----------------------|
| NAME        | My Full-Stack Project |
| DESCRIPTION | Full Stack Project    |

| Field          | Value                                                                                                |
|----------------|------------------------------------------------------------------------------------------------------|
| ORGANIZATION   | <b>Default</b>                                                                                       |
| SCM TYPE       | Git                                                                                                  |
| SCM URL        | <code>ssh://git.lab.example.com/var/opt/gitlab/git-data/repositories/git/full-stack-setup.git</code> |
| SCM CREDENTIAL | <b>student-git</b>                                                                                   |

- 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  | <b>Devops</b>             |

- 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  | <b>Devops</b>              |

- Create a new Job Template called **Set up Load Balancer** with the following configuration:

| Field | Value                |
|-------|----------------------|
| NAME  | Set up Load Balancer |

| Field       | Value                     |
|-------------|---------------------------|
| DESCRIPTION | Set up all load balancers |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <code>lb-setup.yml</code> |
| 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 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 lab.

There is a grading script that you will run at the end of the all comprehensive review labs. That script evaluates your work in this lab.

This concludes the lab.

## ► 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 to download the Ansible materials for the Project that the new Job Templates will use, based on the following information:

| Field           | Value                                                                                                                         |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------|
| NAME            | <b>student-git</b>                                                                                                            |
| DESCRIPTION     | Student Git Credential                                                                                                        |
| ORGANIZATION    | <b>Default</b>                                                                                                                |
| CREDENTIAL TYPE | Source Control                                                                                                                |
| USERNAME        | <b>git</b>                                                                                                                    |
| SCM PRIVATE KEY | Copy the contents of the <b>/home/student/.ssh/lab_rsa</b> private key file on the <b>workstation</b> machine into this field |

- Create a new Project based on the following information:

| Field       | Value                 |
|-------------|-----------------------|
| NAME        | My Full-Stack Project |
| DESCRIPTION | Full Stack Project    |

| Field          | Value                                                                                   |
|----------------|-----------------------------------------------------------------------------------------|
| ORGANIZATION   | <b>Default</b>                                                                          |
| SCM TYPE       | Git                                                                                     |
| SCM URL        | ssh://git.lab.example.com/var/opt/gitlab/git-data/repositories/git/full-stack-setup.git |
| SCM CREDENTIAL | <b>student-git</b>                                                                      |

- 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  | <b>Devops</b>             |

- 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  | <b>Devops</b>              |

- Create a new Job Template called **Set up Load Balancer** with the following configuration:

| Field | Value                |
|-------|----------------------|
| NAME  | Set up Load Balancer |

| Field       | Value                     |
|-------------|---------------------------|
| DESCRIPTION | Set up all load balancers |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <code>lb-setup.yml</code> |
| CREDENTIAL  | <b>Devops</b>             |

- 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            | <b>student-git</b>                                                                                                |
| DESCRIPTION     | Student Git Credential                                                                                            |
| ORGANIZATION    | <b>Default</b>                                                                                                    |
| CREDENTIAL TYPE | Source Control                                                                                                    |
| USERNAME        | <b>git</b>                                                                                                        |
| SCM PRIVATE KEY | Copy the contents of the <b>/home/student/.ssh/lab_rsa</b> private key file on <b>workstation</b> 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                                                                                     |
| SCM URL        | ssh://git.lab.example.com/var/opt/gitlab/git-data/repositories/git/full-stack-setup.git |
| SCM CREDENTIAL | <b>student-git</b>                                                                      |

- 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. Click **My Full-Stack Project** 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, beneath the first one.
  - 4.7. In the second section, 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          |
| DESCRIPTION | Set up all databases      |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <b>database-setup.yml</b> |
| CREDENTIAL  | <b>Devops</b>             |

- 5.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.
6. Grant the **Devops** Team an **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, beneath 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    | <code>webserver-setup.yml</code> |
| CREDENTIAL  | <b>Devops</b>                    |

7.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.

**8.** Grant the **Devops** Team the **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 load balancers |
| JOB TYPE    | Run                       |
| INVENTORY   | Dynamic Inventory         |
| PROJECT     | My Full-Stack Project     |
| PLAYBOOK    | <code>lb-setup.yml</code> |
| CREDENTIAL  | <b>Devops</b>             |

9.5. Leave the other fields untouched and click **SAVE** to create the new Job Template.

#### 10. Grant the **Devops** Team the **Admin** role on the **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, beneath the first one.
- 10.5. In the second section, 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 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 lab.

There is a grading script that you will run at the end of the all comprehensive review labs. That script evaluates your work in this lab.

This concludes the lab.

## ▶ 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 | <b>environment_name</b>                                           |
| 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   | <b>Default</b>                                 |
| TYPE           | Email                                          |
| HOST           | <b>localhost</b>                               |
| RECIPIENT LIST | <b>student@localhost</b>                       |
| SENDER EMAIL   | <b>system@lab.example.com</b>                  |
| 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 that everything is configured as specified.



**Note**

The next activity in this chapter will instruct you to launch the Workflow Job Template.

There is a grading script that you will run at the end of the all comprehensive review labs. That script evaluates your work in this lab.

This concludes the lab.

## ► 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 | <b>environment_name</b>                                           |
| 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   | <b>Default</b>                                 |
| TYPE           | Email                                          |
| HOST           | <b>localhost</b>                               |
| RECIPIENT LIST | <b>student@localhost</b>                       |
| SENDER EMAIL   | <b>system@lab.example.com</b>                  |
| 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    | <b>Default</b>                         |
| EXTRA VARIABLES | <b>environment_name: "Development"</b> |

- 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 a list of actions to be performed in the right panel.
  - 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 action 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, 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, select **On Success** and click **SELECT**.
- 2.15. Click **SAVE** to save the Workflow Job Template.
- 3.** Grant **Devops** Team the **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, beneath the first one.
  - 3.5. In the second section, 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                                                                |
| 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, located at the top of the Survey editor window, is set to **ON**.

- 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   | <b>Default</b>                                 |
| TYPE           | Email                                          |
| HOST           | <b>localhost</b>                               |
| RECIPIENT LIST | <b>student@localhost</b>                       |
| SENDER EMAIL   | <b>system@lab.example.com</b>                  |
| 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 that everything is configured as specified.



### Note

The next activity in this chapter will instruct you to launch the Workflow Job Template.

There is a grading script that you will run at the end of the all comprehensive review labs. That script evaluates your work in this lab.

This concludes the lab.

## ▶ 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-cr3** script with the **grade** argument, to confirm success on these exercises. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab review-cr3 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 action opens the Survey that you just created and asks for your input.

- 1.4. Click **NEXT**, and then **LAUNCH** to launch the Workflow. This action redirects you to a detailed status page of the running Workflow.
- 1.5. Observe the running Jobs of the Workflow. 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 by 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-cr3** script with the **grade** argument, to confirm success on these exercises. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab review-cr3 grade
```

This concludes the comprehensive review.

