

# Configuration Management with Ansible and Terraform



## Working with Ansible Roles and Vault



# Learning Objectives

By the end of this lesson, you will be able to:

- Implement Ansible roles in a playbook to organize and reuse configurations
- Configure the Ansible roles directory structure to manage multiple roles efficiently
- Create a personalized collection of roles on Ansible Galaxy to standardize configurations across projects
- Install a collection from Automation Hub and apply it in a playbook for enhanced automation capabilities
- Create a secure playbook using Ansible Vault to manage sensitive information like passwords and keys



# Learning Objectives

By the end of this lesson, you will be able to:

- 🔗 Develop a robust playbook with advanced error handling and troubleshooting mechanisms to ensure reliability
- 🔗 Analyze the use of inventory variables to determine their impact on configuration flexibility
- 🔗 Evaluate the capabilities of Ansible Tower in automating and managing playbook executions at a large scale





## **Ansible Roles**

# What Is an Ansible Role?

It is a self-contained and portable unit of automation that groups related tasks and associated variables, files, handlers, and other assets in a known file structure.



It can create bundles of automation content that you can run in one or more plays, reuse across playbooks, and share with other users in collections.

# What Is an Ansible Role?

Key facts about Ansible roles:

01

They must be used within a playbook.

02

They are defined using YAML files with a predefined directory structure.

03

They are sets of tasks that configure a host to serve a certain purpose.

# What Is an Ansible Role?

04

They enhance code reusability by providing predefined configurations and tasks.

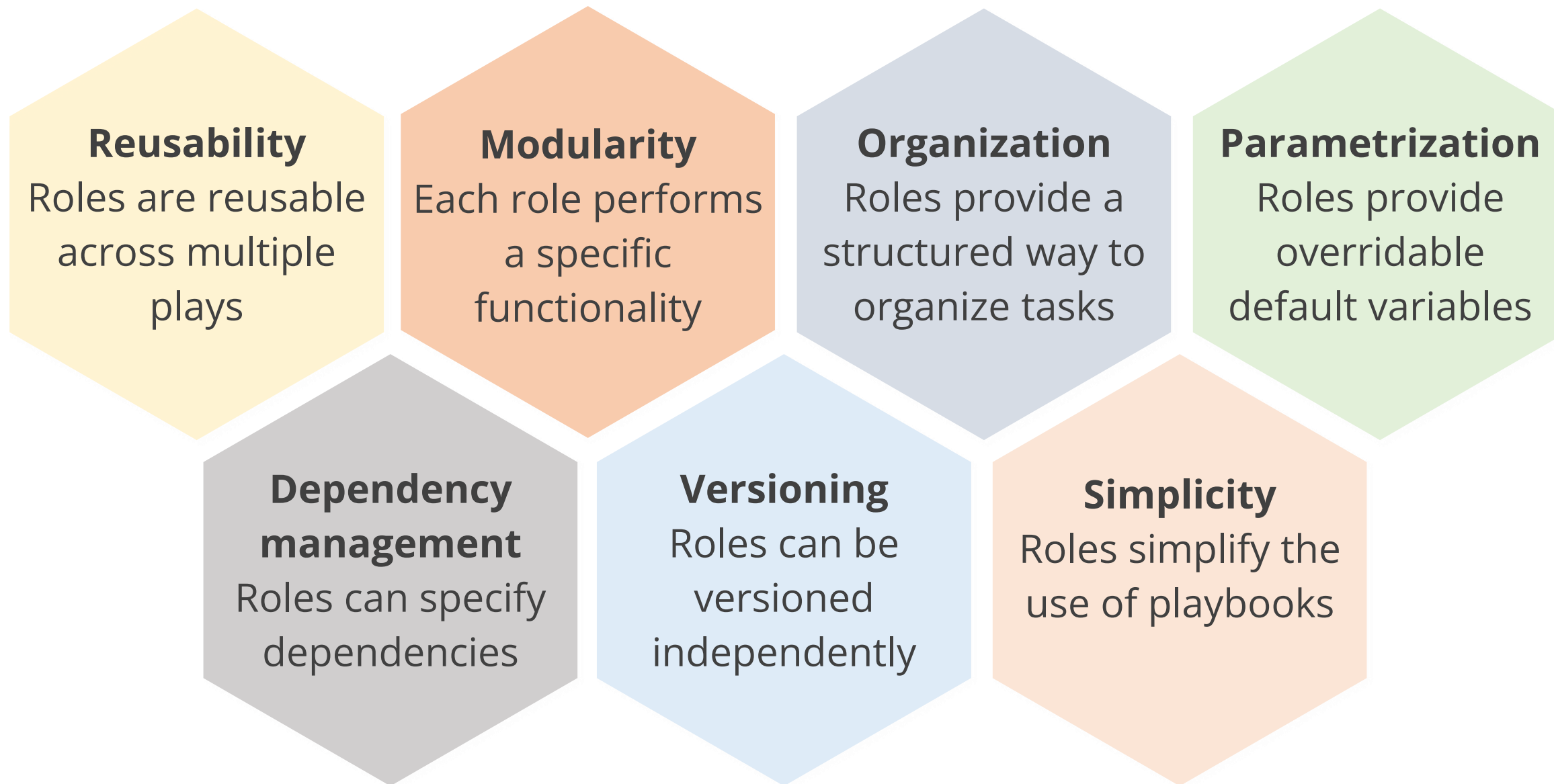
05

They can be easily modified due to their structured format.



# Why Use an Ansible Role?

Below are the key reasons for using Ansible roles with Ansible playbook:



# Ansible Roles: Use Cases

Ansible roles offer a way to group tasks, handlers, variables, files, templates, and modules to automate IT processes in a structured and reusable manner. Below are some of its use cases:

Server provisioning and configuration

Automate the setup of new servers to ensure consistency and reduce manual errors

Application deployment

Deploy applications across different environments (development, staging, production) with consistency

## Ansible Roles: Use Cases

Security and compliance

Ensure that systems comply with security policies and standards

Patch management

Automate the process of applying patches and updates to systems to maintain security and performance

Disaster recovery and backup

Automate backup and recovery processes to ensure data integrity and availability in case of a disaster

## Quick Check



You are working on an automation project and need to organize your Ansible playbooks for better reusability and maintainability. Which of the following actions best demonstrates the use of an Ansible role in this scenario?

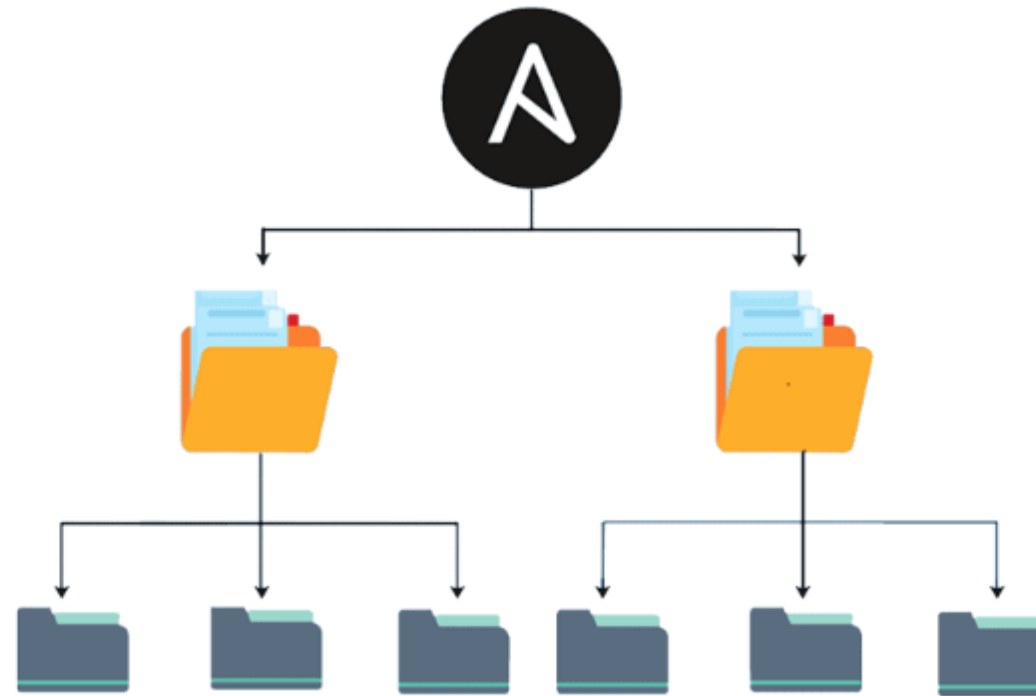
- A. Writing a single, large playbook that includes all tasks and variables
- B. Creating a custom module to handle specific tasks within the playbook
- C. Grouping related tasks, variables, and handlers into a structured directory and referencing them in your playbooks
- D. Using Ansible Tower to manage and execute your playbooks



# **Ansible Role Directory**

# What Is an Ansible Role Directory?

An Ansible role directory is a predefined directory structure that helps organize various components of a role, such as tasks, handlers, variables, templates, and files.



Each role must include at least one of the seven standard directories, and unused directories can be omitted.

# Ansible Role Directory Structure

The following are the seven standard directories and files:

**tasks/main.yml**

Lists tasks that the role provides to the play for execution

**handlers/main.yml**

Imports handlers into the parent play for use by the role or other roles and tasks in the play

**defaults/main.yml**

Contains default variables for the role which can be overridden

**vars/main.yml**

Stores variables that have higher precedence than those in **defaults/main.yml**

# Ansible Role Directory Structure

**files/stuff.txt**

**meta/main.yml**

**templates/something.j2**

Contains one or more files that are available for the role and its children

Contains metadata for the role, including role dependencies on other roles

Stores Jinja2 templates that can be used to generate configuration files dynamically

By default, Ansible searches for a **main.yml** file in most of the role directories to find the relevant content.



# Ansible Role Directory Structure

## Example:

```
roles/
  common/                # this hierarchy represents a "role"
    tasks/               #
      main.yml           #  <-- tasks file can include smaller files if
warranted
  handlers/              #
    main.yml             #  <-- handlers file
  templates/             #  <-- files for use with the template resource
    ntp.conf.j2          #  <----- templates end in .j2
  files/                 #
    bar.txt              #  <-- files for use with the copy resource
    foo.sh               #  <-- script files for use with the script resource
  vars/                  #
    main.yml             #  <-- variables associated with this role
  defaults/              #
    main.yml             #  <-- default lower priority variables for this role
  meta/                  #
    main.yml             #  <-- role dependencies
  library/               # roles can also include custom modules
  module_utils/          # roles can also include custom module_utils
  lookup_plugins/        # or other types of plugins, like lookup in this case
```

## Quick Check

You are tasked with creating an Ansible role for setting up a web server. Which of the following directory structures correctly represents the seven standard directories in an Ansible role?

- A. bin, etc, lib, opt, sbin, usr, var
- B. handlers, tasks, templates, files, vars, defaults, meta
- C. scripts, configs, logs, data, vars, handlers, meta
- D. docs, src, build, tests, configs, roles, tasks





## **Creating an Ansible Role**

# Creating an Ansible Role

The following command will create a new role in the current directory:

## Syntax

```
ansible-galaxy init role_name
```

To create a role within a specified directory, use the following syntax:

## Syntax

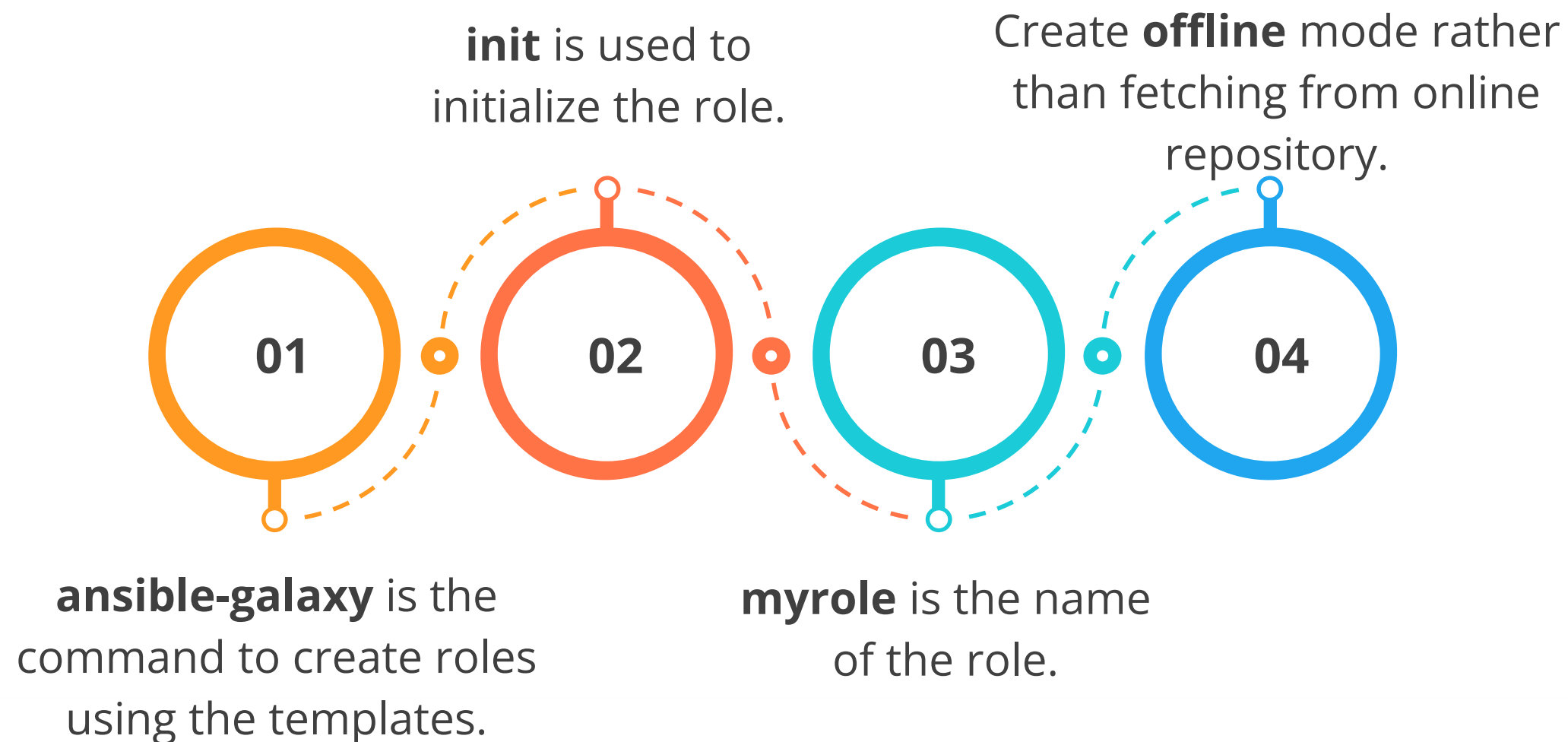
```
ansible-galaxy init directory_path/role_name
```

This command will build a directory that follows the standard role directory structure. Once the role directory is created, you can define tasks, default variables, and other components of an Ansible role.

## Example: Creating an Ansible Role

```
ansible-galaxy init /etc/ansible/roles/myrole --offline
```

The following steps occur when the above command is executed for creating a role:



# Tasks in an Ansible Role

A task in an Ansible role is a discrete action that Ansible performs on a managed node. Below are the steps to specify tasks within an Ansible role:

Navigate to the role directory.  
For example, **cd**  
**/etc/ansible/roles/myrole**

Locate the **tasks** directory in  
which there will be a file  
named **main.yml** to define  
the tasks

Edit the **main.yml** file to  
define tasks.  
For example, **nano**  
**tasks/main.yml**

# Tasks in an Ansible Role

Here is an example **tasks/main.yml** file for installing and starting the Apache web server:

```
---
# tasks file for myrole

- name: Install Apache
  apt:
    name: apache2
    state: present
  become: yes

- name: Start Apache service
  service:
    name: apache2
    state: started
    enabled: yes
  become: yes
```

# Storing and Finding a Role

Ansible roles are stored in several default locations. These include:

- The **roles/** directory relative to the playbook file
- The configured **roles\_path**
- The default search path, including **~/.ansible/roles**, **/usr/share/ansible/roles** and **/etc/ansible/roles**
- The directory where the playbook file is located

If the role is stored in a different location, the role path option must be set so that Ansible can find the roles.



# Using Ansible Roles

The following are the ways to use an Ansible role:

01

At the **play level** with the roles option

02

At the tasks level with **include\_role**

03

At the tasks level with **import\_role**

04

As a dependency on another role

# Using Ansible Roles at Play Level

Ansible roles are used in a play using the **roles** option:

## Example:

```
---  
- hosts: webservers  
  roles:  
    - common  
    - webservers
```

This code snippet demonstrates how to apply multiple roles (**common** and **webservers**) to the hosts specified in the **webservers** group.

# Using Ansible Roles at Play Level

When the **roles** option is used at a play level, each role **x** looks for a **main.yml** (also **main.yaml** and **main**) in the following directories:

If <b>roles/x/tasks/main.yml</b> exists	Ansible adds the tasks in that file to the play
If <b>roles/x/handlers/main.yml</b> exists	Ansible adds the handlers in that file to the play
If <b>roles/x/vars/main.yml</b> exists	Ansible adds the variables in that file to the play
If <b>roles/x/defaults/main.yml</b> exists	Ansible adds the variables in that file to the play
If <b>roles/x/meta/main.yml</b> exists	Ansible adds any role dependencies in that file to the list of roles

Additionally, any copy, script, template, or include tasks (in the role) can reference files in **roles/x/{files, templates, tasks}/** without having to path them relatively or absolutely.

# Using Ansible Roles at Play Level

Ansible executes a play using the **roles** option at the play level in the following sequence:

01

Executes any **pre\_tasks** defined in the play

02

Runs any handlers triggered by **pre\_tasks**

03

Processes each role listed in the **roles:** section, in the specified order

## Using Ansible Roles at Play Level

04

Executes any tasks defined in the play

05

Runs any handlers triggered by the roles or tasks

06

Executes any **post\_tasks** defined in the play

07

Runs any handlers triggered by **post\_tasks**

# Using Ansible Roles Dynamically

Users can reuse roles dynamically anywhere in the **tasks** section of a play using **include\_role**.

01

The included roles run in the order they are defined.

02

If there are other tasks before an **include\_role** task, the other tasks will run first.

## Example: Using Ansible Roles Dynamically

```
---  
  
- hosts: webservers  
  tasks:  
    - name: Print a message  
      ansible.builtin.debug:  
        msg: "this task runs before the example role"  
  
    - name: Include the example role  
      include_role:  
        name: example  
  
    - name: Print a message  
      ansible.builtin.debug:  
        msg: "this task runs after the example role"
```

## Example: Using Ansible Roles Dynamically

The following shows that keywords, such as variables and tags, can be included while using roles in a play:

```
---
- hosts: webservers
  tasks:
    - name: Include the foo_app_instance role
      include_role:
        name: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
      tags: typeA
  ...
```

Ansible applies the tag only to the **include\_role** section which allows the execution of the selected tasks from the role.



## Example: Using Ansible Roles Dynamically

The below example shows how to add Ansible roles dynamically using conditions:

```
---
- hosts: webservers
  tasks:
    - name: Include the some_role role
      include_role:
        name: some_role
      when: "ansible_facts['os_family'] == 'RedHat'"
```

# Why Use Dynamic Ansible Roles

Below are the advantages of using dynamic Ansible roles using **include\_role**:



Flexibility

Flexible and  
conditional execution  
of tasks



Scalability

Easily scale  
automation by  
reusing roles



Version  
control

Management of  
different versions of  
roles

This approach enhances the modularity and maintainability of the automation scripts.

# Using Static Ansible Roles

Users can reuse roles statically anywhere in the tasks section of a play using **import\_role**.

01

Roles are parsed and included during playbook parsing.

02

The tasks from the imported roles are executed in the defined sequence.

## Note:

When a tag is added to an **import\_role** statement, Ansible applies the tag to all tasks within the role.

## Example: Using Ansible Roles Staticly

```
---

- hosts: webservers
  tasks:
    - name: Print a message
      ansible.builtin.debug:
        msg: "before we run our role"

    - name: Import the example role
      import_role:
        name: example

    - name: Print a message
      ansible.builtin.debug:
        msg: "after we ran our role"
```

# Why Use Static Ansible Roles?

Below are the advantages of using static Ansible roles using **import\_role**:

Portability

Ensures uniform execution in different environments

Isolation

Ensures isolated execution of roles in a predictable order

Version control

Allows for better version control

Performance

Reduces execution time and increases efficiency

This approach enhances the consistency and reliability of the automation scripts.

# Multiple Execution of a Role

## Running a role multiple times in one playbook

Ansible executes each role once, even if it is defined multiple times.

Roles are executed multiple times only if the parameters defined on the role are different for each definition.

Ansible only runs the role **xyz** once in this play.

### Example

```
---  
- hosts: webservers  
  roles:  
    - xyz  
    - abc  
    - xyz
```

# Multiple Execution of a Role

There are two options to force Ansible to run a role more than once:

01

Passing different parameters

02

Using the **allow\_duplicates: true**  
parameter

# Multiple Execution of a Role

Difference between the two options to force multiple Ansible role execution:

Passing different parameters	Allowing duplicates
Execute the role multiple times by passing different sets of parameters	Execute the role multiple times with the same set of parameters by explicitly allowing duplicates
Perform different tasks or configurations based on parameters	Re-apply the same role without changing parameters
Include or import the role multiple times, each with different variables	Include or import the role multiple times with <b>allow_duplicates: true</b>



## Example: Passing Different Parameters

This playbook runs the **xyz** role twice:

### Example:

```
---  
- hosts: webservers  
  roles:  
    role: xyz  
    message: "first"  
  
    role: xyz  
    message: "second"
```

Ansible runs **xyz** twice because each role definition has different parameters.

## Example: Allow Duplicates

The following configuration executes the same role multiple times by explicitly enabling duplicates:

### Example

```
# playbook.yml
---
- hosts: webservers
  roles:
    - foo
    - foo
# roles/foo/meta/main.yml
---
allow_duplicates: true
```

Using **allow\_duplicates: true** ensures that specific roles are re-applied as needed, providing flexibility and control in complex playbook executions.

## Assisted Practice



### Creating Ansible roles

Duration: 20 Min.

#### Problem statement:

You have been assigned a task to create and configure Ansible roles for managing configurations.

#### Outcome:

By the end of this demo, you will be able to write Ansible roles to organize playbooks into reusable, modular components, making it easier to manage complex configurations by encapsulating related tasks, variables, files, templates, and handlers into structured directories.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Initialize the roles
2. Create and define tasks
3. Organize task execution
4. Create the index.html file
5. Verify the configuration
6. Execute the playbook
7. Check the index.html file in the browser

## Quick Check



You need to reuse an Ansible role called **webserver** in your playbook, ensuring that it runs multiple times with different configurations. Which approach allows you to achieve this dynamically at the task level?

- A. Including the **webserver** role in the roles section of the playbook with default configurations
- B. Using the **include\_role** module within tasks and passing different variables for each execution
- C. Defining multiple **webserver** tasks statically in the tasks section with different variables
- D. Creating multiple identical playbooks each specifying the **webserver** role with different configurations



# **Ansible Galaxy**

# What Is Ansible Galaxy?

It is a community hub for finding, sharing, and downloading Ansible content, including roles and collections. It provides pre-packaged units of work that help streamline and enhance automation projects.



Users can search, install, and manage roles and collections using the **ansible-galaxy** command-line tool.

# Why Use Ansible Galaxy?

Ansible Galaxy serves several key purposes when working with Ansible roles, enhancing the efficiency and effectiveness of automation tasks:



It provides a vast repository of pre-built roles contributed by the community.



It provides roles that are well-documented and follow best practices, ensuring standardization.



It allows users to share their roles with the community, making it easier to collaborate.



It provides roles which makes the playbook modular and manageable.



# Common Ansible Galaxy Commands

The following are some of the most common Ansible Galaxy commands:

**`ansible-galaxy list`**

To display the list of installed roles with version numbers

**`ansible-galaxy remove [role]`**

To remove an installed role

**`ansible-galaxy init`**

To create a role template suitable for submission to Ansible Galaxy

# Galaxy: Collection

Collections are a distribution format for Ansible content that includes roles, modules, plugins, and other resources.



It can be installed and utilized through a distribution server, such as Ansible Galaxy or a Pulp 3 Galaxy server.

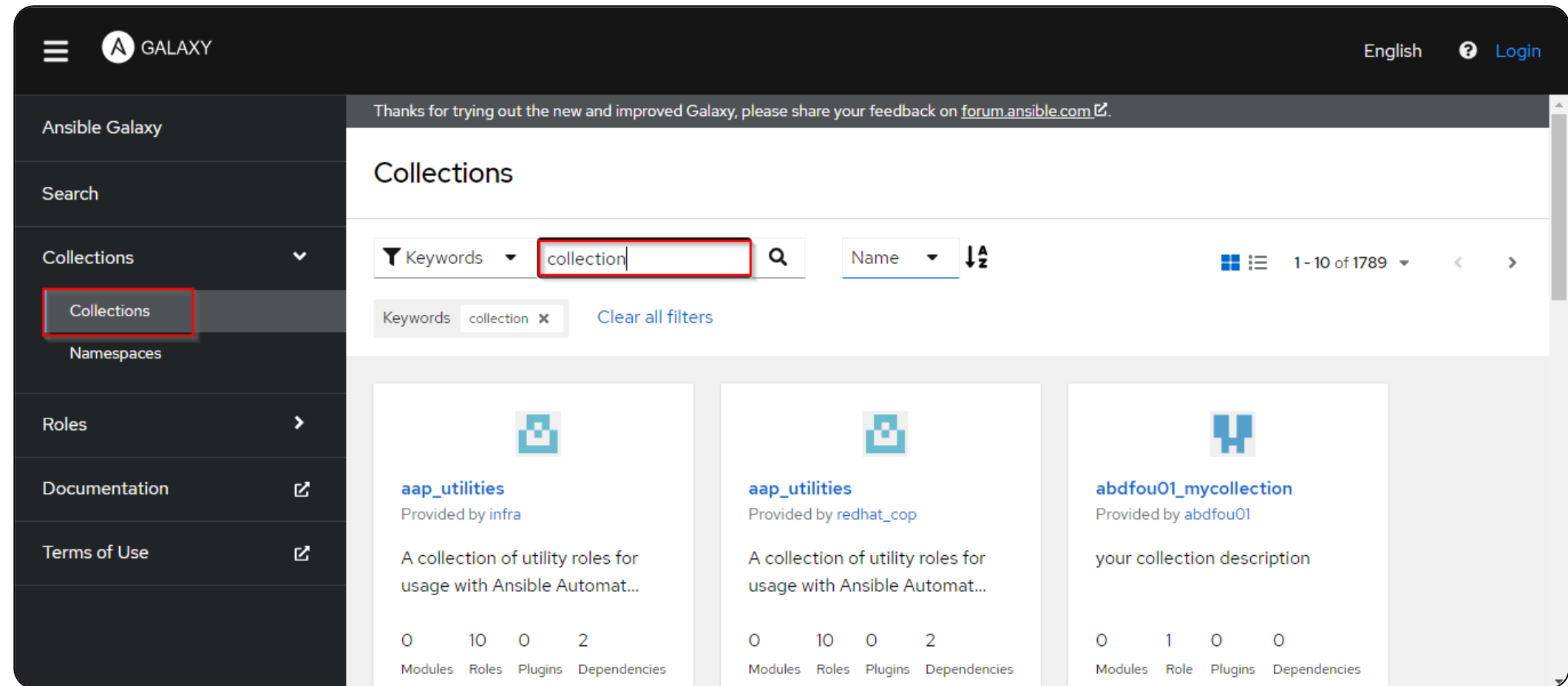
# Galaxy: Collection

To find collections on the Ansible Galaxy console:

Click on the **Search** icon in the left-hand navigation

Set the filter to the **collection**

Set the other filters and press **Enter**



# Galaxy: Collection Structure

A collection structure refers to the organized layout of directories and files that make up an Ansible collection.

01

A collection requires a **galaxy.yml** file at the root level of the collection.

02

This file contains all the metadata that Galaxy and other tools need to package, build, and publish the collection.

# Galaxy: Collection Structure

A collection follows the following simple structure of directories and files:

```
1. collection/
2. |— docs/
3. |— galaxy.yml
4. |— plugins/
5. | |— modules/
6. | |   └─ module1.py
7. | |— inventory/
8. |   └─ .../
9. |— README.md
10. |— roles/
11. | |— role1/
12. | |— role2/
13. |   └─ .../
14. |— playbooks/
15. | |— files/
16. | |— vars/
17. | |— templates/
18. |   └─ tasks/
19. └─ tests/
```

# Installing a Collection from Galaxy

The following command is used to install a collection from Ansible Galaxy using the **ansible-galaxy** command-line tool:

```
ansible-galaxy collection install collection_name
```

The command by default uses **https://galaxy.ansible.com** as the Galaxy server.

To upgrade a collection to the latest available version from the Galaxy server, users can use the **--upgrade** option after the collection name.

# Installing a Collection from Automation Hub

Ansible automation hub is an enterprise-focused repository that provides Ansible collections, including bundles of modules, plugins, roles, and documentation from Red Hat and other partners.

Steps to download collection from Automation Hub:

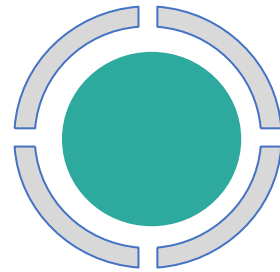
1. Get the Automation Hub API token from **<https://cloud.redhat.com/ansible/automation-hub/token/>**
2. Configure the Red Hat Automation Hub server in the **server\_list** option under the **[galaxy]** section in **ansible.cfg** file

Automation Hub content is available to subscribers only, so users must download an API token and configure their local environment before downloading collections.

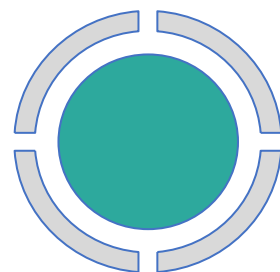
# Installing a Collection from GitHub

Collections can also be installed from a Git repository instead of from Galaxy or Automation Hub.

Installing collections from a Git repository allows developers to review the collection before they build and publish it.



The Git repository must contain a **galaxy.yml** or **MANIFEST.json** file.



This file provides metadata such as the version number and namespace of the collection.



## Example: Installing a Collection from GitHub

```
# Install a collection in a repository using the latest commit
on the branch 'devel'
ansible-galaxy collection install
git+https://github.com/organization/repo_name.git,devel

# Install a collection from a private GitHub repository
ansible-galaxy collection install
git@github.com:organization/repo_name.git

# Install a collection from a local git repository
ansible-galaxy collection install
git+file:///home/user/path/to/repo_name.git
```

These commands allow you to install Ansible collections from various Git repositories, making it flexible to source collections from both remote and local sources.

# Collection Range Identifiers

By default, **ansible-galaxy** installs the latest available version of the specified collections.



Version range identifiers are used to install a specific version of the collection.



Multiple range identifiers can be specified by separating each with a “,”.

## Example: Collection Range Identifiers

The following is an example of installing the most recent version that is greater than or equal to 1.0.0 and less than 2.0.0:

```
ansible-galaxy collection install 'my_namespace.my_collection:>=1.0.0,<2.0.0'
```

This approach ensures that you get a compatible version of the collection without manually checking for the latest suitable version.

# Collection Range Identifier: Version Constraints

The following operators are used to specify the version constraints:

\*

The most recent version

!=

Not equal to the version specified

==

Exactly the version specified

>=

Greater than or equal to the version specified

>

Greater than the version specified

<=

Less than or equal to the version specified

<

Less than the version specified

# Galaxy: Roles

Roles in Ansible Galaxy are community contributed or privately developed collections of playbooks, templates, and other resources that can be easily shared and reused.



Each role typically includes a README file that details its usage and variables.



Ansible Galaxy contains a variety of roles that are continuously evolving and expanding.

# Galaxy: Roles

Galaxy can use Git to add other role sources like GitHub.

By default, roles are placed into the directory:  
**/etc/ansible/roles.**

## Note

Roles must be downloaded before using them in a playbook.

# Install a Role from Galaxy

The **ansible-galaxy** command can be used to install roles from Galaxy or directly from a Git-based source control management (SCM).

## Example:

```
ansible-galaxy install namespace.role_name
```

The command by default uses **<https://galaxy.ansible.com>** as the Galaxy server.

## Example: How to Install a Role

To install a specific version of a role from Galaxy, add a comma and then specify the value of a GitHub release tag.

For example:

```
$ ansible-galaxy install mywebserver.apache,1.0.0
```

It is also possible to point directly to the Git repository and specify a branch name or commit hash as the version.

```
$ ansible-galaxy install  
git+<Github_Repository_URL>,0b7cd353c0250e87a26e0499e59e
```



# Role Attributes

Each role in the file will have one or more of the following attributes:

**src**

Specify the source of the role

**scm**

Specify the SCM

**version**

Specify the role version

**name**

Download the role to a specific name

# Installing Multiple Roles

Multiple roles can be installed by including the roles in a **requirements.yml** file.

The following command installs roles included in the **requirements.yml**:

```
$ ansible-galaxy install -r requirements.yml
```

## Note

Roles and collections can be installed from the same requirements file.

# Role Dependencies

Roles can also depend on other roles, and when a user installs a role that has dependencies, those dependencies will automatically be installed to the **roles\_path**.

There are two ways to define the dependencies of a role:



Using **meta/requirements.yml**



Using **meta/main.yml**

# Role Dependencies



Users can create the file **meta/requirements.yml** and define dependencies in the same format as used for **requirements.yml**.  
A user can import the specified roles in their tasks from there.



Users can specify role dependencies in the **meta/main.yml** file by providing a list of roles under the dependencies section.

## Quick Check



You need to install multiple roles and collections in your Ansible project, ensuring that all dependencies are managed automatically. Which file format and command should you use to achieve this?

- A. Create a **requirements.txt** file and use **ansible-galaxy install -r requirements.txt**
- B. Create a **roles.yml** file and use **ansible-galaxy install -r roles.yml**
- C. Create a **requirements.yml** file and use **ansible-galaxy install -r requirements.yml**
- D. Create a **collections.yml** file and use **ansible-galaxy install -r collections.yml**



# **Ansible Vault**

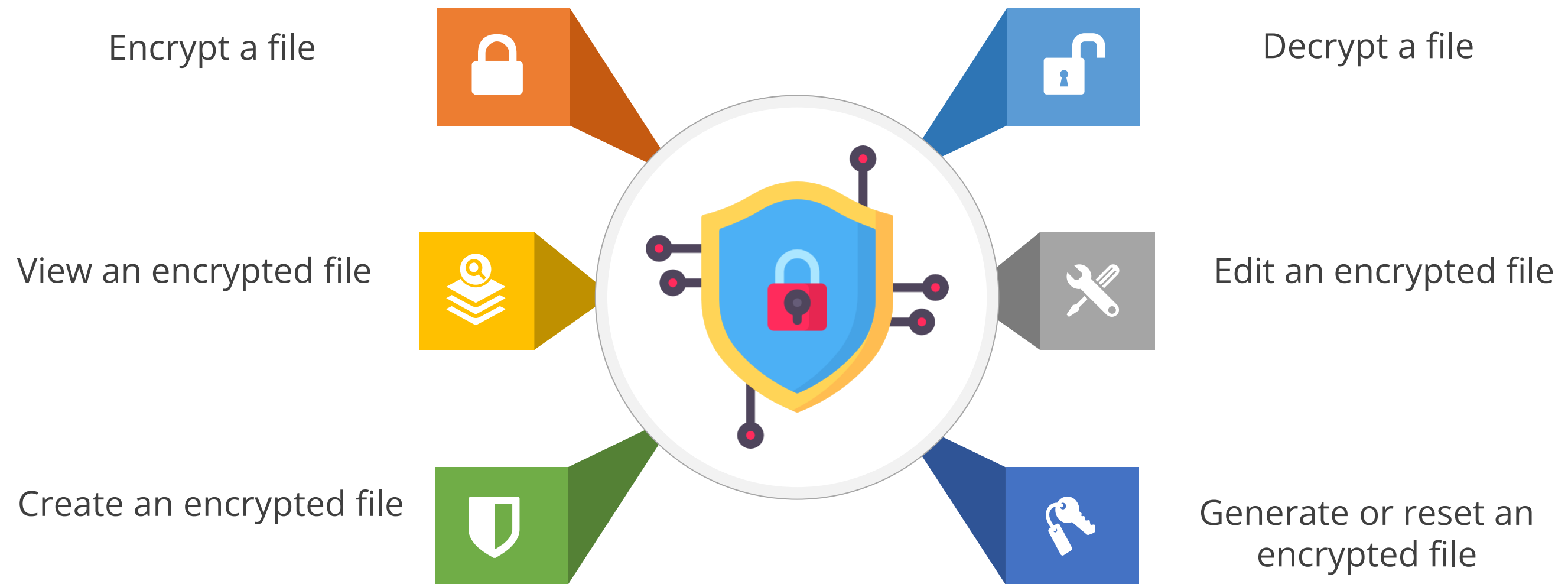
# What Is Ansible Vault?

It is a feature of Ansible that provides encryption for sensitive data, such as passwords, keys, and other confidential information.



It integrates with third-party key management services like Amazon AWS KMS, enabling secure cloud management and storage of encrypted secrets.

# Operations Performed by Ansible Vault





# Encryption Levels in Ansible

Ansible vault provide following levels of encryption:

## File-level encryption

Ansible Vault can encrypt any structured data file used by Ansible, including inventory variables, role variables, task files, handlers, and arbitrary file

## Variable-level encryption

Ansible Vault can encrypt single values inside a YAML file using the **!vault** tag to indicate special processing for those values.

# Encryption with Ansible Vault

Below are the steps to encrypt or decrypt content with Ansible Vault:

01

Access passwords stored in a third-party tool using a script

02

Encrypt or decrypt content using the `ansible-vault` command-line tool with passwords

03

Store encrypted content safely in source control and share it securely

# Encryption with Ansible Vault

Encrypted variables and files can be used in ad hoc commands and playbooks by providing the relevant passwords used during encryption.



**ansible.cfg**

Users have the option to adjust the **ansible.cfg** file to specify the location of a password file or to set it to always prompt for the password.

# Encryption with Ansible Vault

Users can encrypt variables and files with Ansible Vault using:

**!vault**

This tag tells Ansible and YAML that the content needs to be decrypted

|

This character allows multi-line strings

**--vault-id**

This label specifies which vault ID decrypts the content

# Encrypting Individual Variables with Ansible Vault

The **ansible-vault encrypt\_string** command used to create a basic encrypted variable.

The options passed in the command are:

01

A source for the vault password

02

The string to encrypt and the string name

## Syntax:

```
ansible-vault encrypt_string <password_source> '<string_to_encrypt>' --name  
'<string_name_of_variable>'
```

# Encrypting Individual Variables with Ansible Vault

The following command is used to encrypt the string **samplestring** using the password stored in **a\_password\_file**:

## Example:

```
ansible-vault encrypt_string --vault-password-file a_password_file  
'samplestring' --name 'the_secret'
```

## Output:

```
the_secret: !vault |  
    $ANSIBLE_VAULT;1.1;AES256  
    62313365396662343061393464336163383764373764613633653634306231386433626436623361  
    6134333665353966363534333632666535333761666131620a663537646436643839616531643561  
    63396265333966386166373632626539326166353965363262633030333630313338646335303630  
    3438626666666137650a353638643435666633633964366338633066623234616432373231333331  
    6564
```

# Encrypting Individual Variables with Ansible Vault

The user can view the original value of the encrypted variable in the debug module.

## Example:

```
ansible localhost -m ansible.builtin.debug -a var="the_secret" -e "@vars.yml" --vault-id dev@a_password_file
```

## Output:

```
localhost | SUCCESS => {  
  "the_secret": "samplestring"  
}
```

# Encrypting Files with Ansible Vault

It can encrypt any structured data file used by Ansible. The full file is encrypted in the vault.  
Here are some key types of files that Ansible Vault can encrypt:

Group variables files  
from inventory

Variables files in  
roles

Tasks and handler  
files

Variables files  
loaded by  
**include\_vars** or  
**vars\_files**



# Encrypting Files with Ansible Vault

Here are some key types of files that Ansible Vault can encrypt:

Host variable files  
from inventory

Default files in  
roles

Binary files or  
arbitrary files

Variables files  
passed to ansible-  
playbook with -e  
@file.yml or -e  
@file.json

# How to Encrypt Files with Ansible Vault

To create a new encrypted data file named **sample.yml** using the test vault password from **multi\_password\_file**, execute the following command:

```
ansible-vault create --vault-id test@multi_password_file sample.yml
```

Executing this command opens an editor to add content. Upon closing the editor, the file is saved as encrypted content. The file header shown below displays the vault ID used for its creation:

```
``$ANSIBLE_VAULT;1.2;AES256;test``
```

# How to Encrypt Files with Ansible Vault

The command to encrypt an existing file is **ansible-vault encrypt**. It can operate on multiple files as shown below:

```
ansible-vault encrypt foo.yml bar.yml baz.yml
```

The command to view the contents of an encrypted file without editing, is the **ansible-vault view**. It can display multiple files as shown below:

```
ansible-vault view foo.yml bar.yml baz.yml
```

# Ansible Vault Commands

The commonly used commands while encrypting files using Ansible Vault are:

**ansible-vault edit foo.yml**

Modifies an encrypted file in place without needing to manually decrypt and re-encrypt it

**ansible-vault rekey foo.yml**

Changes the encryption password of the specified file, enhancing security without altering the file content

**ansible-vault decrypt foo.yml**

Decrypts an encrypted file, making it accessible for editing or inspection without encryption

# Format of Files Encrypted with Ansible Vault

Ansible Vault creates Unicode Transformation Format (UTF-8) encoded txt files:

Example:

**\$ANSIBLE\_VAULT;1.1;AES256**  
or  
**\$ANSIBLE\_VAULT;1.X;AES256;vault-id-label**

The file format includes a header terminated by a newline. This header comprises up to four elements separated by semicolons.

# Format of Files Encrypted with Ansible Vault: Header Components

The four header components are:

**\$ANSIBLE\_VAULT**

It is the format ID.

**1.X**

1.X is the vault format version.

**AES256**

It is the cipher algorithm used to encrypt the data.

**vault-id-label**

It is the vault ID label encrypts the data (optional).

# Format of Files Encrypted with Ansible Vault: Example

Except for the header, the remaining content of the file consists of **vault text**, which is a fortified version of the encrypted ciphertext.

Header

```
$ANSIBLE_VAULT;1.1;AES256
```

```
62313365396662343061393464336163383764373764613633653634306231386433626436623361  
6134333665353966363534333632666535333761666131620a663537646436643839616531643561  
63396265333966386166373632626539326166353965363262633030333630313338646335303630  
3438626666666137650a353638643435666633633964366338633066623234616432373231333331  
6564
```

Vault text

# Assisted Practice



## Implementing Ansible Vault

Duration: 20 Min.

### Problem statement:

You have been assigned a task to securely manage sensitive files using Ansible Vault, ensuring encryption and protection of sensitive data.

### Outcome:

By the end of this demo, you will be able to implement Ansible Vault to securely store sensitive data such as passwords, API keys, and other confidential information within Ansible playbooks.

**Note:** Refer to the demo document for detailed steps



# Assisted Practice: Guidelines



Steps to be followed:

1. Create and edit a vault
2. Reset the password for the vault
3. Execute the YAML file

## Quick Check

You are tasked with securely storing sensitive information, such as API keys and passwords, within your Ansible playbooks. You decide to use Ansible Vault to encrypt this data. Which Ansible Vault command is used to encrypt a file containing sensitive information?

- A. `ansible-vault lock`
- B. `ansible-vault hide`
- C. `ansible-vault encrypt`
- D. `ansible-vault secure`





## **Error Handling and Troubleshooting**

# Error Handling in Ansible Using Blocks

Users can manage how Ansible handles task errors by utilizing blocks that include **rescue** and **always** sections.

```
tasks:
- name: Handle the error
  block:
    - name: Print a message
      ansible.builtin.debug:
        msg: 'I execute normally'

    - name: Force a failure
      ansible.builtin.command: /bin/false

    - name: Never print this
      ansible.builtin.debug:
        msg: 'I never execute, due to the above task failing, :-( '
  rescue:
```

```
- name: Print when errors
  ansible.builtin.debug:
    msg: 'I caught an error, can do stuff here to fix it, :-)'
```

Rescue blocks define the tasks to run when an earlier task in a block fails.

Ansible only runs rescue blocks when a task returns to a **failed** state.

Incorrect task definitions and unreachable hosts will not trigger the rescue block.

# Error Handling in Ansible Using Blocks

The code specified in the `always` section will be executed regardless of the task status of the preceding block.

```
- name: Always do X
  block:
    - name: Print a message
      ansible.builtin.debug:
        msg: 'I execute normally'

    - name: Force a failure
      ansible.builtin.command: /bin/false

    - name: Never print this
      ansible.builtin.debug:
        msg: 'I never execute :-( '
  always:
    - name: Always do this
      ansible.builtin.debug:
        msg: "This always executes, :-)"
```

# Error Handling in Ansible Using Blocks

```
- name: Attempt and graceful roll back demo
  block:
    - name: Print a message
      ansible.builtin.debug:
        msg: 'I execute normally'

    - name: Force a failure
      ansible.builtin.command: /bin/false

    - name: Never print this
      ansible.builtin.debug:
        msg: 'I never execute, due to the above task failing, :-('
  rescue:
    - name: Print when errors
      ansible.builtin.debug:
        msg: 'I caught an error'

    - name: Force a failure in middle of recovery! >:-)
      ansible.builtin.command: /bin/false

    - name: Never print this
      ansible.builtin.debug:
        msg: 'I also never execute :-('
  always:
    - name: Always do this
      ansible.builtin.debug:
        msg: "This always executes"
```

The combination of **always** and **rescue** allows for complex error handling.

Whenever a block fails, the **rescue** section executes tasks to address the failure.

The results from the **block** and **rescue** sections do not influence the **always** section.

# Error Handling in Ansible Using Blocks

If an error occurs within the block and the rescue task succeeds, Ansible will revert the failed status of the original task for the run.

Ansible then continues to proceed as if the original task had succeeded.

The rescued task is deemed successful and does not trigger the **max\_fail\_percentage** or **any\_errors\_fatal** configurations.

Ansible still registers a failure in the playbook statistics.

# Error Handling in Ansible Using Blocks

Utilize blocks with **flush\_handlers** in a rescue task to ensure that all handlers run, even if an error occurs.

```
tasks:
  - name: Attempt and graceful roll back demo
    block:
      - name: Print a message
        ansible.builtin.debug:
          msg: 'I execute normally'
          changed_when: yes
          notify: run me even after an error

      - name: Force a failure
        ansible.builtin.command: /bin/false
    rescue:
      - name: Make sure all handlers run
        meta: flush_handlers
handlers:
  - name: Run me even after an error
    ansible.builtin.debug:
      msg: 'This handler runs even on error'
```



# Error Handling in Ansible Using Blocks

In Ansible, two new variables are crucial for tasks within the rescue portion of a block:

01



**ansible\_failed\_task:**

This variable stores the task that failed and triggered the rescue section.

02



**ansible\_failed\_result:**

This variable captures the return value when a task fails.

# Troubleshooting Techniques

Some important techniques for troubleshooting in Ansible are:

**Check for bad syntax**

Use **ansible-playbook playbooks/PLAYBOOK\_NAME.yml --syntax-check** command to verify the syntax of your playbook

**Flush the redis cache**

Clear the cache by running **ansible-playbook playbooks/PLAYBOOK\_NAME.yml --flush-cache** to prevent stale data from affecting operations

**Run a playbook in dry-run mode**

Execute **ansible-playbook playbooks/PLAYBOOK\_NAME.yml --check** to simulate the playbook's actions without making any changes

# Troubleshooting Techniques

Run a playbook in verbose mode

Add **-v** (or **-vv**, **-vvv**, **-vvvv**, **-vvvvv**) to **ansible-playbook** command for detailed command line logs

Task debugger

Activate Ansible's debugger at task or play level for real-time task troubleshooting

Run a playbook interactively

Run **ansible-playbook playbook.yml --step** to confirm each task before execution

Run a playbook from a particular task

Use **ansible-playbook playbook.yml --start-at-task="<Your\_task>"** to begin at a designated point

## Quick Check



You are writing an Ansible playbook to deploy an application. You want to ensure that if any task fails, you can run specific recovery steps and then continue with the next tasks. You decide to use blocks for error handling. Which directive in Ansible is used within a block to specify tasks that should run if a task fails?

- A. always
- B. rescue
- C. recover
- D. fail

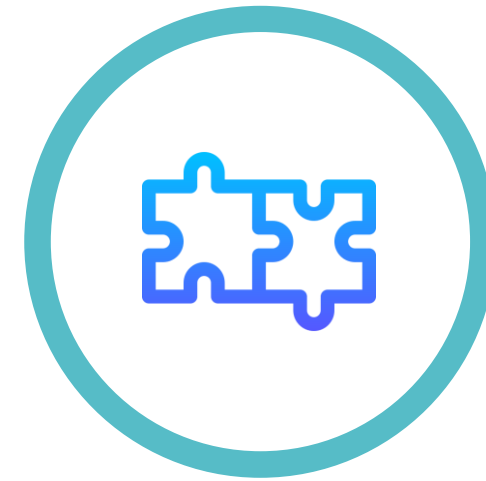


## **Working with Dynamic Inventory**

# What Is Dynamic Inventory?

It is a feature in automation tools like Ansible that automatically retrieves and manages inventory data from various sources in real time.

- It provides information about public and private cloud providers, cobbler system information, and CMDB (Configuration Management Database).
- Inventory plugins are preferred over scripts because they align with the latest updates in the Ansible core code and offer more customization options.



Inventory Plugins



Inventory Scripts

# Dynamic Inventory Plugins

Inventory plugins allow users to connect to various data sources to generate the host inventory that Ansible uses to target jobs.

## Points to Remember:

- The inventory plugins bundled with Ansible are enabled by default or can be used with the **auto** plugin.
- The **ansible.cfg** configuration file can override the default list of enabled plugins. Below is the default list of enabled plugins that ship with Ansible:

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

# Dynamic Inventory Scripts

Ansible uses inventory scripts to fetch host information from external inventory systems.

```
{
  "group001": {
    "hosts": ["host001", "host002"],
    "vars": {
      "var1": true
    },
    "children": ["group002"]
  },
  "group002": {
    "hosts": ["host003", "host004"],
    "vars": {
      "var2": 500
    },
    "children": []
  }
}
```

Inventory scripts must accept **--list** and **--host <hostname>** arguments, returning detailed information about host groups and individual hosts.



## Assisted Practice



### Implementing dynamic inventories

Duration: 20 Min.

#### Problem statement:

You have been assigned a task to demonstrate how to work with dynamic inventories in Ansible, enabling the automatic fetching and usage of inventory data from external sources.

#### Outcome:

By the end of this demo, you will be able to implement Ansible Vault to securely store sensitive data such as passwords, API keys, and other confidential information within Ansible playbooks.

**Note:** Refer to the demo document for detailed steps

# Assisted Practice: Guidelines



Steps to be followed:

1. Install the required tools
2. Create and configure a dynamic inventory script
3. Create and execute a playbook using the dynamic inventory script

## Quick Check

You are managing a cloud infrastructure where servers are frequently added and removed. To keep your Ansible inventory up-to-date without manual intervention, you decide to use a dynamic inventory. Which feature of Ansible allows you to automatically update your inventory based on the current state of your cloud infrastructure?

- A. Static inventory
- B. Inventory groups
- C. Dynamic inventory plugins
- D. Inventory variables

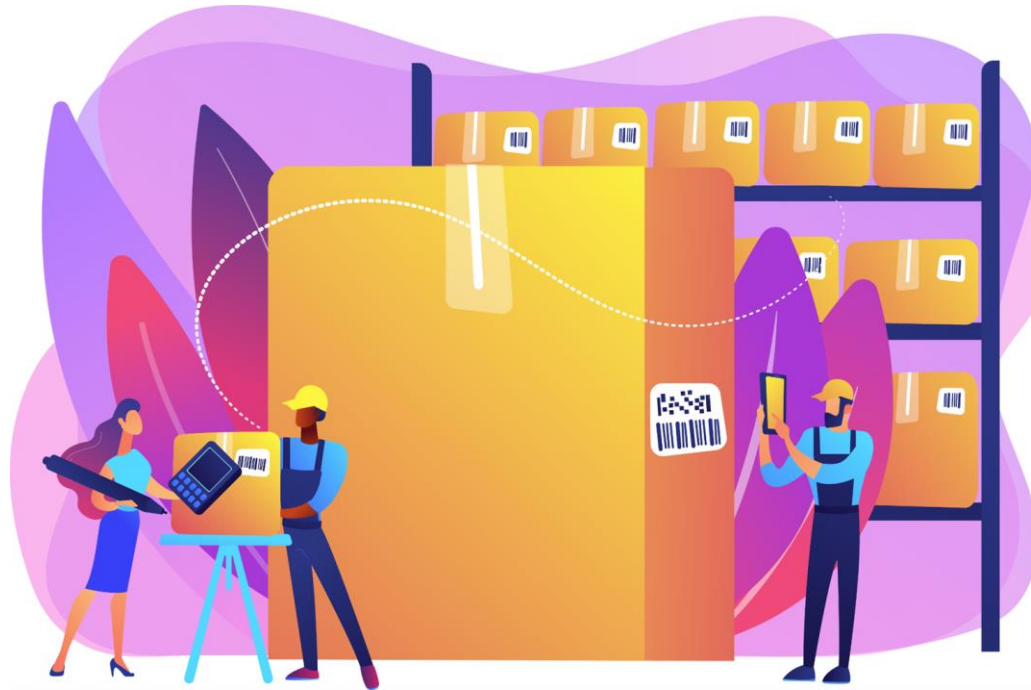




## **Managing Inventory Variables**

# Managing Inventory Variables

Inventory variables in Ansible customize playbook and role behavior by defining configuration settings, connection parameters, and environment-specific values.



After constructing the inventory, users can employ patterns to specify which hosts or groups they want Ansible to operate against.

# Inventory Basics: Formats

Depending on the inventory plugins in use, the inventory file can be in various formats, with INI and YAML being the most utilized formats.

A basic INI file in ***/etc/ansible/hosts*** is shown below:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

# Inventory Basics: Formats

The following code is a sample INI file in YAML format:

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

# Inventory Basics: Hosts and Groups

There are two types of groups available to connect via hosts:



Default groups



Multiple groups



# Connecting Hosts

The following variables influence how Ansible communicates with remote hosts:

Variable	Description
<b>ansible_host</b>	The hostname for connection
<b>ansible_port</b>	The connection port number (Default is 22 for SSH)
<b>ansible_user</b>	The username when connecting to the host
<b>ansible_password</b>	The password to authenticate the host

# SSH Connection

The settings listed below are specific to SSH ports:

Variable	Description
<b>ansible_ssh_private_key_file</b>	Specifies a private key file for SSH when not using ssh-agent
<b>ansible_ssh_common_args</b>	Adds common arguments to the default command lines for SSH, SFTP (SSH File Transfer Protocol), and SCP (Secure Copy)
<b>ansible_sftp_extra_args</b>	Customizes the SFTP command line with additional arguments
<b>ansible_scp_extra_args</b>	Customizes the SCP command line with additional arguments
<b>ansible_ssh_extra_args</b>	Adds extra arguments specifically for the SSH command line
<b>ansible_ssh_pipelining</b>	Enables or disables SSH pipelining to speed up execution
<b>ansible_ssh_executable</b>	Specifies a custom SSH executable instead of the default system SSH

## Non-SSH Connection

It refers to the various connection types other than SSH that can be used to manage remote hosts. The connection type can be altered using the host-specific parameter **ansible\_connection=<connector>**.

The following Non-SSH connectors are available:



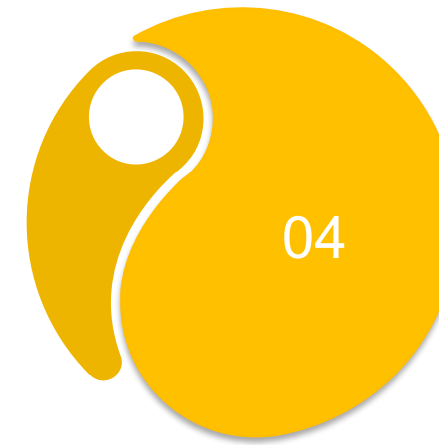
WinRM  
(Windows  
Remote  
Management)



Local  
connection



Docker  
connection



Podman  
connection



SSH with  
bastion  
host

# Non-SSH Connection

The following variables are processed by the Docker connector:

Variable	Description
<b>ansible_host</b>	Name of the Docker container to connect
<b>ansible_user</b>	Username to operate within the container
<b>ansible_become</b>	When set to <b>true</b> , become_user operates in the container
<b>ansible_docker_extra_args</b>	A string containing any additional arguments recognized by Docker

# Deployment in Non-SSH Connection

The following code is an example of deployment to the containers:

```
- name: Create a jenkins container
  community.general.docker_container:
    docker_host: myserver.net:4243
    name: my_jenkins
    image: jenkins

- name: Add the container to inventory
  ansible.builtin.add_host:
    name: my_jenkins
    ansible_connection: docker
    ansible_docker_extra_args: "--tlsverify --tlscacert=/path/to/ca.pem"
    ansible_user: jenkins
    changed_when: false

- name: Create a directory for ssh keys
  delegate_to: my_jenkins
  ansible.builtin.file:
    path: "/var/jenkins_home/.ssh/jupiter"
    state: directory
```

## Quick Check

You are setting up an Ansible inventory to manage servers with different connection methods, including non-SSH connections. Which Ansible inventory variable is used to specify a non-SSH connection type for a host?

- A. `ansible_connection`
- B. `ansible_host`
- C. `ansible_port`
- D. `ansible_user`





# **Ansible Tower**

# What Is Ansible Tower?

Ansible Tower, now known as Red Hat Ansible Automation Platform, is a web-based solution intended to streamline the utilization of Ansible for IT automation.

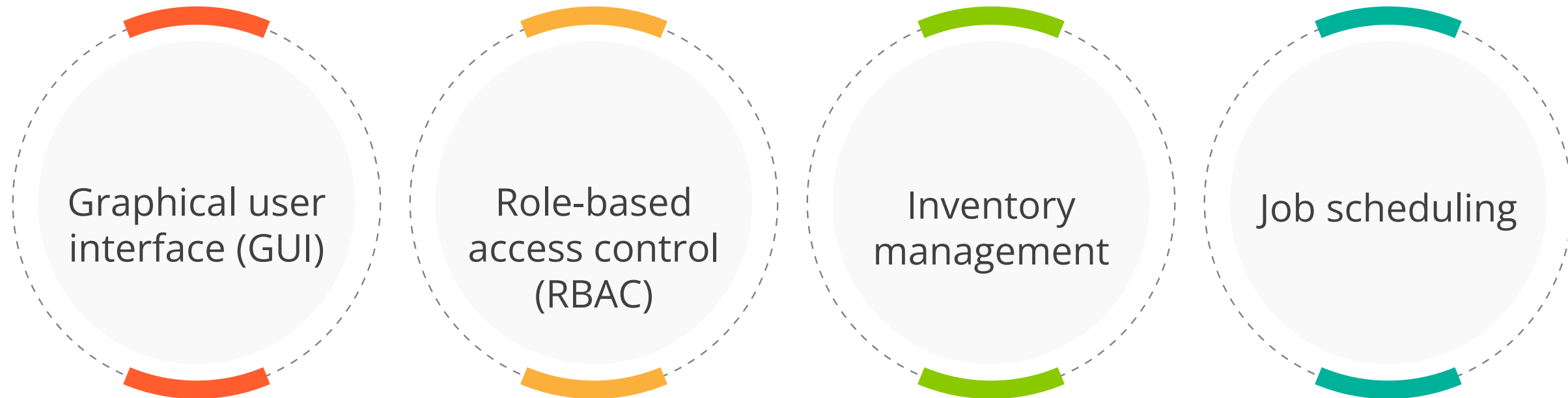


It is used for enterprise-level automation of IT tasks and provides a centralized platform for overseeing and orchestrating Ansible automation on a large scale.

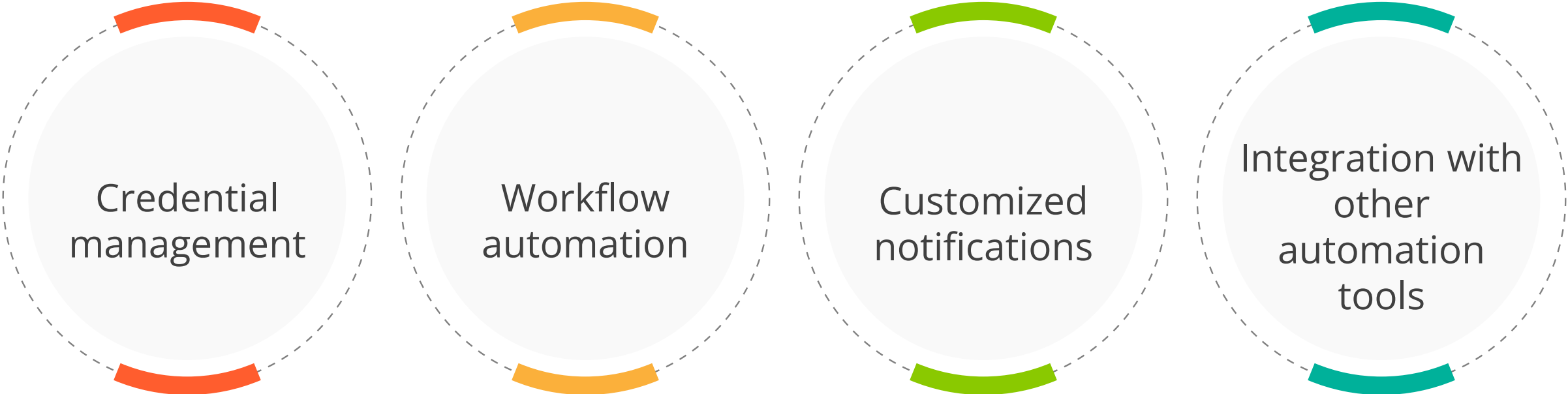


# Ansible Tower Features

Tower is a graphically-enabled framework accessible via a web interface and a REST API endpoint for Ansible, the open-source IT orchestration engine. It has the following features:



# Ansible Tower Features



Credential  
management

Workflow  
automation

Customized  
notifications

Integration with  
other  
automation  
tools

## Quick Check



You are managing a large number of Ansible playbooks and need a centralized system to help orchestrate and manage these playbooks across multiple teams and environments. You have decided to use Ansible Tower. Which feature of Ansible Tower allows you to control and manage user access to different playbooks and inventories?

- A. Job templates
- B. Workflows
- C. Role-Based Access Control (RBAC)
- D. Surveys

# Key Takeaways

- Roles are not playbooks. They are small functions that can be used individually within playbooks.
- Ansible Galaxy is a free Galaxy website for finding, downloading, and sharing community-developed roles and collections.
- Collections can be used to package and distribute roles, modules, playbooks, and plugins.
- Roles can be dependent on other roles, and when a user installs a role that has dependencies, those dependencies will automatically be installed to the **roles\_path**.



# Key Takeaways

- Ansible Vault is a tool designed to encrypt confidential information, allowing users to safeguard sensitive data like passwords from being exposed in plain text.
- Ansible Vault has the capability to encrypt any structured data file utilized within Ansible. The entire file is encrypted within the vault for added security.
- Ansible Tower simplifies Ansible usage for IT departments through its web-based interface.
- Ansible uses a group of lists known as inventory to work against numerous managed nodes in the infrastructure simultaneously.



# Initializing Ansible Roles and Inventory

**Duration: 25 Min.**

**Project agenda:** To automate the deployment and management of an Apache web server using Ansible roles, securely handle sensitive information using Ansible Vault, and dynamically manage the inventory using a custom Python script for AWS EC2 instances

**Description:** As a DevOps engineer at a growing tech company, you are tasked with automating the deployment of Apache web servers. The company's infrastructure is dynamic, requiring frequent updates and the ability to manage sensitive configuration data securely. Your objective is to create an Ansible role for installing and configuring Apache, use Ansible Vault to manage sensitive data like passwords and implement a dynamic inventory to handle the constantly changing server environment.



# Initializing Ansible Roles and Inventory

Duration: 25 Min.

## Perform the following:

1. Initialize the Ansible role for Apache
2. Define tasks for the Ansible role
3. Configure and use Ansible Vault
4. Implement dynamic inventory

**Expected deliverables:** An Ansible playbook that automates Apache web server deployment and management, complete with detailed installation and execution documentation.





**Thank you**