Ansible

Jai

🙏 🙏 Namaskaram 🙏 🙏

Introduction to Ansible

Definition: Ansible is an open-source, command-line IT automation tool written in Python.
Purpose: It is used for automating IT tasks across different areas.

Uses of Ansible

System Configuration

Example: Creating a mqm user across all servers in a new MQ project setup.
Example: Opening a specific port on hundreds of servers for a new application.

Software/Application Deployment

Example: Installing any command or a web server across multiple servers.

System Updates and Patching

Efficiently apply updates or patches across multiple servers.

Ansible Features

Configuration Management: Primarily used for configuring systems.
Architecture: Follows a push model and is agentless.

Operation Modes:

Ansible CLI Commands (Ad-hoc commands).
Ansible Playbooks (Written in YAML)

Prerequisites for the Course

Server Provisioning: Knowledge of server provisioning using VMware or cloud services.
Unix/Linux Environment: Understanding of basic commands, VI/VIM editors, and SSH connections.
Scripting for Module Development: Knowledge in shell scripting or Python scripting for developing custom modules or collections (optional).

Key Takeaways

Ansible's Simplicity: Despite its powerful features, Ansible is designed to be straightforward, making IT automation accessible to those with basic programming or scripting knowledge.
Community Modules: Emphasis on utilizing community modules and collections for most real-world scenarios, highlighting the collaborative nature of Ansible's ecosystem.

Ansible Distributions Overview

Two Distributions from 2.10 Onwards: Ansible Core and Ansible Community.
Ansible Core is the basic Ansible tool used for running ad-hoc commands or playbooks.
Ansible Community builds on top of Ansible Core, adding more collections or modules.

Understanding Modules and Collections

Modules: Small programs to execute specific tasks, like installing a package on servers.
Collections: Introduced in Ansible 2.10, collections are groups of modules, roles, and other resources that can be used to automate a broader set of tasks.
Example: To install a package, use a module designed for that purpose. With Ansible Community, users access a broader set of these modules (now referred to as collections).

Installation and Version Management

Installation: Different methods are available, but pip command is a common approach.
Finding Versions: Visit the official Ansible documentation to locate the current versions of Ansible Core and Community, including their changelogs.

Example Installation Commands:

For Ansible Community: pip3 install ansible==7.0.0
For Ansible Core: pip3 install ansible-core==2.14.0

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

Installation of Ansible Community :

```
sudo yum install python3-pip -y
pip3 install ansible==8.0.0
ansible --version
ansible-community --version
ansible-doc -l | wc -l
```

Installation of Ansible Core:

```
sudo yum install python3-pip -y
pip3 install ansible-core==15.0.0
ansible --version
ansible-doc -l | wc -l
```

Introduction to Ansible

•**Overview:** Ansible is an open-source IT automation tool used to manage various IT needs such as configuration management, application deployment, task automation, and multi-node orchestration.

•**Agentless Architecture:** Unlike other management tools that require a software agent on the client nodes, Ansible operates in an agentless manner, leveraging existing SSH protocols to communicate and execute tasks on remote servers.

Working with Remote Servers using Ansible

Installation and Setup:

Ansible is installed on a control node, also referred to as the Ansible Admin Node or Ansible Controller Node.
Managed nodes do not require Ansible installation, just Python and SSH access.

Connectivity and Communication:

Ansible uses SSH for secure communication with remote servers (managed nodes). No additional SSH software installation is required on the managed nodes as they typically come with SSH by default.

Agentless Nature:

Being agentless, Ansible doesn't require any agent software to be installed on the remote nodes, making it lightweight and reducing management overhead.
Example: If you have 100 servers to manage, you don't need to install Ansible on all of them. Only the control node needs Ansible installed.

Execution Model:

Ansible operates on a "push model" where it pushes small programs, called modules, to the remote nodes to execute tasks.
Each task is executed by the corresponding module on the remote server using the server's Python interpreter.
Example: To update a package on all servers, Ansible pushes a package management module to each server, executes it using Python, then cleans up after the task is complete.

Push Model and Agentless Architecture:

The push model ensures that changes are initiated from the control node, allowing for centralized management and execution of tasks.
The agentless architecture simplifies setup and reduces the potential attack surface since there's no need to manage software agents on managed nodes.

ssh-keygen
ssh-copy-id ansible@172.31.17.56

ssh 'ansible@172.31.17.56'

```
[ansible@ip-172-31-44-194 ~]$ ansible -i inventory all -m ping
The authenticity of host '172.31.43.130 (172.31.43.130)' can't be established.
ED25519 key fingerprint is SHA256:lN6bEL8IS8ytipnLNXYA1qewRQPsUeDBMik8INgSV3k.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? ^C [ERROR]: User interrupted execution
[ansible@ip-172-31-44-194 ~]$
[ansible@ip-172-31-44-194 ~]$ export ANSIBLE_HOST_KEY_CHECKING=False
```

```
[ansible@ip-172-31-44-194 ~]$ vi inventory
[ansible@ip-172-31-44-194 ~]$ ansible -i inventory all -m ping
[WARNING]: Platform linux on host 172.31.17.56 is using the discovered Python i
interpreter could change the meaning of that path. See https://docs.ansible.com/
information.
172.31.17.56 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.9"
    },
    "changed": false,
    "ping": "pong"
}
```

```
[ansible@ip-172-31-44-194 ~]$ touch ansible.cfg
[ansible@ip-172-31-44-194 ~]$ ansible --version
ansible [core 2.15.9]
  config file = /home/ansible/ansible.cfg
  configured module search path = ['/home/ansible/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/local/lib/python3.9/site-packages/ansible
  ansible collection location = /home/ansible/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/local/bin/ansible
  python version = 3.9.16 (main, Sep  8 2023, 00:00:00) [GCC 11.4.1 20230605 (Red Hat 11.4.1-2)] (/usr/bin/python3)
  jinja version = 3.1.3
  libyaml = True
[ansible@ip-172-31-44-194 ~]$ vi ansible.cfg
[ansible@ip-172-31-44-194 ~]$ cat ansible.cfg
[defaults]
inventory=./inventory
[ansible@ip-172-31-44-194 ~]$ ansible all -m ping
[WARNING]: Platform linux on host 172.31.17.56 is using the discovered Python interpreter at /usr/bin/python3.9, but :
interpreter could change the meaning of that path. See https://docs.ansible.com/ansible-core/2.15/reference_appendices
information.
172.31.17.56 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.9"
    },
    "changed": false,
    "ping": "pong"
}
```

```
[ansible@ip-172-31-44-194 ~]$ vi ansible.cfg
[ansible@ip-172-31-44-194 ~]$ cat ansible.cfg
[defaults]
inventory=./inventory
host_key_checking=False
[ansible@ip-172-31-44-194 ~]$
```

Ansible Ad-hoc Commands

Overview of Ansible Commands

Introduction to Ansible Commands

Ansible operations can be performed using two main methods: Ansible ad-hoc commands (or CLI commands) and Ansible Playbooks.
Ad-hoc commands are suitable for simple, one-time tasks across your managed nodes.

Ad-hoc Commands Basics

Start with the ansible command followed by specifications on what to execute.
Limitation: Only one operation or task can be executed at a time.

Use Cases for Ad-hoc Commands

Ideal for straightforward tasks such as checking uptime or Java version across managed nodes.
Example: ansible all -m ansible.builtin.command -a uptime checks uptime on all nodes.

Practical Examples

Checking Uptime

Command: ansible all -m ansible.builtin.command -a uptime

This command uses the ansible.builtin.command module to execute the uptime command on all managed nodes.

Installing a Package

Example: Installing Nginx on managed nodes.

Command: Use the appropriate Ansible module for package installation with root privileges.

Example: ansible all -m ansible.builtin.package -a "name=httpd state=present" -b

↳ -b = become = sudo

↳ For this to work

visudo → ansibleuser ALL=(ALL) NOPASSWD: ALL

Modules in Ansible

What is a Module?

Modules are small programs (usually Python) that Ansible executes on managed nodes to perform specific tasks.

Selection of the module depends on the task requirement (e.g., command for executing commands, user for creating users).

Module Name Structure

From Ansible 2.10 onwards, modules follow the namespace.collection.module structure.

To list available modules: ansible-doc -l

To view module documentation: ansible-doc <module_name>

Executing Ad-hoc Commands

Command Structure

Start with ansible, followed by the host pattern (which nodes to target), the module to use, and any necessary arguments.

Host Patterns

Examples include specifying individual IPs, using all for all nodes in the inventory, or using patterns to target specific groups.

Connectivity Validation

Use the ping module for testing connectivity: ansible all -m ansible.builtin.ping

Demonstrates successful Ansible communication with the managed nodes.

```
ansible 172.31.31.205,172.31.22.210 -m ansible.builtin.ping
ansible 172.31.31.205 -m ansible.builtin.ping
ansible all -m ansible.builtin.ping
```

Executing Unix or Linux commands via Ansible ad-hoc commands into a structured presentation. This will include key points, examples, and insights into using command and shell modules effectively.

Choosing the Right Module: For executing Unix or Linux commands, Ansible provides two primary modules: command and shell.

Executing Commands with command and shell Modules
Command Module Usage:
Designed for executing commands directly without shell processing.
Syntax Example: ansible all -m ansible.builtin.command -a "free -mh"
Use Case: Ideal for straightforward commands where no shell-specific features (like pipes, redirects, or environment variables) are required.

Shell Module Usage:
Executes commands through the shell on the managed nodes.
Syntax Example: ansible all -m ansible.builtin.shell -a "free -mh"
Use Case: Necessary when you need to leverage shell-specific features or execute complex commands involving environment variables, pipelines (|), redirections (>, <), or logical operators (&&, ||).

Checking Memory Usage:

Using command: ansible all -m ansible.builtin.command -a "free -mh"

Using shell: ansible all -m ansible.builtin.shell -a "free -mh"

Outcome: Both modules provide the memory usage (free -mh command output) across all managed nodes, showcasing the basic use case where command suffices.

Executing Multiple Commands:

Attempt with command: ansible all -m ansible.builtin.command -a "free -mh; date"

Success with shell: ansible all -m ansible.builtin.shell -a "free -mh; date"

Analysis: The command module fails to execute because it doesn't process shell operators like ;. The shell module successfully executes both commands, demonstrating its capability to handle complex command executions.

Differences Between command and shell

Environment and Special Characters: command does not process shell environments or special characters, making it more secure and faster for straightforward tasks. shell should be used for commands needing shell features.

Performance: command might perform faster than shell for simple commands since it doesn't involve shell processing overhead.

Choosing Between command and shell

Simple Commands: Use command for efficiency and security.

Complex Commands: Use shell when shell features or complex command structures are needed.

```
# Basic Command Execution without Spaces
# This command doesn't require quotations around the argument since it doesn't contain spaces.
ansible all -m ansible.builtin.shell -a "date"
# Output: Displays the current date and time on all managed nodes.

# Attempting to run a command with spaces without proper quotation (this will fail or produce unintended results).
ansible all -m ansible.builtin.shell -a free -mh

# Using Single Quotations for Commands with Spaces
# Properly enclose arguments that include spaces with single quotations.
ansible all -m ansible.builtin.shell -a 'free -mh'
# Output: Displays memory usage information on all managed nodes, successfully handling the space in the argument.

# Using Double Quotations for Commands with Spaces
# Double quotations can also be used, and are necessary when the command itself includes single quotes.
ansible all -m ansible.builtin.shell -a "free -mh"
# Output: Similarly, displays memory usage information on all managed nodes.

# Nested Quotations in Commands
# If your command includes single quotations (e.g., when filtering output with grep), enclose the entire argument in double quotes.
ansible all -m ansible.builtin.shell -a "grep 'Mem' | cut -d' ' -f2"
# This command looks for lines containing "Mem" in the output of another command, demonstrating how to use nested quotations.
```

Complex Command Example with Pipes and Filters
This command chains multiple commands together to process the output of `free -mh`, demonstrating handling of complex arguments.
```
ansible all -m ansible.builtin.shell -a "free -mh | grep 'Mem' | tr -s ' ' | cut -d' ' -f2"
```
Explanation:
1. `free -mh` to display memory usage.
2. `grep 'Mem'` to filter lines containing "Mem".
3. `tr -s ' '` to squeeze multiple spaces into a single space.
4. `cut -d' ' -f2` to extract the second field, using space as the delimiter.
Output: Shows the specific memory information for "Mem" from the `free -mh` command on all managed nodes.

Note: Remember to replace `all` with your actual target hosts or groups as defined in your Ansible inventory.

Process of Ansible Ad-hoc Commands

Inventory and SSH Connection:

Inventory Read: Ansible starts by reading the list of managed nodes from the inventory file.

SSH Connection: Ansible then establishes SSH connections to these nodes using the current login user from the Ansible control node.

Module Execution:

Module Transfer: The selected module is transferred to a temporary location on the managed nodes, typically in the user's home directory.

Task Execution: Ansible executes the task on the managed nodes using Python.

Module Cleanup: After task completion, Ansible cleans up by deleting the temporary module.

Python Version Compatibility:

Control Node Python Version: Ansible core 2.14 uses Python versions 3.9 to 3.11.

Managed Node Python Version: Python 2.7 or later is generally acceptable; however, matching the control node's Python range is better for compatibility.

Handling Non-Python Environments:

Even without Python on managed nodes, Ansible can still execute tasks using modules like raw and script.

Parallel Execution and Batches:

Forks Parameter: Controls how many tasks Ansible executes in parallel, with a default value of 5.

Batch Execution: Ansible executes tasks on a set number of nodes at a time, based on the forks value.

Modifying Forks Value

Ad-hoc Fork Modification:

Environment Variable: Set the ANSIBLE_FORKS variable before running ad-hoc commands to adjust parallel execution.

```
export ANSIBLE_FORKS=10
ansible all -m ansible.builtin.shell -a "date"
```

Persistent Configuration Change:

Ansible Configuration File: Set the default forks value in ansible.cfg under the [defaults] section.

```
[defaults]
forks=10
```

What is a Fork in Ansible?

Think of a "fork" in Ansible as a worker or an agent that can handle tasks independently. If you have multiple tasks to do, instead of doing them one after another, you can assign each task to a separate worker and have all the tasks done at the same time. In Ansible, these workers operate in parallel, which means they can run multiple tasks across multiple servers simultaneously.

How Does Parallel Execution Work?

When you execute an Ansible ad-hoc command or a playbook, and you have multiple servers (managed nodes) you want to manage, Ansible can perform actions on more than one server at a time. This is where the forks value comes into play.

If you set the forks to 10, Ansible can connect and perform the specified tasks on up to 10 servers at the same time.
If you have fewer than 10 servers, Ansible will connect to all of them at once because the forks value exceeds the number of servers.
If you have more servers, say 20, Ansible will manage 10 of them at first, and once tasks on one or more of those servers are completed, it will start executing tasks on the remaining servers until all servers have been managed.

Example Scenarios:

10 Servers, Forks Set to 5:
Ansible will start tasks on 5 servers first.
Once any of those tasks are completed, it will pick the next server and start a task there, and so on until all servers are handled.

10 Servers, Forks Set to 10:
Ansible will start tasks on all 10 servers at the same time.

20 Servers, Forks Set to 10:
Ansible will start tasks on 10 servers.

As soon as it finishes tasks on some of the first batch, it will continue with the remaining servers, keeping up to 10 tasks running at the same time until all servers have been managed.

Why Not Always Use a High Forks Value?

Using a high forks value can speed up the execution by parallelizing tasks. However, it also means that your Ansible control node and network must handle multiple connections and tasks at once. If your control node doesn't have enough resources (CPU, memory), or if your network is limited, trying to do too much at once can slow down everything or lead to errors.

It's like having too many apps open on your phone; if the phone doesn't have enough power to handle them all, it might freeze or crash. The same goes for your Ansible control node.

Ansible Ad-Hoc Command to Work with Files

File Modules to Work with files | file, Copy, lineinfile, blockinfile

Ansible Modules for File Operations

Overview of File Modules:

Ansible includes a range of modules for different file-related operations.

These modules are part of Ansible core and therefore have the prefix ansible.builtin. when using their fully qualified collection names.

List of File Modules:

file Module: Manages files, directories, and symlinks. It can be used to create, delete, or modify the properties of file system entities.

stat Module: Retrieves file or filesystem status details, similar to the Unix/Linux stat command.

copy Module: Copies files from the local machine to remote locations. It can also create files with specific content on managed nodes.

lineinfile Module: Adds, removes, or modifies a single line in a file.

blockinfile Module: Inserts, updates, or removes a block of lines in a file.

template Module: Deploys files to remote nodes using Jinja2 templates, allowing for dynamic content generation.

fetch Module: Fetches files from remote nodes to the local machine.

Using the Modules:

Select the appropriate module based on the file operation required.

For each module, there are parameters and options that you can specify to perform the needed actions.

Accessing Module Documentation

Command Line Help:

Use the ansible-doc command followed by the module name to access the documentation and examples from the command line.

```
ansible-doc ansible.builtin.file
```

Online Documentation:

Visit the official Ansible documentation website and search for the module you need help with, like ansible.builtin.file.
Review the page to understand the module's options and to find sample playbook snippets.

```
# 1. Create a new directory on all managed nodes
ansible all -m ansible.builtin.file -a "path=/tmp/jai state=directory"

# 2. Retrieve file status of '/etc/passwd' on all managed nodes
ansible all -m ansible.builtin.stat -a "path=/etc/passwd"

# 3. Copy a local file to a remote location on all managed nodes
ansible all -m ansible.builtin.copy -a "src=/local/path/to/file dest=/remote/path"

# 4. Modify a line in a file on all managed nodes
ansible all -m ansible.builtin.lineinfile -a "path=/remote/path/to/file line='Your new line of text' state=present"

# 5. Insert a block of text in a file on all managed nodes
ansible all -m ansible.builtin.blockinfile -a "path=/remote/path/to/file block='|\n  This is a block\n  of text to insert'"

# 6. Deploy a dynamic file using a Jinja2 template on all managed nodes
ansible all -m ansible.builtin.template -a "src=/local/path/to/template.j2 dest=/remote/path/to/file"

# 7. Fetch a file from all managed nodes to the local machine
ansible all -m ansible.builtin.fetch -a "src=/remote/path/to/file dest=/local/path"

# Note: Replace placeholders like '/local/path/to/file' and '/remote/path' with actual paths.
```

Ad-Hoc Commands with File Module to Create a File

Ansible File Module Essentials

1.Purpose of the File Module:

1. It's utilized for file creation, deletion, and setting permissions.
2. Can manage both files and directories on managed nodes.

```
# Create a new file on all managed nodes in the /tmp directory
ansible all -m ansible.builtin.file -a "path=/tmp/test state=touch"
# Output: A new file named 'test' is created in the /tmp directory on all managed nodes.

# Create a new file on all managed nodes in the /tmp directory with specific permissions (755)
ansible all -m ansible.builtin.file -a "path=/tmp/testnew state=touch mode='0755'"
# Output: A new file named 'testnew' is created in the /tmp directory with 755 permissions on all managed nodes.

# Create a new directory on all managed nodes in the /tmp directory with specific permissions (755)
ansible all -m ansible.builtin.file -a "path=/tmp/new_directory state=directory mode='0755'"
# Output: A new directory named 'new_directory' is created in the /tmp directory with 755 permissions.

# Validate the existence and get details of the '/tmp/test' file on all managed nodes
ansible all -m ansible.builtin.file -a "path=/tmp/test"
# Output: Shows the details of the '/tmp/test' file if it exists.
```

Change permissions of the '/tmp/test' file to 777 on all managed nodes
ansible all -m ansible.builtin.file -a "path=/tmp/test mode='0777'"
Output: Changes the permissions of the '/tmp/test' file to 777.

Delete the '/tmp/test' file from all managed nodes
ansible all -m ansible.builtin.file -a "path=/tmp/test state=absent"
Output: The '/tmp/test' file is deleted from all managed nodes.

Re-validate the absence of the '/tmp/test' file on all managed nodes
ansible all -m ansible.builtin.file -a "path=/tmp/test"
Output: Confirms that the '/tmp/test' file no longer exists (will show 'state=absent').

Interview Reference [Playbook to Adhoc Command]

Translating Playbook Tasks into Ad-hoc Commands

Understanding Playbook Tasks:

Playbook tasks are defined with key-value pairs, whereas ad-hoc command arguments use a key=value format.

Converting Playbooks to Ad-hoc Commands:

Identify the module and its arguments in the playbook.

Construct the ad-hoc command by starting with ansible, specifying the host pattern, module, and then appending arguments with -a.

Example Conversion:

Given a playbook task to create a directory:

```
- name: Create a directory
  ansible.builtin.file:
    path: /path/to/directory
    state: directory
    mode: '0755’
```

The corresponding ad-hoc command would be:

```
ansible all -m ansible.builtin.file -a "path=/path/to/directory state=directory mode='0755'"
```

Ad-Hoc Commands with stat

Ansible stat Module Overview

Purpose of stat Module:

It retrieves statistics about file system entries.
Useful in conditional logic within playbooks for tasks like checking file existence, ownership, or file checksums.

Using stat Module in Ad-hoc Commands:

Syntax for ad-hoc command: ansible [host-pattern] -m ansible.builtin.stat -a "path=[file_path]"
The module returns a set of details about the specified file or directory.

Key Outputs:

exists: Indicates whether the file or directory exists.
gid: Group ID of the file.
group: Group ownership of the file.
uid: User ID of the owner.
owner: User name of the owner.
mtime: Last modification time.
islnk: Boolean that shows if the path is a symlink.

```
ansible all -m ansible.builtin.stat -a "path=/etc/passwd"
```

Checking File Statistics:

Gather information about a file's state and attributes on managed nodes.
Use the exists key to determine if a file is present before performing further actions.

Fetching File System Stats:

Apply the stat module to directories and even entire file systems.

Ad-Hoc Commands with Copy Module

Ansible Copy Module Overview

Purpose of the Copy Module:

Copies files from the control node to managed nodes.
Can create files on managed nodes with specific content.

Syntax for the Copy Module in Ad-hoc Commands:

Basic structure: ansible [host-pattern] -m ansible.builtin.copy -a [arguments]

Using the Copy Module:

Copy a File: To copy a file from the control node to managed nodes.
Set Permissions: Specify file permissions with the mode option.
File Ownership: Set the owner and group for the file (optional).
Inline Content: Create a file with inline content directly on managed nodes.

Backing Up Files:

Use the backup option to create a backup of the destination file if it already exists and is different.

```
# Copy 'demo.sh' from the control node to the '/tmp' directory on all managed nodes
ansible all -m ansible.builtin.copy -a "src=~/demo.sh dest=/tmp/demo.sh mode='0644'"
# Output: The 'demo.sh' file is copied to the '/tmp' directory with permissions set to 0644.

# Verify the copied content on the managed nodes by reading the content of 'demo.sh'
ansible all -m ansible.builtin.command -a "cat /tmp/demo.sh"
# Output: Displays the content of 'demo.sh' from the '/tmp' directory.

# Update the 'demo.sh' file on the control node and add the line 'USER' to it
echo "USER" >> ~/demo.sh

# Re-copy the updated 'demo.sh' file to the managed nodes with a backup of the old file
ansible all -m ansible.builtin.copy -a "src=~/demo.sh dest=/tmp/demo.sh mode='0644' backup=yes"
# Output: The updated 'demo.sh' file is copied, and a backup is created if the file differs from the previous version.

# Create a new file 'demo.txt' on the managed nodes with the inline content 'This is demo data'
ansible all -m ansible.builtin.copy -a "content='This is demo data' dest=/tmp/demo.sh backup=yes"
# Output: 'demo.txt' is created with the specified content. Backup is not created since it's the file's first creation.
```

Interview Reference [Playbook to Adhoc Command]

```
# Convert a sample playbook task into an ad-hoc command
# Sample playbook task for reference:
# - name: Copy 'demo.sh' to managed nodes
#   ansible.builtin.copy:
#     src: ~/demo.sh
#     dest: /tmp/demo.sh
#     mode: '0644'

# The equivalent ad-hoc command is:
ansible all -m ansible.builtin.copy -a "src=~/demo.sh dest=/tmp/demo.sh mode='0644'"
# Output: Performs the same action as the playbook task, copying 'demo.sh' to the '/tmp' directory on all managed nodes with the specified mode.
```

Ad-Hoc Commands with lineinfile Module to Append/Replace/Delete Lines

Ansible lineinfile Module Overview

Functionality of the lineinfile Module:

The lineinfile module is akin to the sed command in Unix/Linux.
It allows you to manage lines in a file, such as appending, inserting, modifying, or removing them.

Adding Lines to a File:

To append a line, you must specify the file path and the line content.
The module will only add the line if it does not already exist in the file.

Inserting Lines Based on Regex:

You can specify a regex to determine where to insert a new line.
You can use insert_after or insert_before to control the placement of the new line.

Replacing and Deleting Lines:

Use backrefs=yes and state=present along with regex to replace a line.
Set state=absent to remove lines that match the regex pattern.

```
# Append a line to a file on managed nodes (fails if file does not exist)
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh line='This is a demo line'"
# Output: Appends the specified line to '/tmp/demo.sh' if the file exists.

# Append another line to the same file
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh line='This is another line'"
# Output: Appends the new line to '/tmp/demo.sh'.

# Check the current content of the file
ansible all -m ansible.builtin.command -a "cat /tmp/demo.sh"
# Output: Shows the current lines in '/tmp/demo.sh'.

# Insert a line after a matching pattern using regex
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh insertafter='file' line='This is an extra line'"
# Output: Inserts the new line after the first occurrence of 'file' in '/tmp/demo.sh'.

# Insert a line after the first match of a string in the file
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh insertafter='^is' line='This is a new extra line' firstmatch=yes"
# Output: Inserts the new line after the first occurrence of a line starting with 'is'.
```

Replace a line in the file that matches a given regex
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh regex='^This is a demo line$' line='This line has been replaced'"
Output: Replaces the line that exactly matches 'This is a demo line' with 'This line has been replaced'.

Delete a specific line in the file
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh regex='to be deleted' state=absent"
Output: Removes lines containing 'to be deleted' from '/tmp/demo.sh'.

Delete all lines containing a specific string
ansible all -m ansible.builtin.lineinfile -a "path=/tmp/demo.sh regex='is' state=absent"
Output: Removes all lines that contain 'is' from '/tmp/demo.sh'.

Ad-Hoc Command with Fetch Module to Download from Managed Nodes

Understanding the fetch Module in Ansible

Purpose of the fetch Module:

The fetch module is used to download files from managed nodes to the control node.
It is not used as frequently as other modules but is important for tasks like backup or retrieval of remote system information.

Basic Use of fetch Module:

Syntax for fetch module in ad-hoc command: ansible [host-pattern] -m ansible.builtin.fetch -a [arguments]

Downloading a File from Managed Nodes:

Define the source file path on the managed nodes and the destination path on the control node.
The directory structure of the managed nodes will be preserved in the destination.

```
# Download a file from all managed nodes to the Ansible control node
ansible all -m ansible.builtin.fetch -a "src=/tmp/demo.sh dest=/home/ansibleuser"
# Output: The file demo.sh is downloaded from all managed nodes and placed in the specified downloads directory.

# Navigate to the downloads directory to inspect the downloaded files
cd /home/ansibleuser
# Output: You will find a directory structure mimicking the managed nodes' paths with the downloaded files.

# Flatten the directory structure while downloading the file
ansible all -m ansible.builtin.fetch -a "src=/tmp/demo.sh dest=/home/ansibleuser flat=yes"
# Output: The file demo.sh is downloaded directly to the downloads directory without preserving the remote directory structure.

# Use the inventory_hostname variable to include the managed node's hostname in the downloaded file's name
ansible all -m ansible.builtin.fetch -a "src=/tmp/demo.sh dest=/home/ansibleuser/{{ inventory_hostname }}_demo.sh flat=yes"
# Output: The files are downloaded to the downloads directory and renamed to include the managed nodes' hostnames.

# Change to the project directory if needed and re-run the command if an error occurs due to wrong directory
cd /path/to/your/project/directory
ansible all -m ansible.builtin.fetch -a "src=/tmp/demo.sh dest=/home/ansibleuser/{{ inventory_hostname }}_demo.sh flat=yes"
# Output: The command runs successfully from the correct project directory, downloading and renaming the files as specified.
```

Ansible Modules for Package Installation

Understanding Package Modules:

yum Module: Used for managing packages on RHEL/CentOS systems.
dnf Module: A newer package manager for Fedora and RHEL 8+; can be used instead of yum.
apt Module: For managing Debian/Ubuntu packages.
package Module: A generic module that works with the underlying package management system but is less commonly used due to lack of fine control.

Executing Package Management Commands:
Root Privileges: Package installation typically requires superuser privileges, which you can provide using the -b (or --become) option.

Installing Packages:
Specify the package name and desired state (present, absent, latest) using the name and state arguments.
Example: Installing nginx with yum: `ansible rhel_hosts -m ansible.builtin.yum -a "name=nginx state=present" –b`

Uninstalling Packages:
Set the state to absent to remove packages.
Example: Removing nginx: `ansible rhel_hosts -m ansible.builtin.yum -a "name=nginx state=absent" –b`

Updating Packages:
Set the state to latest to update to the latest version available.
Example: Updating nginx: `ansible rhel_hosts -m ansible.builtin.yum -a "name=nginx state=latest" -b`

```
# Install nginx on all RHEL/CentOS nodes using yum
ansible rhel_hosts -m ansible.builtin.yum -a "name=httpd state=present" -b

# Install nginx on all Debian/Ubuntu nodes using apt
ansible ubuntu_hosts -m ansible.builtin.apt -a "name=nginx state=present" -b

# Remove nginx from all RHEL/CentOS nodes
ansible rhel_hosts -m ansible.builtin.yum -a "name=nginx state=absent" -b

# Update nginx to the latest version on all Debian/Ubuntu nodes
ansible ubuntu_hosts -m ansible.builtin.apt -a "name=nginx state=latest" -b
```

Ansible Ad-Hoc Commands on Ansible Controller Node

```
# Ping the localhost to check Ansible's ability to execute locally
ansible localhost -m ansible.builtin.ping
# Output: Should return SUCCESS, indicating that the controller node can run Ansible tasks on itself.

# If facing permissions issues on older versions of Ansible, set up SSH keys:
# Generate SSH keys (if not done previously)
ssh-keygen -t rsa -b 2048 -N "" -f ~/.ssh/id_rsa

# Append the public key to authorized_keys to allow passwordless SSH to localhost
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

# Secure the authorized_keys file by setting appropriate permissions
chmod 600 ~/.ssh/authorized_keys

# Now, run any ad-hoc command, like checking the Ansible version, on the controller
ansible localhost -m ansible.builtin.command -a "ansible --version"
# Output: Displays the Ansible version installed on the controller node.
```

Idempotent

Understanding Idempotency in Ansible

Definition of Idempotency:

Ansible tasks are designed to be idempotent, meaning they will only make changes if the desired state is not already present on the managed node.

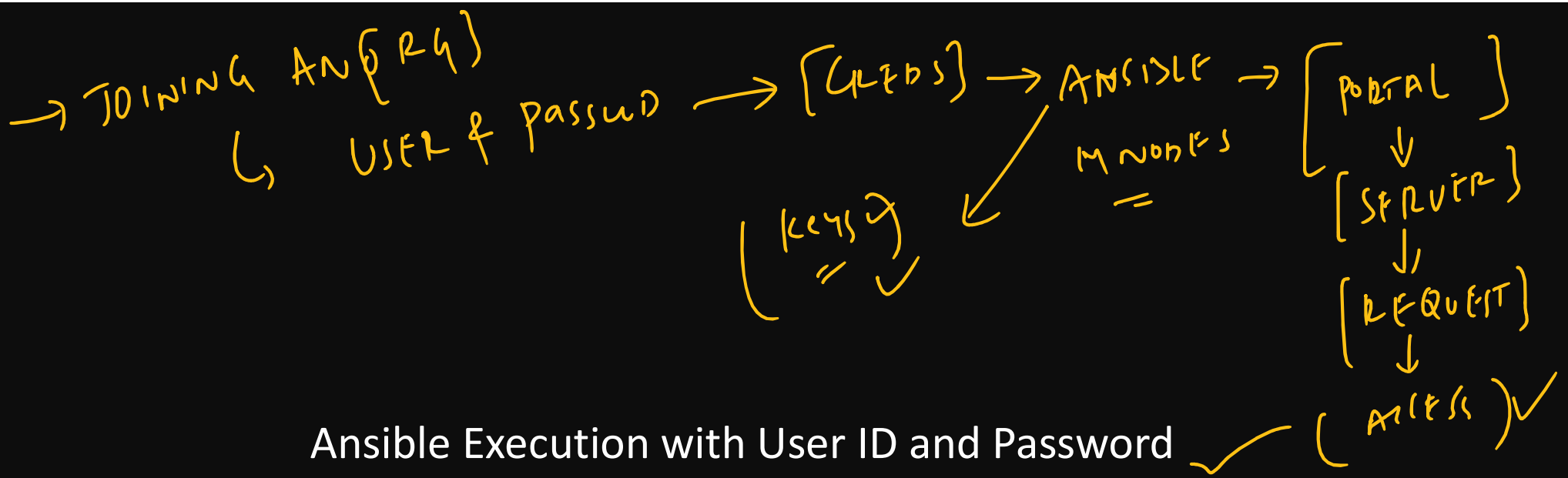
Ad-hoc Commands and Idempotency:

When you run an Ansible ad-hoc command to ensure a certain state, such as a file with specific content exists, Ansible checks the current state and only makes changes if necessary.

Output Interpretation:

The "changed": false output indicates no action was needed because the target state already existed.
If Ansible makes a change to reach the desired state, you will see "changed": true.

```
# Verify the existence and content of a file called demo.txt in /tmp on all managed nodes
ansible all -m ansible.builtin.stat -a "path=/tmp/demo.txt"

# Attempt to copy demo.txt again, expecting no change if the content matches
ansible all -m ansible.builtin.copy -a "src=/local/path/demo.txt dest=/tmp/demo.txt"
# Output should show "changed": false if the file content is the same.

# Modify the content of demo.txt locally
echo "New content" > /local/path/demo.txt

# Re-run the copy command after modifying the content of demo.txt
ansible all -m ansible.builtin.copy -a "src=/local/path/demo.txt dest=/tmp/demo.txt"
# Output should now show "changed": true, reflecting the update.

# Attempt to install nginx using the yum module, which is expected to show no change if already installed
ansible all -m ansible.builtin.yum -a "name=nginx state=present"
# Output should show "changed": false if nginx is already installed.
```



Ansible Execution with User ID and Password

Ansible Execution with User ID and Password
User Creation and Password Setting:

Assume you have three servers where you need to create a user jai.
Set a password for jai on all servers. Ensure it's the same across all to avoid confusion.

Enable Password Authentication:
Edit the sshd_config file on each server to set PasswordAuthentication to yes.
Restart the SSH service to apply changes: systemctl restart sshd.

Using Ansible with User Credentials:
Log into the Ansible control node with your user ID and password.
Verify Ansible installation with ansible --version.

Create a project directory and navigate into it.
Create an inventory file listing all servers.
Write an Ansible configuration file with the following content:
[defaults]
inventory = ./inventory
host_key_checking = false

Executing Ad-hoc Commands with Password Authentication:

Run the ansible command with the -k flag to prompt for a password.
Use the -u flag to specify the username if different from the current logged-in user.

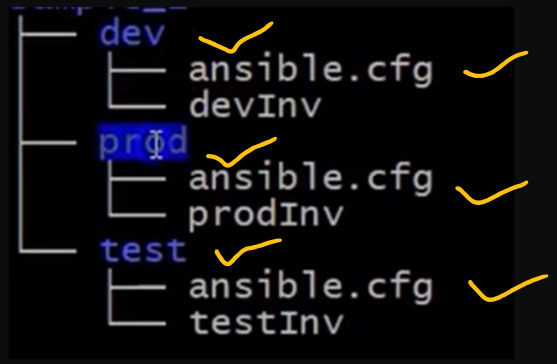
Ansible Inventory

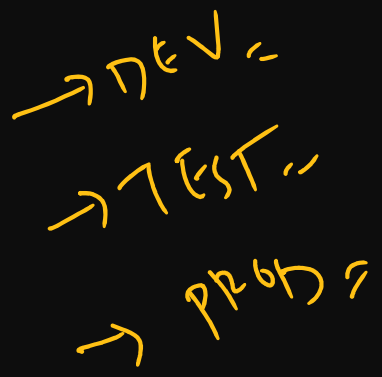
Managing Multiple Environments with Ansible Inventory Files

Separate Environment Directories:

Create a directory for each environment within your Ansible project.
Each directory contains a dedicated inventory file for that environment.

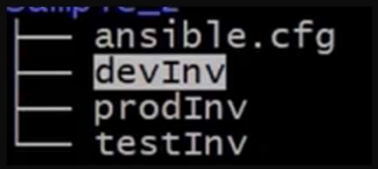
```
# Create directory structure for multiple environments
mkdir -p /homr/ansibleuser/{dev,test,prod}

# Navigate to the development environment directory
cd /path/to/dir1/dev

# Run an Ansible command using the development inventory
ansible all -m ansible.builtin.ping -i devInventory
```





Single Directory with Multiple Inventory Files

Maintain one ansible.cfg file in the main project directory.
Create separate inventory files within the main directory for each environment.



For a project using separate inventory files without changing directories:
cd /path/to/dir2

Use the -i flag to specify the production environment inventory
ansible all -m ansible.builtin.ping -i prodInventory

Unified Inventory File with Groups:

Use one inventory file, but define groups for each environment.

```
# Development Servers
[dev]
dev1.example.com
dev2.example.com

# Testing Servers
[test]
test1.example.com
test2.example.com

# Production Servers
[prod]
prod1.example.com
prod2.example.com
```

```
# Variables for Development Environment
[dev:vars]
ansible_ssh_user=devuser
ansible_ssh_pass=VAULT|dev_password_encrypted

# Variables for Testing Environment
[test:vars]
ansible_ssh_user=testuser
ansible_ssh_pass=VAULT|test_password_encrypted

# Variables for Production Environment
[prod:vars]
ansible_ssh_user=produser
ansible_ssh_pass=VAULT|prod_password_encrypted
```

```
# Run a command against all development servers
ansible dev -m ansible.builtin.ping

# Run a command against all testing servers
ansible test -m ansible.builtin.ping

# Run a command against all production servers
ansible prod -m ansible.builtin.ping

# List all hosts in the 'prod' environment group from the unified inventory
ansible prod --list
```

Ansible Inventory File Types

Introduction to Ansible Inventory

•**Purpose of Inventory in Ansible:** Ansible utilizes an inventory file to identify and group managed nodes (servers) for executing commands and playbooks.

•**Managed Nodes Information:** The inventory contains details like IP addresses or Fully Qualified Domain Names (FQDNs) of the servers Ansible will manage.

Types of Inventory Files

1.Static Inventory File:

1. **Definition:** A file where the information about managed nodes is manually updated by the user.
2. **Characteristics:**
    1. Requires manual entry of IPs or FQDNs.
    2. Commonly used and straightforward.
3. **Example:** A text file with a list of server IP addresses that you update when new servers are added to your infrastructure.

2.Dynamic Inventory File:

1. **Definition:** A file that automatically fetches managed nodes information using code or logic, without manual updates.
2. **Need for Dynamic Inventory:**
    1. Essential for cloud environments with autoscaling, where the number of servers fluctuates.
    2. Automatically updates the inventory with current servers, easing management in dynamic environments.
3. **Methods to Create Dynamic Inventory:**
    1. **Inventory Scripts:** Older method, involves scripts that dynamically generate inventory data.
    2. **Inventory Plugins:** Recommended modern approach, utilizing plugins specific to cloud providers or technologies.
4. **Example:** Using an AWS EC2 inventory plugin to dynamically fetch and update inventory with instances currently running in an AWS account.

Ansible Configuration File

Introduction to Ansible Configuration File (ansible.cfg)

Purpose: Defines default values for Ansible operations, such as inventory source, host key checking, and parallel execution limits (forks).

Location Priorities for ansible.cfg

Ansible looks for the configuration file in several locations, with the following order of priority:

ANSIBLE_CONFIG Environment Variable:

Highest priority. " export ANSIBLE_CONFIG=/home/ansibleuser/ansible.cfg"

If set, Ansible uses the configuration file specified by this environment variable.

Current Directory:

If ANSIBLE_CONFIG is not set, Ansible searches the current directory for an ansible.cfg file.

User Home Directory:

Looks for .ansible.cfg in the user's home directory if not found in the current directory.

Global Configuration (/etc/ansible/ansible.cfg):

Lowest priority.

Used if none of the above locations contain an ansible.cfg file.

Best Practices and Considerations

Project-Specific Configuration:

It's recommended to maintain a separate directory for your project, including a dedicated ansible.cfg file, to encapsulate project-specific settings.

Avoiding Global Configuration Conflicts:

Refrain from relying solely on the global /etc/ansible/ansible.cfg for project-specific settings to prevent conflicts or unintended behavior when running Ansible commands across different projects.

Security and Configuration Management:

Be mindful of who has access to your Ansible configurations, especially when working on shared systems, to avoid unauthorized changes or executions.

Ansible Facts

Introduction to Ansible Facts

Definition: Ansible facts are system and environment information that Ansible automatically gathers from managed nodes using the setup module.

Types of Information: Facts include details such as the operating system family, release version, and available RAM size on the managed nodes.

Format: Facts are stored in a dictionary or JSON format, making it easy to query and use within Ansible playbooks.

Gathering Facts with the Setup Module

Purpose: The setup module is used by Ansible to collect facts about managed nodes without the need for manual logic or scripts.

Execution: Running the `ansible.builtin.setup` module through ad-hoc commands or playbooks triggers the fact gathering process.

Examples:

To gather facts from all servers in your inventory: `ansible all -m ansible.builtin.setup`

To gather facts from a specific server: `ansible server1 -m ansible.builtin.setup`

Use Cases and Examples

Understanding Fact Structure:

When you execute the setup module, Ansible returns a vast amount of information about the managed node, including architecture, BIOS dates, distribution, and versions.

Filtering Facts:

In scenarios where only specific information is required, Ansible allows the use of filters to narrow down the facts presented.
Example: To display only the distribution of your managed nodes, you can use the filter parameter with the setup module:

```
ansible all -m ansible.builtin.setup -a "filter=ansible_distribution"
```

Fetching Distribution and Version:

- To further refine the output, you can request additional details such as the distribution version by adding more filters.
- **Example:** To get both distribution and version, you can modify the filter like so:

```
ansible all -m ansible.builtin.setup -a "filter=ansible_distribution,ansible_distribution_version"
```

Importance of Ansible Facts

- **Automation Efficiency:** By leveraging automatic fact gathering, Ansible simplifies the process of obtaining vital system information, reducing manual effort and potential errors.

- **Dynamic Decision Making:** Facts enable Ansible playbooks to make decisions based on the actual state and characteristics of managed nodes, such as conditionally executing tasks based on the OS version.

- **Customization and Filtering:** The ability to filter facts allows users to focus on relevant information, enhancing playbook clarity and performance.

Understanding the Output:

The output from the setup module is extensive, providing details such as architecture, BIOS date, distribution name, and version.

Key Information Examples:

ansible_architecture for CPU architecture.
ansible_distribution and ansible_distribution_version for the OS type and version.
ansible_memory_mb for memory size.

Filtering Facts:

You can refine the information returned by the setup module using the filter option to only show specific facts.
Syntax: ansible all -m ansible.builtin.setup -a 'filter=ansible_distribution*'
Example Usage: To display only the distribution name and version, you can specify a filter like ansible_distribution and ansible_distribution_version.

Basic Fact Gathering:

•Run an ad-hoc command to gather all facts about your managed nodes.

```
ansible all -m ansible.builtin.setup
```

Filtered Fact Gathering:

•To only display the distribution of your managed nodes:

```
ansible all -m ansible.builtin.setup -a 'filter=ansible_distribution’
```

Combining Filters:

•If you're interested in both the distribution and its version:

```
ansible all -m ansible.builtin.setup -a 'filter=ansible_distribution,ansible_distribution_version'
```

Key Takeaways

•**No Logic Needed:** The beauty of Ansible facts lies in their automatic collection. There’s no need to write custom logic or playbooks to fetch basic system information.

•**Customizability:** Through the use of filters, you can tailor the setup module's output to show exactly the information you need.

•**Utility in Playbooks:** Facts enable playbooks to be dynamic and responsive to the characteristics of the target nodes, allowing for more precise and condition-based task execution.

Create Custom Facts

Deploying Custom Facts

Create facts.d Directory:

Use Ansible's file module to create the /etc/ansible/facts.d directory on all managed nodes with necessary permissions.

Copy Fact Script to Managed Nodes:

Utilize the copy module to transfer the custom fact script to the /etc/ansible/facts.d directory on each managed node.

Utilizing Custom Facts

Running the Setup Module: Execute the setup module via an Ansible ad-hoc command or within a playbook. This action will gather default facts along with your newly created custom facts.

Accessing Custom Facts: Custom facts can be accessed under the ansible_local key when reviewing the gathered facts.

Filtering Facts

To simplify the output and focus on custom facts, you can use the filter argument with the setup module. This approach lets you isolate and view only the facts of interest.

Creating a custom fact script (java.fact) for Java version: vi java.fact

```
#!/bin/bash
echo "{\"java_version\": \"$(java -version 2>&1 | awk -F '"' '/version/ {print $2}')\"}“
```

Deploying and executing the script:

Create the directory: ansible all -m file -a "path=/etc/ansible/facts.d state=directory" –b -K
Copy the script: ansible all -m copy -a "src=java.fact dest=/etc/ansible/facts.d/java.fact mode=755" –b –K

Gathering facts with the setup module:
ansible all -m setup -a "filter=ansible_local"

Custom facts in Ansible allow for greater flexibility and control over the information gathered from managed nodes. By following these steps, you can easily extend Ansible's default fact-gathering functionality to include any specific data required for your automation tasks. This capability enhances decision-making and task execution based on the unique characteristics of your infrastructure.

Introduction to Playbooks

Introduction to Playbooks

we will dive into the world of Ansible playbooks, a powerful feature of Ansible used for automating tasks across managed nodes. We'll start with a basic understanding of playbooks, compare them with ad-hoc commands, and illustrate their advantages with examples.

Overview of Ansible Automation Methods

- **Ansible Ad-Hoc Commands:** Quick and simple commands for performing tasks on managed nodes.
- **Ansible Playbooks:** More complex and structured scripts written in YAML for automation of multiple tasks.

- Installing an HTTPD Package
- Objective: Install the HTTPD package on managed nodes across different environments (dev, test, prod).

- Method 1: Ad-Hoc Commands

- Example Command: ansible <dev_group> -m ansible.builtin.yum -a "name=httpd state=present" -b
- Limitation: Not suitable for repetitive tasks across different environments or when conditional logic is needed.

Introducing Ansible Playbooks

Playbooks: Allow the inclusion of logic, conditions, loops, and variables, making them more powerful and flexible than ad-hoc commands or shell scripts.

Example Use Case: Install HTTPD package on managed nodes, handling different OS families with appropriate package managers.

Basic Structure of an Ansible Playbook
Playbooks Start with --- (YAML Format)
Plays: A playbook contains one or more plays. Each play targets a group of hosts and outlines tasks to be executed.
Tasks: Defined operations using modules, executed on targeted hosts.
Features: Supports variables, conditions, loops, and templates for dynamic and conditional task execution.

```yaml
---
- name: Install HTTPD on Web Servers
  hosts: "{{ env }}"
  become: yes
  tasks:
    - name: Install HTTPD
      package:
        name: httpd
        state: present
      when: ansible_os_family == "RedHat"
    - name: Install Apache
      package:
        name: apache2
        state: present
      when: ansible_os_family == "Debian"
```

This playbook dynamically selects the package manager based on the OS family of the managed node.

Execution of a Playbook
Command: ansible-playbook webServer.yaml -e env=dev

Playbooks are reusable and can be tailored for different environments with command-line arguments.

Key Takeaways
Flexibility and Reusability: Playbooks offer a structured, reusable format for automating complex tasks across multiple environments.

Conditional Logic: Ability to include logic and conditions caters to complex scenarios, making automation more effective.

Playbook Syntax: Playbooks are written in YAML, starting with ---. Plays and tasks are clearly structured for readability and maintainability.

Introduction to Ansible Playbooks

•**Key Point:** Ansible playbooks are automation scripts written in YAML format for configuring, deploying, and managing servers.

Understanding Ansible Ad-Hoc Commands

•**Overview:** Before diving into playbooks, it's crucial to understand Ansible ad-hoc commands.

•**Example:** Using the debug module to print a message.

```
ansible all -m ansible.builtin.debug -a 'msg="Welcome to Ansible playbooks“’
```

Insight: The debug module is helpful for printing messages or variables, essential for troubleshooting and confirming playbook actions.

Step-by-Step Guide to Your First Playbook

Playbook Basics:

File extension: .yml or .yaml
Must start with three hyphens ---
Contains one or more plays

Play Structure:

Each play must have targets (hosts) and tasks sections.
Tasks are the operations or actions you want to perform.

Writing the Playbook:

Example task: Displaying a welcome message using the ansible.builtin.debug module.
Use a reference or template when starting out.

Playbook Development Steps:

Create a file, e.g., display_msg.yml.
Start with three hyphens.
Define a single play with a host and tasks.

Tasks and Logic:

Define the target hosts.
Add tasks using appropriate modules and arguments.

Execution and Output:

Execute the playbook using ansible-playbook display_msg.yml.
Understand the output, including task execution on designated hosts.

Advanced Playbook Features

Gathering Facts:

By default, Ansible gathers system facts before executing tasks.
Control this behavior with gather_facts: false.

Indentation and Syntax:

Avoid using tabs; stick to spaces for indentation.
YAML syntax requires careful attention to indentation and structure.

Adding Descriptive Names:

Optionally add names to plays and tasks for clarity.
Names appear in playbook execution output, aiding in readability.

Practical Tips for Writing Playbooks

- **Start Simple:** Begin with basic tasks to understand how Ansible modules and arguments work together.
- **Use References:** Look at existing playbooks for structure and syntax guidance.
- **Experiment:** Modify playbooks and observe how changes affect execution and output.
- **Indentation Matters:** YAML is sensitive to indentation; use a consistent method to avoid errors.

```
---
- name: Display Welcome Message
  hosts: all
  gather_facts: false
  tasks:
    - name: Print Welcome Message
      ansible.builtin.debug:
        msg: "Welcome to Ansible Playbooks"
    - name: Print Second Message
      ansible.builtin.debug:
        msg: "Exploring Ansible Basics"
```

Additional Resources

- **Ansible Documentation:** For in-depth understanding and more examples.
- **Online Communities:** Forums and discussion boards for troubleshooting and tips.

Executing Playbooks on Ansible Controller Node

```
---
- name: Example playbook on localhost
  hosts: localhost
  connection: local
  gather_facts: false
  tasks:
    - name: Print a message
      ansible.builtin.debug:
        msg: "This is executed on the localhost."
```

Advantages of Local Execution

- **Fast Setup:** Ideal for quick testing of playbook syntax or functionality without external dependencies.
- **Learning and Testing:** Provides a safe environment to learn Ansible features or debug playbooks before deploying them to production environments.
- **No SSH Needed:** Simplifies configurations by removing the need for SSH keys or SSH connections setup for the localhost.

Syntax Validation in Ansible

- **Importance of Syntax Validation:** Ensures playbooks are free from syntactical errors before execution.
- Helps maintain the playbook's quality and adherence to best practices.
- Saves time by identifying errors early in the development process.

Method 1: Using ansible-playbook --syntax-check

Command Overview:
ansible-playbook <playbook-name.yml> --syntax-check
This command checks the syntax of the playbook without executing any tasks.

Example and Outcome:
Correct Syntax:
If the playbook has no syntax errors, the command will not return any errors.
Syntax Error Detected:
Example command: ansible-playbook playbook.yml --syntax-check
Possible error: Missing spaces or indentation issues. The command will pinpoint the location of the syntax error for correction.

Benefits:
Quick and straightforward way to validate playbook syntax.
Integrated into Ansible, no additional installation required.

Method 2: Using ansible-lint
Tool Overview:

ansible-lint is a more comprehensive tool that checks for syntax errors, best practices, and potential runtime issues.
It is not included with Ansible by default and requires separate installation via pip: pip install ansible-lint.
Running ansible-lint:

Command: ansible-lint <playbook-name.yml>
The tool not only checks for syntax errors but also enforces best practice guidelines, such as naming conventions and correct indentation.
Examples and Corrections:

Naming Convention Warning:
Warnings about names not starting with an uppercase letter. Correction involves capitalizing the first letter of names.
Indentation Warning:
Incorrect indentation levels. Correction involves adjusting spaces to match the expected indentation.

Validating Ansible playbook syntax is a critical step in the playbook development process. It ensures that playbooks are error-free, adhere to best practices, and are maintainable. By utilizing ansible-playbook --syntax-check for a quick syntax check and ansible-lint for a comprehensive analysis, developers can significantly improve the quality of their Ansible projects. Regular and consistent syntax validation is not just a good practice; it's a cornerstone of efficient Ansible playbook development.

Working with Variables

Introduction to Ansible Variables

•**Purpose of Variables:** In Ansible, like in other programming or scripting languages, variables are used to store values that can be reused throughout your playbooks.

•**Types of Variables:**

- Custom Variables
- Registered Variables
- Facts
- Inventory Variables (Host Vars and Group Vars)
- Default Variables

Custom Variables

•**Definition and Usage:** Custom variables are user-defined and can be specified directly in playbooks or in separate files for organization.

•**YAML Syntax:** Variables in Ansible playbooks (which use YAML format) are defined with a key-value pair representation. For example:

```
x: 2
y: 4.5
myName: "Your Name"
```

Scalar Variables: These are basic variables that hold single values, such as numbers or strings.

Variable Naming Rules

- Variables names should consist of letters, numbers, and underscores.
- Must start with a letter, which can be uppercase or lowercase.

- Defining Custom Variables in Playbooks
- Use the vars section in your playbook to define custom variables.
- Example of defining variables in a playbook:

```
---
- hosts: localhost
  vars:
    x: 2
    y: 4.5
    myName: “JaiTIF"
  tasks:
    - name: Display variable x
      ansible.builtin.debug:
        msg: "x value is {{ x }}"
```

Displaying Variable Values

Use the debug module with the msg argument to display variable values.

Variable values can be referenced within the msg using Jinja2 templating syntax, encapsulated in double curly braces {{ }}.

```
tasks:
  - name: Display x variable value
    ansible.builtin.debug:
      msg: "x value is {{ x }}"

  - name: Display y variable value
    ansible.builtin.debug:
      msg: "y value is {{ y }}"

  - name: Display myName variable value
    ansible.builtin.debug:
      msg: "myName value is {{ myName }}"
```

```
tasks:
  - name: Display variables
    ansible.builtin.debug:
      msg:
        - "x value is {{ x }}"
        - "y value is {{ y }}"
        - "myName value is {{ myName }}"
```

```
tasks:
  - name: Display all variables in one task
    debug:
      msg: "x value is {{ x }}, y value is {{ y }}, myName value is {{ myName }}"
```

Data Types of Variables

Introduction to Variables and Data Types in Ansible

Variables: Used to store values or data in Ansible.

Data Types: Different types of data can be stored in variables, including:

Integer: Whole numbers, e.g., 4.

Float: Decimal numbers, e.g., 5.6.

String: Textual data, e.g., "Hello World". Strings with spaces need to be enclosed in quotes.

Boolean: True or False values, often used in conditional expressions. true/yes and false/no are interchangeable.

Defining and Displaying Variable Values

This playbook defines variables of different data types and uses the debug module to display their values.

```
---
- name: Display Variable Types
  hosts: localhost
  gather_facts: false
  vars:
    a: 4
    b: 5.6
    c: Ansible
    d: "Hello World"
    e: true
    f: false
  tasks:
    - name: Display Integer Variable
      ansible.builtin.debug:
        msg: "The value of 'a' is {{ a }}"
    - name: Display Float Variable
      ansible.builtin.debug:
        msg: "The value of 'b' is {{ b }}"
    - name: Display String Variable
      ansible.builtin.debug:
        msg: "The value of 'c' is {{ c }}"
    - name: Display String with Spaces Variable
      ansible.builtin.debug:
        msg: "The value of 'd' is '{{ d }}'"
    - name: Display Boolean Variable (True)
      ansible.builtin.debug:
        msg: "The value of 'e' is {{ e }}"
    - name: Display Boolean Variable (False)
      ansible.builtin.debug:
        msg: "The value of 'f' is {{ f }}"
```

Identifying Data Types with type_debug

This playbook builds upon the first by not only displaying the variable values but also using type_debug to display each variable's data type.

```
---
- name: Identify and Display Variable Data Types
  hosts: localhost
  gather_facts: false
  vars:
    a: 4
    b: 5.6
    c: Ansible
    d: "Hello World"
    e: true
    f: false

  tasks:
    - name: Display Variable 'a' Value and Data Type
      ansible.builtin.debug:
        msg:
          - "The value of 'a' is {{ a }}"
          - "Its data type is {{ a | ansible.builtin.type_debug }}"

    - name: Display Variable 'b' Value and Data Type
      ansible.builtin.debug:
        msg:
          - "The value of 'b' is {{ b }}"
          - "Its data type is {{ b | ansible.builtin.type_debug }}"

    - name: Display Variable 'c' Value and Data Type
      ansible.builtin.debug:
        msg:
          - "The value of 'c' is {{ c }}"
          - "Its data type is {{ c | ansible.builtin.type_debug }}"

    - name: Display Variable 'd' Value and Data Type
      ansible.builtin.debug:
        msg:
          - "The value of 'd' is '{{ d }}'"
          - "Its data type is {{ d | ansible.builtin.type_debug }}"

    - name: Display Variable 'e' Value and Data Type
      ansible.builtin.debug:
        msg:
          - "The value of 'e' is {{ e }}"
          - "Its data type is {{ e | ansible.builtin.type_debug }}"

    - name: Display Variable 'f' Value and Data Type
      ansible.builtin.debug:
        msg:
          - "The value of 'f' is {{ f }}"
          - "Its data type is {{ f | ansible.builtin.type_debug }}"
```

Data Structures in Ansible

Introduction to Data Structures in Ansible

•**Variables in Ansible:** Variables can store information that can be used in your playbooks. Scalar variables store a single value.

•**Data Structures:** When you need to store multiple values, you use data structures. Ansible supports two primary types:

- **List (Sequence):** An ordered collection of items.
- **Dictionary (Map):** A collection of key-value pairs.

Defining Data Structures

Starting a Playbook:

Begin with three hyphens (---) to indicate the start of a YAML file, which Ansible playbooks are written in.
Define a play with a list item starting with a hyphen.
Using vars for Variable Definitions:

The vars section is where you define all your variables, including scalar, list, and dictionary variables.

Scalar Variable Example:

```
vars:
  myValue: 4
```

myValue is a scalar variable holding a single value, 4.

List (Sequence) Variable Example

```
myList:
  - first value
  - second value
  - third value
```

myList is a list variable. Lists are defined with each item starting with a hyphen

Dictionary (Map) Variable Example:

```
myDictionary:
  myFirst: something
  mySecond: something else
```

myDictionary is a dictionary variable. Dictionaries are defined with key-value pairs.

Displaying Variable Values with Debug Module

To display variables, use the debug module in the tasks section.
Example task to display all variable values:

```
tasks:
  - name: Display variables
    ansible.builtin.debug:
      msg: "{{ myValue }} {{ myList }} {{ myDictionary }}"
```

Disable default gathering of facts with gather_facts: false if not needed to speed up playbook execution.

Syntax Variations

Lists and dictionaries can be defined in multiple ways:

List Alternative Syntax:
Inline with square brackets: [3, 7, 8]

Dictionary Alternative Syntax:
Inline with curly braces: {"key1": "value1", "key2": "value2"}

Using Scalar Variables, Lists, and Dictionaries

```
---
- name: Demonstrate Data Structures
  hosts: localhost
  gather_facts: false
  vars:
    # Scalar variable
    myValue: 4

    # List/Sequence variable
    myList:
      - first value
      - second value
      - third value

    # Dictionary/Map variable
    myDictionary:
      myFirst: something
      mySecond: something else

  tasks:
    - name: Display all variables
      ansible.builtin.debug:
        msg: "Scalar: {{ myValue }}, List: {{ myList }}, Dictionary: {{ myDictionary }}"
```

Alternative Syntax for Lists and Dictionaries

```
---
- name: Demonstrate Alternative Data Structures Syntax
  hosts: localhost
  gather_facts: false
  vars:
    # Scalar variable
    myValue: 7

    # List/Sequence variable with alternative syntax
    myList: [3, 7, 8]

    # Dictionary/Map variable with alternative syntax
    myDictionary: {"myFirst": "another thing", "mySecond": "yet another thing"}

  tasks:
    - name: Display variables using alternative syntax
      ansible.builtin.debug:
        msg: "Scalar: {{ myValue }}, List: {{ myList }}, Dictionary: {{ myDictionary }}"
```

Read Variable Values from a File

Introduction to Variables in Ansible

•Variables are crucial for dynamic Ansible playbooks.
•Traditionally, variables can be defined directly inside playbooks.

Advantages of External Variable Files

•**Enhanced Organization:** Separating variables from playbook logic for cleaner code.
•**Reusability:** Easily reuse variable files across different playbooks.
•**Environment Specific Configuration:** Manage variables for different environments by using separate files (e.g., dev, test, prod).

Defining Variables in External Files

Example:

```
---
myVar: some value
myList: [3,4,5,6]
```

JSON Files:

Use curly braces ({}) to contain the object.
Key-value pairs represent variable names and their values.

```
{
 "x": 123,
 "myDictionary": {
  "first": 1,
  "second": 2
 }
}
```

Accessing External Variables in Playbooks

Use the vars_files section to specify external files.

List the paths to your YAML or JSON files containing variables.

Playbook Example:

```
---
- hosts: localhost
  gather_facts: false
  vars_files:
    - varsWithYaml.yml
    - varsWithJson.json
  tasks:
    - name: Display variables
      ansible.builtin.debug:
        msg: "myVar value: {{ myVar }}, myDictionary value: {{ myDictionary }}"
```

Practical Tips

Debugging: Use the ansible.builtin.debug module to print variable values, aiding in troubleshooting.

Skipping Default Facts Gathering: Set gather_facts: false to speed up playbook execution if gathering facts is unnecessary

Creating the Variable Files

YAML Variable File: varsWithYaml.yml

```
---
myVar: some value
myList: [3, 4, 5, 6]
```

This YAML file defines a simple string variable myVar and a list variable myList.

JSON Variable File: varsWithJson.json

```
{
  "x": 123,
  "myDictionary": {
    "first": 1,
    "second": 2
  }
}
```

This JSON file defines an integer variable x and a dictionary myDictionary with two keys.

```yaml
---
- hosts: localhost
  gather_facts: false
  vars_files:
    - varsWithYaml.yml
    - varsWithJson.json
  tasks:
    - name: Display variable 'myVar' from YAML
      ansible.builtin.debug:
        msg: "myVar value: {{ myVar }}"

    - name: Display list 'myList' from YAML
      ansible.builtin.debug:
        msg: "myList values: {{ myList }}"

    - name: Display variable 'x' from JSON
      ansible.builtin.debug:
        msg: "x value: {{ x }}"

    - name: Display dictionary 'myDictionary' from JSON
      ansible.builtin.debug:
        msg: |
          myDictionary values:
          first: {{ myDictionary.first }},
          second: {{ myDictionary.second }}
```

Ansible Playbook: readVarsFromFiles.yml

This playbook demonstrates how to include the external variable files and access their variables:

Expected Output:

- The playbook runs tasks that print the values of myVar, myList, x, and myDictionary using the debug module.
- For myVar, you should see "some value" as output.
- For myList, the output will be the list [3, 4, 5, 6].
- For x, expect to see "123".
- For myDictionary, it should print the dictionary contents, showing "first: 1, second: 2".

Passing variables from command line

Introduction to Variables in Ansible

- Variables can be defined inside playbooks or in separate files (JSON/YAML).
- Accessing variables in playbooks enhances flexibility and reusability.

Command-Line Arguments for Variables

- Besides defining variables in playbooks or files, Ansible allows passing variables directly from the command line, known as command-line arguments.

display_var.yml

```
---
- name: Display Variables Example
  hosts: localhost
  tasks:
    - name: Display the value of 'x'
      debug:
        msg: "The value of x is: {{ x }}"

    - name: Display the value of 'y'
      debug:
        msg: "The value of y is: {{ y }}"
```

How to Pass Variables as Command-Line Arguments

Basic Syntax

Syntax: `ansible-playbook playbook.yml --extra-vars "variable=value"`

Variables passed from the command line are treated as strings.

Even numeric values are considered strings when passed this way.

```
# Passing a single string variable
ansible-playbook display_vars.yml --extra-vars "x=hi"

# Passing multiple string variables
ansible-playbook display_vars.yml --extra-vars "x=hi y=bye"

# Using the shortcut `-e` for --extra-vars with multiple variables
ansible-playbook display_vars.yml -e "x=hi y=bye"

# Passing a numeric value and a list
ansible-playbook display_vars.yml -e "{ 'x': 5, 'y': [1, 2, 3] }"

# Passing complex data types: a string and a map (dictionary)
ansible-playbook display_vars.yml -e "{ 'x': 'hello', 'y': {'key1': 'value1', 'key2': 'value2'} }"
```

Introduction to vars_prompt in Ansible

Purpose: vars_prompt is used to dynamically read variable values at runtime, akin to the read command in shell scripting or the input statement in Python scripting.

Syntax Overview: To use vars_prompt, include it in your playbook with name and prompt keys.

Key Components

name: Specifies the variable name that you want to define.

prompt: Provides a message displayed on the command line when executing the playbook, guiding the user on what input is needed.

Example Playbook: Reading a Single Variable

Playbook Name: vars_promptUsage.yml

Structure:

Start with YAML syntax initiation (---).

Define the host

Include a vars_prompt section before the tasks.

Define tasks using the debug module to display the variable's value.

```
---
- hosts: localhost
  gather_facts: false
  vars_prompt:
    - name: x
      prompt: "Enter your x value"
  tasks:
    - name: Display the value of x
      ansible.builtin.debug:
        msg: "The value of x is {{ x }}"
```

Execution and Output

Run the playbook using ansible-playbook vars_promptUsage.yml.
Input the value for x when prompted. (Note: The input won't display on the command line for security reasons.)
The playbook displays the value of x using the debug module.

Reading Multiple Variables

Modification for Multiple Inputs: To read multiple variables, extend the vars_prompt section with additional entries.

```
vars_prompt:
  - name: x
    prompt: "Enter x"
  - name: y
    prompt: "Enter y"
```

Example with Multiple Variables

```
---
- hosts: localhost
  gather_facts: false
  vars_prompt:
    - name: x
      prompt: "Enter x"
    - name: y
      prompt: "Enter y"
  tasks:
    - name: Display the values of x and y
      ansible.builtin.debug:
        msg: "The values are x: {{ x }} and y: {{ y }}"
```

```
---
- hosts: localhost
  gather_facts: false
  vars:
    # Static variable defined directly in the playbook
    greeting: "Welcome to Ansible"

  vars_prompt:
    # Dynamic variable prompted at runtime
    - name: user_name
      prompt: "What is your name?"
      private: no  # This makes the typed response visible in the
command line

  tasks:
    - name: Display a personalized welcome message
      ansible.builtin.debug:
        msg: "{{ greeting }}, {{ user_name }}!"
```

Combining vars and vars_prompt

- The vars section allows you to define variables that remain constant each time the playbook is executed.
- The vars_prompt section is used for variables whose values you want to specify at runtime, making your playbooks interactive and flexible.
- This approach is particularly useful for scenarios where certain playbook parameters need to be dynamic or user-defined, while others can be predefined for consistency and efficiency.

Introduction to Variable Management in Ansible

Overview of Variables in Ansible: Explanation of various methods to define variables (vars section, vars_prompt, and vars_files), emphasizing that variables defined through these methods are available to all nodes in playbooks.

Understanding host_vars and group_vars

Execution Context: Introduction using a simple playbook with variables x and y, demonstrating their scope when executed on localhost and then on a dev group, illustrating how variables apply to nodes within that group.

host_vars

Purpose and Usage: The need for host_vars to assign unique variable values to individual hosts within a group.
Example Configuration: Shows how to define host_vars directly in the inventory for specific nodes and alternatively, by creating files named after the hostnames within a host_vars directory. This allows for host-specific variable values outside of the playbook or inventory file.

Introduction to group_vars

Group-specific Variables: Similar to host_vars, but for defining variables that apply to all hosts within a given group.
Implementation Options: Discusses defining group_vars within the inventory and through a directory structure, mirroring the host_vars approach but at the group level.

host_vars

```
---
- name: Demonstrate host_vars
  hosts: dev
  tasks:
    - name: Print variables x and y
      debug:
        msg: "x={{ x }}, y={{ y }}"
```

Create Inventory File
Create or edit your inventory file named inventory.

```
[dev]
node1 ansible_host=192.168.1.101
node2 ansible_host=192.168.1.102
```

[OR]

```
mkdir host_vars
```

Within host_vars, create files named after each host, node1 and node2, and define variables x and y in each.

```
# For node1 (host_vars/node1)
---
x: 5
y: 9


# For node2 (host_vars/node2)
---
x: 2
y: 3
```

```
ansible-playbook -i inventory playbook_hostvars.yml
```

group_vars

```
---
- name: Demonstrate group_vars
  hosts: dev
  tasks:
    - name: Print variables x and y
      debug:
        msg: "x={{ x }}, y={{ y }}"
```

Edit or Create the Inventory File

```
[dev]
node1 ansible_host=192.168.1.101
node2 ansible_host=192.168.1.102

[dev:vars]
x=11
y=22
```

```
mkdir group_vars
```

Within group_vars, create a file named after the group, dev, and define common variables.

```
# For the dev group (group_vars/dev)
---
x: 11
y: 22
```

```
ansible-playbook -i inventory playbook_groupvars.yml
```

Variable Precedence

An exploration of how Ansible determines the priority of variables from different sources.
Insight: host_vars override group_vars, and direct playbook variables (vars section) have the highest priority.

Documentation Reference: The official Ansible documentation provides a detailed precedence order, important for understanding how variable values are chosen.

Order of precedence from lowest to highest
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html

command line values (for example, -u my_user, these are not variables)
role defaults (defined in role/defaults/main.yml) 1
inventory file or script group vars 2
inventory group_vars/all 3
playbook group_vars/all 3
inventory group_vars/* 3
playbook group_vars/* 3
inventory file or script host vars 2
inventory host_vars/* 3
playbook host_vars/* 3
host facts / cached set_facts 4
play vars
play vars_prompt
play vars_files
role vars (defined in role/vars/main.yml)
block vars (only for tasks in block)
task vars (only for the task)
include_vars
set_facts / registered vars
role (and include_role) params
include params
extra vars (for example, -e "user=my_user")(always win precedence)

Passing Var in Debug Module

```
# debug_module_with_var.yml
---
- name: Demonstrate debug module with msg and var
  hosts: localhost
  vars:
    x: 8
    y: 7
  tasks:
    - name: Display a message with msg
      debug:
        msg: "This is a test message using msg."

    - name: Display the value of 'x' using var
      debug:
        var: x

    - name: Display multiple variables 'x' and 'y' using msg
      debug:
        msg: "x value: {{ x }}, y value: {{ y }}"

    - name: Display 'x' and 'y' using var (in a single task)
      debug:
        var: x,y
```

What Happens in the Playbook:

The first task uses msg to display a custom message.
The second task shows how to use var to output the value of a single variable (x) without needing curly braces or quotes.
The third task again uses msg to display values of multiple variables (x and y), demonstrating custom formatting.
The fourth task uses var to display multiple variables at once, but note that you can't format the output as freely as with msg.

Comments and Observations

Simplicity of var: You'll notice that using var makes it very straightforward to display variable values without additional formatting or the need for braces.
Flexibility of msg: The msg argument allows for more complex messages, combining text and variable values in a custom format.
Limitations with Multiple Variables in var: While var can display multiple variables, it doesn't allow for the custom formatting that msg does.

Register and setfacts Variables

Introduction to Ansible Register Variables

•**Definition:** Register variables in Ansible are special variables that store the output of a task. They're used to capture results that you can use later in your playbook.

•**Purpose:** Allows for conditional execution of tasks based on the outcomes of previous tasks, data manipulation, and information retrieval from managed nodes.

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Execute uname command
      ansible.builtin.command:
        cmd: uname
      register: unameOutput

    - name: Display uname command output
      ansible.builtin.debug:
        var: unameOutput.stdout
```

```
---
- hosts: servers
  tasks:
    - name: Check disk space on root partition
      ansible.builtin.shell:
        cmd: df -h / | tail -n 1 | awk '{print $5}'
      register: diskSpace

    - name: Alert if disk space usage is over 80%
      ansible.builtin.debug:
        msg: "WARNING: Disk space usage on root partition is over 80%"
      when: diskSpace.stdout | replace("%", "") | int > 80
```

This playbook checks the disk space usage on the root partition and displays a warning if the usage exceeds 80%. This is particularly useful for system administrators who need to monitor disk space to prevent service disruptions.

Understanding set_fact in Ansible

Introduction to set_fact

Purpose: set_fact is a module used within Ansible playbooks to define new variables or modify existing variables during the execution of a playbook.

Flexibility: Allows for dynamic adjustments of variables based on conditions or results of previous tasks.

Basic Usage of set_fact

Scenario: You start with a variable x with an initial value and aim to modify it as the playbook runs.

Example:

Initial variable definition: x: 5

Modification using set_fact: Change x to 8 and introduce a new variable y with the value "ansible playbooks".

```
- hosts: localhost
  tasks:
    - name: Display initial value of x
      ansible.builtin.debug:
        msg: "The initial value of x is {{ x }}"

    - name: Modify x and introduce y
      ansible.builtin.set_fact:
        x: 2024
        y: “Jai playbooks"

    - name: Display new values of x and y
      ansible.builtin.debug:
        msg:
          - "The new value of x is {{ x }}"
          - "The value of the new variable y is {{ y }}"
```

Dynamic Variable Management: set_fact provides powerful flexibility in playbook design, allowing for dynamic variable adjustments based on runtime conditions.

Use Cases: Ideal for scenarios where the playbook's flow or decisions depend on the outcomes of previous tasks, environment states, or user inputs.

Using Operations on Variables

Arithmetic Operators in Ansible:

Operators Overview:

Addition (+)
Subtraction (-)
Multiplication (*)
Division (/)
Floor Division (//)
Modulo (%)
Power/Exponent (**)
Using Variables:

Define two variables, a and b, which will be used in our arithmetic operations.

```
- hosts: localhost
  vars:
    a: 10
    b: 5
  tasks:
    - name: Display arithmetic operations
      debug:
        msg:
          - "Addition of a and b is: {{ a + b }}"
          - "Subtraction of a from b is: {{ a - b }}"
          - "Multiplication of a and b is: {{ a * b }}"
          - "Division of a by b is: {{ a / b }}"
          - "Floor division of a by b is: {{ a // b }}"
          - "Modulo of a and b is: {{ a % b }}"
          - "Power of a to b is: {{ a ** b }}"
```

Key Points and Tips:

Flexibility with Data Types:

- Ansible's arithmetic operations work with integers and floats. Feel free to experiment with different data types to see how Ansible handles them.
- Remember, for arithmetic operations in Ansible playbooks, encapsulate your expressions within {{ }}
- These operations can be incredibly useful in real-world scenarios, such as calculating disk space requirements, splitting tasks among a dynamic number of hosts, or defining timeouts and intervals dynamically based on variables.

Filters and Methods in Ansible

Introduction to Filters and Methods in Ansible
Purpose: Filters and methods are used to perform operations on data within Ansible playbooks. Examples include manipulating strings (e.g., changing case) and more complex data processing.

Key Difference:
Filters are applied using a pipeline (|) syntax and are specific to Ansible.
Methods follow a dot (.) notation, invoking Python methods directly on variables or data.

```
---
# Start of the playbook
- hosts: localhost       # Target localhost for the operation
  gather_facts: false    # Disable gathering facts to speed up execution

  # Define a variable 'a' with the value 'ansible'
  vars:
    a: "ansible"

  # Define tasks to be executed
  tasks:
    # First task: Display the original value of 'a'
    - name: Display variable value
      debug:
        msg: "a value is: {{ a }}"

    # Second task: Convert 'a' to uppercase using the 'upper' filter
    - name: Convert string to uppercase with filter
      debug:
        msg: "a in uppercase (filter): {{ a | upper }}"

    # Third task: Convert 'a' to uppercase using the 'upper' method
(Python style)
    - name: Convert string to uppercase with method
      debug:
        msg: "a in uppercase (method): {{ a.upper() }}"
```

- **Filters** are more Ansible-centric, providing a wide range of built-in options for data manipulation directly within your playbooks.

- **Methods** leverage Python's built-in capabilities, offering a familiar approach for those with Python experience.

Filters in Ansible

```yaml
---
- name: Demonstrate Ansible Filters
  hosts: localhost
  gather_facts: false
  vars:
    text: "ansible"
    number_list: [4, 2, 3, 1]
    string_list: ["zebra", "octopus", "giraffe", "antelope"]
    mixed_list: ["ansible", 2, "2", 3]
    dict_variable: {'c': 3, 'a': 1, 'b': 2}
    undefined_variable: null
    date_string: "2022-01-01"

  tasks:
    - name: Text Manipulation Filters
      debug:
        msg:
          - "{{ text | upper }}"
          - "{{ text | lower }}"
          - "{{ 'hello world' | capitalize }}"
          - "{{ 'hello world' | title }}"

    - name: Numeric and List Filters
      debug:
        msg:
          - "{{ number_list | random }}"
          - "{{ number_list | min }}"
          - "{{ number_list | max }}"
          - "{{ string_list | join(',') }}"
          - "{{ string_list | sort }}"
          - "{{ mixed_list | unique }}"

    - name: Dictionary Filters
      debug:
        msg:
          - "{{ dict_variable | dictsort }}"

    - name: Conditional Filters
      debug:
        msg:
          - "{{ undefined_variable | default('This is a default value because the variable was undefined.') }}"
          - "{{ 'true' | bool }}"

    - name: Date and Time Filters
      debug:
        msg:
          - "{{ date_string | to_datetime }}"
          - "{{ ansible_date_time.iso8601 | default(omit) | strftime('%Y-%m-%d %H:%M:%S') if ansible_date_time is defined else 'ansible_date_time is not defined' }}"

    - name: Advanced Filters
      debug:
        msg:
          - "{{ 'hello world' | regex_replace('world', 'ansible') }}"
          - "{{ 'ansible' | hash('sha256') }}"
```

Comparison Operators

Introduction to Comparison Operators

Purpose: Comparison operators are used to compare two values. The outcome of such a comparison is a boolean value, either True or False.

Application: They can be applied to both numbers (integers or floats) and strings.

Result: The result of using comparison operators is always a boolean data type.

Comparison Operators for Numbers

Operators:

Equal to (==)

Not equal to (!=)

Greater than (>)

Less than (<)

Greater than or equal to (>=)

Less than or equal to (<=)

Comparison Operators for Strings

Operators:

Equal to (==)

Not equal to (!=)

Syntax for Using Comparison Operators in Ansible

Variables: To compare variables in Ansible, place them inside curly braces ({{ }}) and use a comparison operator between them.

Direct Data:

For numbers, directly place them within the comparison syntax.

For strings, ensure they are within quotations.

```
---
- name: Demonstrate Comparison Operators
  hosts: localhost
  vars:
    x: 6
    y: 10
    p: "Ansible"
    q: "Automation"
    r: "Automation"
  tasks:
    - name: Display comparison results for numbers
      debug:
        msg: "X == Y is {{ x == y }}; X != Y is {{ x != y }}"
    - name: Display comparison results for strings
      debug:
        msg: "P == Q is {{ p == q }}; Q == R is {{ q == r }}"
```

Execution and Results
When you execute this playbook, you'll see output similar to the following:

For numbers (x=6 and y=10):

X == Y will be False because 6 is not equal to 10.
X != Y will be True because 6 is indeed not equal to 10.
For strings (p="Ansible", q="Automation", and r="Automation"):

P == Q will be False because "Ansible" is not equal to "Automation".
Q == R will be True because "Automation" is equal to "Automation".

Boolean Output: The critical takeaway is that the output of using comparison operators in Ansible (or any programming context) is a boolean value (True or False). This boolean output is particularly useful in conditional statements where you might want to execute certain tasks based on the comparison's outcome.

Introduction to Logical Operators in Ansible

Overview of Logical Operators

Logical operators allow the combination of multiple conditions or reverse the logic of a condition.
Three primary logical operators: and, or, not.
Usage of Logical Operators

and Operator: Returns True if both conditions are true.
or Operator: Returns True if at least one condition is true.
not Operator: Inverts the result of a condition (True becomes False, and vice versa).

Introduction to Ansible Test Operators

•**Purpose:** Test operators in Ansible are used to evaluate conditions or states of variables and data within templates and tasks.

•**Common Uses:** These operators can test if a variable is defined/undefined, check data types (none, even, odd, lower, upper), and validate file states (exist, file, link, directory).

```
---
- name: Demonstrate Ansible Test Operators with True/False Results
  hosts: localhost
  vars:
    x: 10
    my_name: 'AnsibleUser'
    my_path: '/tmp/example_file' # Ensure this path exists on your system and is a
file
    my_link_path: '/tmp/example_link' # Ensure this is a symbolic link on your
system
  tasks:
    - name: Output if variable 'x' is defined
      ansible.builtin.debug:
        msg: "'x' is defined: {{ x is defined }}"

    - name: Output if 'my_path' is a file
      ansible.builtin.debug:
        msg: "'my_path' is a file: {{ my_path is file }}"

    - name: Output if 'my_link_path' is a symbolic link
      ansible.builtin.debug:
        msg: "'my_link_path' is a symbolic link: {{ my_link_path is link }}"

    - name: Output if 'my_name' is lower case
      ansible.builtin.debug:
        msg: "'my_name' is in lower case: {{ my_name is lower }}"

    - name: Output if 'x' is an even number
      ansible.builtin.debug:
        msg: "'x' is even: {{ x is even }}"
```

Operations on List or Sequence

```
---
- name: List Operations Playbook
  hosts: localhost
  gather_facts: no
  vars:
    # Define the first list
    mylist: [5, 8, 3, 7, 4]
    # Define the second list
    mynewlist: [6, 7, 8, 9]

    # Merge the first and second list directly in vars
    mergedlist: "{{ mylist + mynewlist }}"

  tasks:
    # Display the original list
    - name: Display the original list
      debug:
        msg: "Original list: {{ mylist }}"

    # Find the number of elements in the list
    - name: Display the number of elements in the list
      debug:
        msg: "Number of elements: {{ mylist | length }}"

    # Find the minimum number in the list
    - name: Display the minimum number in the list
      debug:
        msg: "Minimum number: {{ mylist | min }}"

    # Find the maximum number in the list
    - name: Display the maximum number in the list
      debug:
        msg: "Maximum number: {{ mylist | max }}"

    # Access the first element using positive index
    - name: Display the first element (positive index)
      debug:
        msg: "First element: {{ mylist[0] }}"

    # Access the second element using negative index
    - name: Display the second element (negative index)
      debug:
        msg: "Second element using negative index: {{
mylist[-4] }}"

    # Use the first filter to get the first element
    - name: Display the first element using the 'first' filter
      debug:
        msg: "First element (filter): {{ mylist | first }}"

    # Use the last filter to get the last element
    - name: Display the last element using the 'last' filter
      debug:
        msg: "Last element (filter): {{ mylist | last }}"

    # Display the merged list
    - name: Display the merged list
      debug:
        msg: "Merged list: {{ mergedlist }}"

    # Dynamically create and display a merged list using
set_fact
    - name: Merge lists using set_fact
      set_fact:
        dynamic_mergedlist: "{{ mylist + mynewlist }}"

    - name: Display the dynamically merged list
      debug:
        msg: "Dynamically merged list: {{
dynamic_mergedlist }}"

    # Join list elements with a hyphen
    - name: Display joined list elements with hyphen
      debug:
        msg: "Joined list (hyphen): {{ mylist | join('-') }}"
```

Understanding how to manipulate and utilize lists within Ansible playbooks is crucial for dynamic and flexible automation strategies.

Operations on Strings

```
---
- name: Demonstrate String Operations in Ansible
  hosts: localhost
  vars:
    my_string: "Hello Ansible World"

  tasks:
    - name: Perform and display string operations
      debug:
        msg:
          - "Original string: {{ my_string }}"
          - "Length of the string: {{ my_string | length }}"
          - "Lowercase: {{ my_string | lower }}"
          - "Uppercase: {{ my_string | upper }}"
          - "Title case: {{ my_string | title }}"
          - "First character: {{ my_string[0] }}"
          - "Last character: {{ my_string[-1] }}"
          - "First two characters: {{ my_string[:2] }}"
          - "From third index: {{ my_string[3:] }}"
          - "Two characters from third index: {{ my_string[3:5] }}"
          - "Split by space: {{ my_string | split }}"
          - "Split by 'o': {{ my_string | split('o') }}"
```

Operations on Dictionaries

```
---
- name: Demonstrate dictionary operations in Ansible
  hosts: localhost
  gather_facts: no
  vars:
    # Define a dictionary with simple and complex keys
    dictionary:
      webserver: "httpd"
      database: "mysql"
      "linux.service": "systemd"

  tasks:
    # Display the entire dictionary
    - name: Display the dictionary
      debug:
        msg: "{{ dictionary }}"

    # Get all keys from the dictionary
    - name: Display dictionary keys
      debug:
        msg: "{{ dictionary.keys() }}"

    # Get all values from the dictionary
    - name: Display dictionary values
      debug:
        msg: "{{ dictionary.values() }}"

    # Access a specific value with different methods
    - name: Access specific value 'webserver' with .get()
      debug:
        var: dictionary.get('webserver')

    - name: Access specific value 'webserver' with square brackets
      debug:
        msg: "{{ dictionary['webserver'] }}"

    - name: Access specific value 'webserver' with dot notation
      debug:
        msg: "{{ dictionary.webserver }}"

    # Handling keys with special characters
    - name: Access value for key 'linux.service' with .get()
      debug:
        msg: "{{ dictionary.get('linux.service') }}"

    - name: Access value for key 'linux.service' with square brackets
      debug:
        msg: "{{ dictionary['linux.service'] }}"

    # Demonstrate failure case for dot notation with complex keys
    # Uncommenting the below task will result in an error due to the special
character in the key
    # - name: Try to access 'linux.service' with dot notation (Expected to fail)
    #   debug:
    #     msg: "{{ dictionary.linux.service }}"
```

Conditional Statements

Introduction to Conditional Statements in Ansible

Purpose: Conditional statements in Ansible are used to determine whether a specific task should be executed based on the evaluation of a condition.
Syntax: Utilizes the when keyword followed by a condition. The task is executed if the condition evaluates to true.

Understanding when Statement

Acts similarly to the if statement in programming languages.
Controls the execution of tasks based on specified conditions.

Writing Conditions

Conditions can be framed using:
Comparison operators (e.g., ==, !=, >, <)
Membership operators (in, not in)
Logical operators (and, or, not)
Test operators (e.g., is defined, is exists)

Practical Tips

Clear Syntax: Start the when condition aligned with the name and module space for readability.
No Jinja2 Syntax in Conditions: For all types of operators (comparison, membership, logical, test), exclude the curly braces ({{ }}) in the when condition.

Advanced Use

Conditional statements can be used with any task, allowing for dynamic and responsive playbook execution based on the environment or variables.

```yaml
---
- name: Demonstrate Conditional Execution Based on File Existence
  hosts: all
  vars:
    file_path: "/tmp/example_file.txt"  # Specify the file path here

  tasks:
    - name: Check if the specified file exists
      ansible.builtin.stat:
        path: "{{ file_path }}"
      register: file_stat  # Saving the result of the stat module to a variable

    - name: Display a message if the file exists
      ansible.builtin.debug:
        msg: "The file is: {{ file_path }}"
      when: file_stat.stat.exists  # Conditional statement to check file existence
```

Conditional Statements in Ansible

Introduction to Conditional Statements in Ansible

•**Purpose:** Conditional statements are used in Ansible to execute tasks based on specified conditions. This allows for more dynamic and responsive playbook designs.

Basic Conditional Execution

```
---
- name: Simple if-else statement
  hosts: localhost
  gather_facts: no
  vars:
    a: 6
    b: 3

  tasks:
    - name: Display the larger number when 'a' is greater than 'b'
      ansible.builtin.debug:
        msg: "The larger number is: {{ a }}"
      when: a > b

    - name: Display the larger number when 'b' is greater than 'a'
      ansible.builtin.debug:
        msg: "The larger number is: {{ b }}"
      when: b > a
```

Advanced Execution Using Inline If-Else Statement

```
---
- name: Inline if-else conditional statement
  hosts: localhost
  gather_facts: no
  vars:
    a: 6
    b: 3

  tasks:
    - name: Finding larger number with inline if-else statement
      ansible.builtin.debug:
        msg: "The large number is: {{ a if a > b else b }}"
```

Storing Result in Variable and Using set_fact

```
---
- name: Store result in variable using inline if-else
  hosts: localhost
  gather_facts: no
  vars:
    a: 3
    b: 6
    large: "{{ a if a > b else b }}"

  tasks:
    - name: Display the larger number using a variable
      ansible.builtin.debug:
        msg: "The large number is: {{ large }}"

    - name: Finding larger number with set_fact module
      ansible.builtin.set_fact:
        dynamically_determined_largenumber: "{{ a if a > b else b }}"

    - name: Display the larger number determined dynamically
      ansible.builtin.debug:
        msg: "The dynamically determined larger number is: {{ dynamically_determined_largenumber }}"
```

Introduction to Ansible Facts

Introduction to Default Ansible Facts

Overview: Ansible facts are a set of variables automatically gathered from managed nodes during playbook execution. They provide valuable information about the remote systems.
Gathering Facts with Ansible

Setup Module:

The setup module is used to gather default facts about managed nodes.
This gathering occurs as a default task in playbooks.
Example: Using the setup module explicitly in a task to collect system information.

Disabling Fact Gathering:

Fact gathering can be disabled by setting gather_facts: false in a playbook.
This is useful for speeding up playbook execution when facts are not needed.
Example: A playbook with gather_facts: false to skip automatic fact gathering.

Using Facts in Tasks:

Facts are stored in the ansible_facts variable and can be accessed without explicitly defining a variable.
Example: Accessing ansible_facts to retrieve OS family, disk information, or network interfaces.

Explicitly Gathering Facts

```
---
- name: Explicitly Gather Facts about Managed Nodes
  hosts: all
  gather_facts: false

  tasks:
    - name: Gather default facts using setup module
      ansible.builtin.setup:

    - name: Display all gathered facts
      ansible.builtin.debug:
        var: ansible_facts
```

This playbook first disables automatic fact gathering with gather_facts: false and then explicitly gathers and displays the facts using the setup module.

Using Facts to Conditionally Execute Tasks

```
---
- name: Conditional Tasks Based on OS Family
  hosts: all

  tasks:
    - name: Install Apache on RedHat/CentOS family
      ansible.builtin.yum:
        name: httpd
        state: present
      when: ansible_facts['os_family'] == "RedHat"

    - name: Install Apache on Debian/Ubuntu family
      ansible.builtin.apt:
        name: apache2
        state: present
      when: ansible_facts['os_family'] == "Debian"

    - name: Display OS Family
      ansible.builtin.debug:
        msg: "The OS family is {{ ansible_facts['os_family'] }}"
```

It automatically gathers facts and includes conditional tasks to install Apache using the package manager appropriate for the OS family (either YUM for RedHat/CentOS or APT for Debian/Ubuntu). It also displays the OS family using the debug module.

Overview of Systemd and Ansible

Introduction to systemd: systemd is the system and service manager for Unix-like operating systems. It provides capabilities such as starting, stopping, and managing services.

Ansible and systemd: Ansible can interact with systemd to manage services across multiple nodes without direct manual intervention, enhancing automation and consistency.

Using Ansible to Manage systemd Services

Gathering Service Facts with Ansible:

Purpose: Quickly and efficiently gather detailed information about the status of systemd services across managed nodes.

Module: service_facts - This Ansible module collects and lists facts about systemd services.

Stored Facts: The results from running the service_facts module are stored in a default variable called ansible_facts.

Gather Systemd Service Facts

```
---
- name: Gather Systemd Service Facts
  hosts: localhost
  gather_facts: no

  tasks:
    - name: Collect Systemd Service Facts
      ansible.builtin.service_facts:

    - name: Display All Systemd Service Facts
      ansible.builtin.debug:
        var: ansible_facts.services
```

Display Specific Service Status

```
---
- name: Display Specific Systemd Service Status
  hosts: localhost
  gather_facts: no

  tasks:
    - name: Collect Systemd Service Facts
      ansible.builtin.service_facts:

    - name: Display nginx Service Status
      ansible.builtin.debug:
        msg: "nginx service status: {{
ansible_facts.services['nginx.service'].state }}"
      when: "'nginx.service' in ansible_facts.services"

    - name: Display sshd Service Status
      ansible.builtin.debug:
        msg: "sshd service status: {{
ansible_facts.services['sshd.service'].state }}"
      when: "'sshd.service' in ansible_facts.services"
```

Ansible Inventory: Hostname and Host Vars

Introduction to Inventory Variables

Inventory variables are crucial in defining and managing the configuration of hosts within an Ansible playbook.
Two primary variables we'll focus on: inventory_hostname and hostvars.

Inventory Hostname Variable

Definition: The inventory_hostname variable represents the current host's name as defined in the inventory file being targeted by the playbook.

Key Points:

Automatically defined by Ansible.
Reflects the hostname of the node where the playbook is currently running.

Displaying inventory_hostname

```
---
- name: Display inventory_hostname
  hosts: all
  tasks:
    - name: Print inventory hostname
      debug:
        var: inventory_hostname
```

Displaying All hostvars

```
---
- name: Display hostvars
  hosts: localhost
  tasks:
    - name: Print host variables for all hosts
      debug:
        var: hostvars
```

Filtering hostvars for Specific Node Information

```
---
- name: Display group names for the node
  hosts: all
  tasks:
    - name: Print group names the current node belongs to
      debug:
        var: group_names
```

become, become_user, and become_method

Introduction to Privilege Escalation in Ansible

Overview: Ansible's become, become_user, and become_method options provide a mechanism to escalate privileges for executing tasks that require higher permissions.

Scenario: Trying to install a package like Nginx or any other service as a non-root user results in a permission error. These options help overcome such hurdles.

Key Concepts

become: Enables privilege escalation. Setting become: yes tells Ansible to execute the task with elevated privileges.

become_user: Specifies the user to 'become' when executing a task. Default is root, but can be any user.

become_method: Defines how to escalate privileges. Common methods include sudo, su, pbrun, and more, depending on the target system's configuration.

Applying Privilege Escalation

Task-Level: You can apply become options directly within individual tasks if only specific tasks require elevated privileges.
Play-Level: To apply privilege escalation to all tasks within a play, set become options at the play level, simplifying the playbook.

Basic Privilege Escalation to Install Nginx

```
---
- name: Install Nginx using yum with privilege escalation
  hosts: all
  gather_facts: false
  become: yes
  become_user: root
  become_method: sudo
  tasks:
    - name: Ensure Nginx is installed
      ansible.builtin.yum:
        name: nginx
        state: present
```

Task-Level Privilege Escalation

```
---
- name: Mixed privilege playbook
  hosts: all
  gather_facts: false
  tasks:
    - name: Install tree without privileges
      ansible.builtin.yum:
        name: tree
        state: present
      become: no

    - name: Install httpd with privilege escalation
      ansible.builtin.yum:
        name: httpd
        state: present
      become: yes
      become_user: root
```

Advanced Use Case: Custom become_user and become_method

```
become: yes
become_user: dbadmin        # Database administrator user
become_method: su           # Switch user method
tasks:
  - name: Execute DB task
    # Task details here
```

Best Practices

Minimal Use of Root: Use root privileges sparingly, only when necessary, to minimize security risks.

Role-Specific Privileges: Where possible, escalate to a role-specific user (like dbadmin) instead of root to adhere to the principle of least privilege.

Conclusion

Understanding and effectively using become, become_user, and become_method are crucial for Ansible playbook development, especially when managing tasks that require different levels of access rights. Tailoring the use of these options based on the task requirements enhances security while ensuring smooth automation workflows.

import_tasks and include_tasks

Understanding Package Management with Ansible

Introduction to OS-Specific Package Management:

Different Linux distributions use different package managers, e.g., Yum/DNF for RedHat-based systems and apt for Debian-based systems.
The choice of package manager is crucial for automation scripts to work across various environments.

Ansible Modules for Package Installation:

Ansible provides different modules for package management, such as yum, dnf for RedHat family distributions, and apt for Debian family distributions.
The correct module needs to be chosen based on the target managed node's OS family.

Leveraging Ansible Facts for OS Detection

What are Ansible Facts?

Ansible facts are a set of variables that contain information about the managed nodes, including the OS family.
These facts are gathered by the setup module, which is executed by default at the beginning of a playbook run.

Displaying OS Family

```
- name: Display OS family
  hosts: all
  tasks:
    - name: Print OS family
      debug:
        msg: "OS Family: {{ ansible_facts['os_family'] }}"
```

Key Points to Remember

Using when: The when statement is used to conditionally execute a task based on the OS family.

Multiple Packages: To install multiple packages, provide the package names as a list under the name parameter.

Elevated Privileges: Use the become: yes directive at the play level to apply root privileges for all tasks, ensuring package managers can execute properly.

Conditional Execution Based on OS Family

```
- name: Install packages based on OS family
  hosts: all
  become: yes  # Elevated privileges for package installation
  tasks:
    - name: Install packages on RedHat family
      yum:
        name:
          - nginx
          - vim
        state: present
        update_cache: true
      when: ansible_facts['os_family'] == "RedHat"

    - name: Install packages on Debian family
      apt:
        name:
          - nginx
          - vim
        state: present
        update_cache: true
      when: ansible_facts['os_family'] == "Debian"
```

Reuse Ansible-Tasks with import_tasks and include_tasks

Overview of Task Reusability in Ansible

Problem Statement:

In Ansible playbooks, identical tasks are often rewritten across different playbooks, leading to redundancy and maintainability issues.

Solution:

Reusable tasks: Write a task once in a separate YAML file and reuse it wherever required.

Enhances readability: Instead of cluttering a main playbook with numerous tasks, separate tasks into individual files and reference them in the main playbook.

Methods for Reusing Tasks: Import vs. Include

import_tasks:

Static Import: Tasks are imported at playbook parsing time.

Use Case: Ideal for static, unchanging tasks across different playbooks.

Limitation: Doesn't support dynamic inclusion based on runtime variables.

include_tasks:

Dynamic Inclusion: Tasks are included at runtime, allowing for conditional execution.

Use Case: Best for tasks that depend on runtime conditions or variables.

Advantage: Supports dynamic file naming and inclusion, making it more flexible.

Task Files

install_RedHat_package.yml

```
---
# File: install_RedHat_package.yml
- name: Install Nginx on RedHat-based Systems
  yum:
    name: nginx
    state: present
```

install_Debian_package.yml

```
---
# File: install_Debian_package.yml
- name: Install Nginx on Debian-based Systems
  apt:
    name: nginx
    state: present
```

Main Playbook Using import_tasks

```
---
# File: main_playbook_import.yml
- name: Install Nginx on Various OS Families Using import_tasks
  hosts: all
  tasks:
    - name: Import tasks for RedHat family
      import_tasks: install_RedHat_package.yml
      when: ansible_facts['os_family'] == "RedHat"

    - name: Import tasks for Debian family
      import_tasks: install_Debian_package.yml
      when: ansible_facts['os_family'] == "Debian"
```

Main Playbook Using include_tasks

```
---
# File: main_playbook_include.yml
- name: Install Nginx on Various OS Families Using include_tasks
  hosts: all
  tasks:
    - name: Dynamically include task file based on OS family
      include_tasks: "install_{{ ansible_facts['os_family'] }}_package.yml"
```

This playbook dynamically selects the correct task file (install_RedHat_package.yml or install_Debian_package.yml) based on the ansible_facts['os_family'] variable. The use of include_tasks allows for this flexibility, making it possible to adapt the playbook's behavior at runtime based on the target hosts' characteristics.

Handlers

Introduction to Ansible Handlers

Overview: Handlers in Ansible are special tasks that only run when notified by another task. This mechanism is particularly useful for managing service states like restarting a service only if its configuration has changed.

Key Concept: Handlers are triggered by a notify directive in a task. They are idempotent, meaning they will only run once even if notified multiple times in the same playbook run.

Practical Use Case of Handlers

•**Configuration Changes:** Consider a task that updates a configuration file. If the file changes, the service needs to be restarted for changes to take effect. Here, handlers are ideal as they ensure the service is only restarted if the configuration is actually changed.

Best Practices

•**Naming Convention:** Start handler names with a capital letter for readability.
•**Idempotency:** Leverage handlers to maintain idempotency, ensuring that services are not unnecessarily restarted.

Playbook Without Handlers

```
---
- name: Install and start nginx without handlers
  hosts: localhost
  gather_facts: false
  become: yes

  tasks:
    - name: Install nginx
      ansible.builtin.dnf:
        name: nginx
        state: present

    - name: Start nginx service
      ansible.builtin.service:
        name: nginx
        state: started
```

Using when Conditional for Starting nginx

```
---
- name: Install and conditionally start nginx using 'when'
  hosts: localhost
  gather_facts: false
  become: yes

  tasks:
    - name: Install nginx
      ansible.builtin.dnf:
        name: nginx
        state: present
      register: nginx_installation # Store the result of the installation task

    - name: Start nginx service
      ansible.builtin.service:
        name: nginx
        state: started
      when: nginx_installation.changed # Conditionally start nginx only if there were changes
```

Playbook With Handlers

```
---
- name: Install and manage nginx with handlers
  hosts: localhost
  gather_facts: false
  become: yes

  tasks:
    - name: Install nginx
      ansible.builtin.yum:
        name: nginx
        state: present
      notify: Start nginx

  handlers:
    - name: Start nginx
      ansible.builtin.service:
        name: nginx
        state: started
```

Understanding Handlers in the Context

•**Handlers are Idempotent:** They ensure that the desired state is achieved without repeating actions unnecessarily. In this case, nginx is only started if it's not already running or if it's newly installed.

•**When to Use Handlers:** Handlers are particularly useful for service management tasks like starting, stopping, or restarting services in response to configuration changes or installations. They ensure actions are only taken when necessary, based on the outcomes of other tasks.

•**Efficiency and Organization:** By separating tasks that initiate changes and tasks that respond to changes, playbooks become more organized and efficient. Handlers contribute to clearer playbook structure and logic flow.

Tags to Execute Tasks

Introduction to Ansible Tags

Purpose of Tags: Tags in Ansible are a powerful feature that allows for selective execution of tasks in a playbook. They provide a way to run only a specific set of tasks, enhancing flexibility and efficiency during automation.

Basic Concept of Tags

Default Behavior: By default, Ansible executes all tasks in a playbook sequentially.

Controlling Execution: Tags can be added to tasks to control which tasks are executed when running the playbook.

Step-by-Step Guide on Using Tags

Adding Tags to Tasks:

Example: For demonstration, assume a playbook with five tasks, all using the debug module.

Add a tags key to each task in the playbook, assigning a unique identifier as the value.

Using the 'never' Tag:

Assigning the never tag to a task ensures it will not run by default.

Example: After adding tags: never to all tasks, running the playbook without specifying any tags will result in no tasks being executed.

Selective Task Execution:

Example: Assign meaningful tags like first, second, etc., to corresponding tasks.

Run the playbook with the -t or --tags option followed by the desired tag(s) to execute only the tasks associated with those tags.

Executing Multiple Tasks with a Single Tag:

To run multiple tasks under a single tag, assign the same tag value to those tasks.

Example: If tasks 1 and 4 both have the tag 1st4th, running the playbook with -t 1st4th executes both tasks.

Using 'always' and 'never' Together:

The always tag ensures a task runs regardless of other specified tags, taking priority over never.

Example: Assigning tags: always to a task makes it execute by default, even if no tags are specified when running the playbook.

Remember, -t or --tags followed by the tag name is the key to controlling which tasks run.
Consider tag planning as part of your playbook design to maximize efficiency and maintainability.

Basic Playbook with Tags

```
---
- name: Demonstrate tags in Ansible
  hosts: localhost
  tasks:
    - name: Task 1 - Display a message
      debug:
        msg: "This is Task 1"
      tags:
        - first

    - name: Task 2 - Display a message
      debug:
        msg: "This is Task 2"
      tags:
        - second

    - name: Task 3 - Display a message
      debug:
        msg: "This is Task 3"
      tags:
        - third

    - name: Task 4 - Display a message
      debug:
        msg: "This is Task 4"
      tags:
        - fourth
```

Using the 'never' Tag to Exclude Tasks

```
---
- name: Using 'never' tag to exclude tasks
  hosts: localhost
  tasks:
    - name: Task 1 - This task has a 'never' tag
      debug:
        msg: "This task won't run by default because it has a
'never' tag."
      tags:
        - never

    - name: Task 2 - This task also has a 'never' tag
      debug:
        msg: "This task also won't run by default."
      tags:
        - never
```

all tasks are tagged with 'never', which means they won't run by default unless the tag is explicitly called.

Mixing 'never', Specific Tags, and 'always'

```
---
- name: Mixing 'never', specific tags, and 'always'
  hosts: localhost
  tasks:
    - name: Task 1 - Specific tag
      debug:
        msg: "Task 1 with a specific tag."
      tags:
        - first
        - never

    - name: Task 2 - Will run by default
      debug:
        msg: "Task 2 will always run."
      tags:
        - always

    - name: Task 3 - Another task that will run by default
      debug:
        msg: "Task 3 will also always run."
      tags:
        - always
```

This playbook shows how to mix 'never', specific tags for individual tasks, and the 'always' tag for tasks you want to execute by default.

```
ansible-playbook example_playbook.yml --tags first

ansible-playbook example_playbook.yml --tags never

ansible-playbook example_playbook.yml
```

Error Handling in Ansible

Overview of Error Handling in Ansible Playbooks

Ansible provides various ways to handle errors during playbook execution. Understanding these mechanisms is crucial for robust playbook design.
Today's focus is on three key error handling methods: ignore_errors, failed_when, and using the fail module.

Method 1: Using ignore_errors

Purpose: To continue executing the remaining tasks in a playbook even if a certain task fails.

Syntax: ignore_errors: true

Example: Assuming a task to check Nginx version fails because Nginx is not installed, setting ignore_errors: true for this task allows the playbook to proceed with subsequent tasks.

```
- name: Check Nginx version
  command: nginx -v
  ignore_errors: true
```

Insight: Use this option judiciously, as it might lead to ignoring critical failures that should otherwise halt playbook execution for troubleshooting.

Method 2: Using failed_when

Purpose: To define custom conditions under which a task is considered failed.

Syntax: failed_when: condition

Example: If having Nginx installed is undesired, a task can be configured to fail when the check for Nginx version succeeds (indicated by a return code of 0).

```
- name: Ensure Nginx is not installed
  command: nginx -v
  register: nginx_version
  failed_when: nginx_version.rc == 0
```

Insight: This method offers flexibility in defining failure conditions based on task outcomes, useful for scenarios where the success or failure isn't binary.

Method 3: Using the fail Module

Purpose: Explicitly fail the playbook execution based on certain conditions.

Syntax: Use the fail module with a conditional when statement.

Example: Fail the playbook if a specific condition is met, such as Nginx being installed.

```
- name: Fail playbook if Nginx is installed
  fail:
    msg: "Failing the playbook because Nginx is installed."
  when: nginx_installed is defined and nginx_installed
```

Insight: The fail module provides a direct way to halt playbook execution, making it clear under what conditions the playbook should not proceed.

Using ignore_errors in a Package Installation Task

This playbook demonstrates ignoring errors during package installation. This could be useful in scenarios where the package might already be installed or where the failure of installation should not halt the playbook execution.

```
---
- name: Install packages with error ignoring
  hosts: all
  tasks:
    - name: Install httpd (Example for CentOS/RHEL)
      yum:
        name: httpd
        state: present
      ignore_errors: true

    - name: Ensure that another task runs regardless of the httpd install result
      debug:
        msg: "This task runs even if the httpd installation fails."
```

Dynamically Failing a Task with failed_when

This playbook shows how to define custom failure conditions using the failed_when directive. It's set to fail if Nginx is installed (simulated by checking the return code of a command).

```
---
- name: Play to understand about error handling
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Finding nginx version
      ansible.builtin.command:
        cmd: 'nginx -version'
      register: nginxVerOutput
      failed_when: nginxVerOutput.rc == 0
```

Using the fail Module

This playbook uses the fail module to explicitly fail the playbook based on a specific condition. It demonstrates conditional failure if a variable (nginx_installed) indicates that Nginx is installed.

```
---
- name: Error Handling using fail module
  hosts: all
  vars:
    nginx_installed: true # Assuming nginx_installed is a
condition determined earlier

  tasks:
    - name: Fail the playbook if Nginx is installed
      fail:
        msg: "Failing the playbook because Nginx is installed."
      when: nginx_installed
```

```
---
- name: Conditional failure based on software version
  hosts: all
  tasks:
    - name: Check the installed version of a package
      shell: echo "2.0" # Simulating command to check version,
replace with actual command
      register: installed_version

    - name: Fail the playbook if the installed version is not what
we want
      fail:
        msg: "An unsupported version ({{ installed_version.stdout
}}) is installed."
      when: installed_version.stdout != "1.0"
```

block and rescue

Introduction to Error Handling in Ansible

Overview: Discuss how Ansible uses block, rescue, and always for error handling, akin to try, except, and finally in traditional programming.

Purpose: To gracefully handle errors during playbook execution and ensure tasks can fail safely without halting subsequent playbook operations.

Block and Rescue: Basic Concepts

Block: Acts like a try block, containing one or more tasks that Ansible attempts to execute.
Rescue: Functions like an except block, containing tasks that execute if any task within the block encounters an error.
Always: Similar to finally, tasks within this section run regardless of success or failure in the block or rescue sections.

Insights and Tips

Selective Execution: rescue only executes if a task in block fails, making it ideal for error recovery steps.

Always Executed: Use always for cleanup or finalization tasks that must run regardless of earlier outcomes.

Complex Scenarios: These structures support complex error handling and recovery scenarios within Ansible playbooks, improving reliability and user feedback.

Handling NGINX Installation with Block and Rescue

```
---
- name: Validate and ensure NGINX installation
  hosts: localhost
  gather_facts: false

  tasks:
    - name: Attempt to validate NGINX installation
      block:
        - name: Check if NGINX is installed
          command: ls /usr/sbin/nginx
          register: nginx_check
          failed_when: nginx_check.rc != 0

      rescue:
        - name: NGINX is not installed, installing NGINX
          yum:
            name: nginx
            state: present

      always:
        - name: Display NGINX installation status message
          debug:
            msg: "NGINX installation validation and handling process completed."
```

Applying Root Privileges with Block

This playbook showcases installing NGINX, ensuring it's running, and then gathering and displaying its service status. By setting become: yes under the block, all tasks within this block will be executed with elevated privileges, eliminating the need to specify become: yes for each individual task.

```
---
- name: Install NGINX, ensure it is running, and display its status
  hosts: localhost
  gather_facts: false

  tasks:
    - block:
        - name: Installing NGINX
          yum:
            name: nginx
            state: present

        - name: Ensuring NGINX is running
          service:
            name: nginx
            state: started

      become: yes  # This applies root privileges to all tasks within the block

      always:
        - name: Gather service facts
          service_facts:

        - name: Displaying status of NGINX
          debug:
            msg: "The NGINX service is {{
ansible_facts.services['nginx.service'].state }}."
```

Ansible Log File

Introduction to Ansible Logging

Overview: By default, Ansible outputs the results of commands and playbook executions to the console (STDOUT) on the control node. This is immediate but not persistent.

Purpose of Logging: To capture Ansible's operational output for record-keeping, debugging, and auditing purposes.
Configuring Ansible for Logging

Ansible Configuration File (ansible.cfg):

Ansible behaviour can be customized via the ansible.cfg configuration file.
To enable logging to a file, the log_path option must be set.

Setting log_path:
Specify the full path to the log file where you want to capture Ansible's activities.
Example Configuration: In the ansible.cfg file, add:

```
[defaults]
log_path=/home/ansibleuser/logs/ansible.log
```

Ensure the directory (/home/admin/automation/logs/) exists, or Ansible won't be able to write to the log file.
Directory Existence is Crucial:
The directory specified in log_path must exist beforehand; otherwise, Ansible fails to log.
Example: If a warning like "log file ... is not writeable" appears, check if the directory exists and has appropriate permissions.

Usage of remote_src , delegate_to, and run_once

1. remote_src in Ansible Playbooks

Purpose: The remote_src option is used with tasks that involve copying, synchronizing, or otherwise manipulating files. It specifies where the source of the file or directory is located in relation to the task being executed.

Default Behavior: Without remote_src or when it's set to false, Ansible assumes the source file or directory is on the control node.

When Set to true: It tells Ansible that the source file or directory is on the remote target node.

Unarchiving a File on Remote Hosts

```
---
- name: Unarchive a file from a remote source
  hosts: all
  tasks:
    - name: Unarchive a tar file from a remote source
      ansible.builtin.unarchive:
        src: /tmp/dummy.tar.gz
        dest: /home/ansibleuser/dummy
        remote_src: yes
```

This playbook uses the unarchive module to extract an archive located at /tmp/example_files.tar.gz on the remote hosts to the directory /home/user/example_files. Setting remote_src: yes tells Ansible the source is on the remote machine, not locally on the control node.

Using delegate_to in Ansible Playbooks

The delegate_to option allows you to execute a task on a different host than the one targeted in the playbook's hosts directive. This is useful for tasks that need to be run on a specific machine or need to be centralized, such as logging or monitoring actions.

Fetching a File from a Remote Host to Local

```
---
- name: Fetch a file from a specific host
  hosts: all
  tasks:
    - name: Fetch /var/log/example.log from the database server
      ansible.builtin.fetch:
        src: /var/log/example.log
        dest: /tmp/
      delegate_to: db_server
      run_once: true
```

This playbook fetches the log file /var/log/example.log from the host aliased as db_server, regardless of the playbook's target hosts. The task is delegated to db_server, and run_once: true ensures it's executed only once, not for each host in the inventory.

Using run_once in Ansible Playbooks

The run_once option ensures a task is executed only once across all hosts targeted by the playbook, rather than once per host. This is particularly useful for actions that need to be performed only once in an execution context, such as database initialization or gathering data from a single source.

```
[webservers]
webserver1 ansible_host=192.168.1.101
webserver2 ansible_host=192.168.1.102

[dbservers]
dbserver1 ansible_host=192.168.1.201
dbserver2 ansible_host=192.168.1.202
```

```
---
- name: Example playbook using run_once
  hosts: all
  tasks:
    - name: Run a debug message only once
      ansible.builtin.debug:
        msg: "This task runs only once, and this message is from {{ inventory_hostname }}"
      run_once: true
```

How run_once Works in This Context

When you execute this playbook, Ansible targets all hosts specified under all, which includes both webservers and dbservers.

The debug task is designed to print a message indicating from which host it's running. Due to run_once: true, this task will execute only once, despite the playbook targeting multiple hosts.

Because of how Ansible processes hosts (by default, in the order they appear in the inventory unless explicitly ordered otherwise via sorting or other methods), webserver1 will be the host on which the debug task executes. This is because webserver1 is the first host listed in the inventory file under the all category that includes both groups (webservers and dbservers).

Why webserver1?

Ansible processes the inventory from top to bottom unless explicitly instructed otherwise. Since webserver1 appears first in the inventory's listing (considering all hosts), it's the first host Ansible evaluates.

With run_once: true, Ansible ensures the task executes only once across the play, choosing the first host it comes across in its evaluation, which in this case is webserver1.

Playbook Overview – Run By Yourself

Let's create a new example that encompasses tags, blocks, rescue, and handlers within an Ansible playbook. Our goal will be to manage a web server environment where we want to check the versions of installed software (Node.js and Nginx), install them if missing, and restart the services if any changes are made.

Playbook Overview

1. **Check Software Versions:** We'll check the versions of Node.js and Nginx.
2. **Install Missing Software:** If either software is not found, we'll install it.
3. **Use Tags to Selectively Execute Tasks:** Tags will allow us to target specific parts of the playbook.
4. **Utilize Blocks and Rescue for Error Handling:** If checking the version results in an error, indicating software is missing, we'll catch this with a rescue block.
5. **Employ Handlers to Restart Services:** If installation occurs, we'll use handlers to restart the services, ensuring they're using the latest configurations.

```yaml
---
- name: Manage Web Server Environment
  hosts: web_servers
  become: yes
  vars_prompt:
    - name: target_software
      prompt: "Which software to check? (nodejs/nginx/all)"
      private: no

  tasks:
    - name: Check Node.js version
      command: node -v
      register: node_version
      failed_when: node_version.rc != 0
      ignore_errors: yes
      tags:
        - nodejs
        - all
      block:
        - name: Install Node.js
          apt:
            name: nodejs
            state: latest
          when: node_version.rc != 0
          notify: restart nodejs

      rescue:
        - name: Node.js not found, installing...
          debug:
            msg: "Node.js is not installed. Proceeding with installation..."

    - name: Check Nginx version
      command: nginx -v
      register: nginx_version
      failed_when: nginx_version.rc != 0
      ignore_errors: yes
      tags:
        - nginx
        - all
      block:
        - name: Install Nginx
          apt:
            name: nginx
            state: latest
          when: nginx_version.rc != 0
          notify: restart nginx

      rescue:
        - name: Nginx not found, installing...
          debug:
            msg: "Nginx is not installed. Proceeding with installation..."

  handlers:
    - name: restart nodejs
      service:
        name: nodejs
        state: restarted
      listen: "restart nodejs"
      tags: nodejs

    - name: restart nginx
      service:
        name: nginx
        state: restarted
      listen: "restart nginx"
      tags: nginx
```

How This Works

- Execution Flow: The playbook begins by checking the versions of Node.js and Nginx, each wrapped in a block to handle potential failures.
- Error Handling with Blocks and Rescue: If the version check fails (indicating the software is not installed), the rescue section triggers a message and then the playbook proceeds to install the missing software.
- Selective Execution with Tags: By using the --tags option when running the playbook, we can choose to check/install only Node.js, only Nginx, or both.
- Service Restart with Handlers: If a new installation occurs, the respective handler is notified to restart the service, ensuring the latest version is running.

Check and manage Node.js only

```
ansible-playbook playbook.yml --tags nodejs
```

Check and manage Nginx only

```
ansible-playbook playbook.yml --tags nginx
```

Manage both Node.js and Nginx

```
ansible-playbook playbook.yml --tags all
```

Loops

Introduction to Loops in Ansible

1.Overview of Loops:

1. Loops in Ansible allow for the execution of a task multiple times, similar to loops in traditional programming and scripting languages.
2. Utilizing loops can simplify playbook complexity and reduce redundancy.

2.Basic Example without Loops:

1. Initially, creating multiple files requires multiple tasks, each specifying a file name.
2. This method is not efficient as it leads to task repetition with only minor differences (e.g., the file name).

3.Enhancing Efficiency with Loops:

1. Instead of duplicating tasks for each file, a single task can be written with a loop that iterates over a list of file names.
2. This approach significantly reduces playbook length and increases maintainability.

Advanced Looping Techniques

Beyond Basic Loops:

While the loop keyword covers many use cases, Ansible provides additional looping constructs such as with_items, with_dict, and until but in latest versions it is deprecated.

Choosing the Right Looping Method:

Start with the loop keyword for simplicity and readability.

Explore other looping constructs if your use case requires more complex looping logic or behaviour.

Creating Multiple Files

```
---
- name: Loop example to create files
  hosts: all
  tasks:
    - name: Create multiple files using a loop
      file:
        path: "/tmp/{{ item }}"
        state: touch
      loop:
        - "config"
        - "config.default"
        - "config.test"
```

- In this playbook:

- We define a single task to create files.
- The loop directive iterates over each item in the provided list.
- The {{ item }} variable dynamically takes the value of each item in the list, creating a new file for each iteration.

Installing Multiple Packages

```
---
- name: Loop example to install multiple packages
  hosts: all
  become: yes  # Use 'become' to execute tasks with root
privileges
  tasks:
    - name: Ensure multiple packages are installed
      yum:
        name: "{{ item }}"
        state: present
      loop:
        - "httpd"
        - "git"
        - "tree"
```

- This playbook uses the yum module (for CentOS/RHEL systems) to install packages.
- The loop iterates over each package name in the list, installing each one.
- Using become: yes ensures the task has the necessary permissions to install packages.

Loops with List

Introduction to Looping in Ansible Playbooks

Objective

- Understand how to efficiently iterate over lists in Ansible playbooks using loops.
- Learn to manage and simplify playbook code by leveraging Ansible's looping constructs.

Key Concepts

- **Loops**: A fundamental programming concept that Ansible incorporates, allowing tasks to be repeated over a list or a sequence of elements.
- **List Variable**: A method to store multiple values in a single variable, which can then be iterated over with a loop.

Why Use Loops?

- Reduces code duplication by eliminating the need to write the same task repeatedly for different items.
- Enhances playbook readability and maintainability.
- Facilitates dynamic task execution over a set of items.

Playbook Using loop

```
---
- name: Ensure required files are present using loop
  hosts: all
  vars:
    required_files:
      - /tmp/required_file1
      - /tmp/required_file2
      - /tmp/required_file3

  tasks:
    - name: Ensure each required file is present
      ansible.builtin.file:
        path: "{{ item }}"
        state: touch
      loop: "{{ required_files }}"
```

Playbook Using with_list

This playbook accomplishes the same objective as the first example but uses the with_list syntax to iterate over the list of required files.

```
---
- name: Ensure required files are present using with_list
  hosts: all
  vars:
    required_files:
      - /tmp/required_file1
      - /tmp/required_file2
      - /tmp/required_file3

  tasks:
    - name: Ensure each required file is present
      ansible.builtin.file:
        path: "{{ item }}"
        state: touch
      with_list: "{{ required_files }}"
```

Loops for Dictionaries

Creating Multiple Files with Debugging

```
- name: Create multiple files in specified locations with debugging
  hosts: all
  vars:
    files:
      - name: "config1"
        directory: "/temp"
        mode: "0755"
      - name: "config2"
        directory: "/home/admin"
        mode: "0600"
      - name: "config3"
        directory: "/home/admin/books"
        mode: "0500"
  tasks:
    - name: Debug each item
      debug:
        msg: "Creating file {{ item.name }} in {{ item.directory }} with mode {{ item.mode }}"
      loop: "{{ files }}"

    - name: Create files with specific permissions
      ansible.builtin.file:
        path: "{{ item.directory }}/{{ item.name }}"
        state: touch
        mode: "{{ item.mode }}"
      loop: "{{ files }}"
```

Jinja Templates [Template Module]

•**Purpose:** The template module is designed to dynamically create files by replacing variables, conditions, and loops with specific values. This functionality is especially useful in configurations that largely remain the same across different environments, with only a few lines or values needing adjustments.

Key Concepts

•**Dynamic Configuration:** Allows for the dynamic adjustment of configuration files for applications deployed across managed nodes, changing specific lines or values as needed.

•**Variables and Templates:** Utilizes variables within templates to replace static content with dynamic values based on the environment or conditions.

Advantages of Using the Template Module

•**Flexibility:** Enables the customization of files based on the target environment, allowing for a single template to cater to multiple configurations.

•**Efficiency:** Automates the process of updating configurations or content based on specific conditions or variables, reducing manual errors and saving time.

Dynamic Web Server Installation

```
---
- name: Install web server based on OS family
  hosts: all
  become: yes
  vars:
    package_name:
      RedHat: httpd
      Debian: apache2
  tasks:
    - name: Install web server package
      package:
        name: "{{ package_name[ansible_os_family] }}"
        state: present
```

Deploying Dynamic Index.html Using Template Module

First, create a template file named index.j2 in your Ansible project's templates directory. This template includes Jinja2 variables that will be replaced dynamically.

```
<!-- index.j2 -->
<html>
<head>
   <title>Welcome to Our Web Server</title>
</head>
<body>
   <h1>This is the HTTPD web server from {{ ansible_host }} </h1>
</body>
</html>
```

```yaml
---
- name: Install web server and deploy dynamic index.html
  hosts: all
  become: yes
  vars:
    package_name:
      RedHat: httpd
      Debian: apache2
  tasks:
    - name: Install web server package
      package:
        name: "{{ package_name[ansible_os_family] }}"
        state: present

    - name: Deploy index.html from template
      template:
        src: index.j2
        dest: /var/www/html/index.html
      notify: restart web server

  handlers:
    - name: restart web server
      service:
        name: "{{ package_name[ansible_os_family] }}"
        state: restarted
```

Ansible Templates

Introduction to Ansible Templates

Purpose of Templates: Templates in Ansible are used to generate host-specific configuration files from a template source, allowing for dynamic content creation based on variables, conditions, and loops.

Template Files: Typically have a .j2 extension, standing for Jinja2, Ansible's templating language of choice. While not mandatory, this convention helps identify template files.

Using Variables in Templates

Basic Syntax: Variables in Jinja2 templates are referenced using double curly braces {{ variable_name }}.

Example: If you have variables a and b, you can display their values in a template using The inputs are {{ a }} and {{ b }}.

Implementing Conditions

Conditional Statements: Jinja2 supports if conditions to selectively include content in the generated files.

```
{% if a > b %}
The larger number is {{ a }}
{% endif %}
```

Extended Use: Besides simple if statements, Jinja2 supports elif and else for more complex logic.

Utilizing Loops

Looping Over Lists: To iterate over a list of items in a template, use the for loop syntax.

Syntax:

```
{% for item in list %}
- {{ item }}
{% endfor %}
```

Using Variables in a Template

```
---
- name: Demonstrate variables in templates
  hosts: localhost
  vars:
    a: 80
    b: 70
  tasks:
    - name: Generate file from template
      ansible.builtin.template:
        src: templates/variables_example.j2
        dest: /tmp/variables_example.txt
```

Template: templates/variables_example.j2

```
The inputs are {{ a }} and {{ b }}.
```

Implementing Conditions

```
---
- name: Demonstrate conditions in templates
  hosts: localhost
  vars:
    a: 80
    b: 70
  tasks:
    - name: Generate file from template with conditions
      ansible.builtin.template:
        src: templates/conditions_example.j2
        dest: /tmp/conditions_example.txt
```

Template: templates/variables_example.j2

```
The inputs are {{ a }} and {{ b }}.

{% if a > b %}
The larger number is {{ a }}.
{% else %}
The larger number is {{ b }}.
{% endif %}
```

Utilizing Loops

```
---
- name: Demonstrate loops in templates
  hosts: localhost
  vars:
    packages:
      - nginx
      - vim
      - wget
      - curl
  tasks:
    - name: Generate file from template with loops
      ansible.builtin.template:
        src: templates/loops_example.j2
        dest: /tmp/loops_example.txt
```

```
Packages to install:
{% for package in packages %}
- {{ package }}
{% endfor %}
```

Advanced Templating with Loops and Conditions

```
---
- name: Demonstrate advanced templating in Ansible
  hosts: localhost
  vars:
    my_string: "Ansible"
    my_list:
      - nginx
      - vim
      - wget
      - curl
  tasks:
    - name: Generate file from advanced template
      ansible.builtin.template:
        src: templates/advanced_templating_example.j2
        dest: /tmp/advanced_templating_example.txt
```

```
String Iteration:
{% for char in my_string %}
- {{ char }}
{% endfor %}

Package List:
{% for item in my_list %}
- {{ item }}
{% endfor %}
```

Ansible Plugins

Introduction to Ansible Plugins

1.Definition of Plugin:

1. An Ansible plugin is a piece of code designed to enhance or modify the core functionality of Ansible. It's crucial to distinguish between plugins and modules, as they serve different purposes within Ansible.

2.Plugins vs. Modules:

1. **Modules** are primarily used to execute tasks or operations on managed nodes.
2. **Plugins**, on the other hand, alter or add new functionalities to Ansible's operations, extending its capabilities beyond the execution of tasks.

Types of Plugins and Their Uses

Examples of Plugin Types:

Become Plugins: Enable privilege escalation, allowing tasks to run with elevated permissions (e.g., become: yes in playbooks).

Filter Plugins: Transform data within templates and playbooks (e.g., converting strings to lowercase, finding string lengths).

Inventory Plugins: Dynamically generate inventory from external sources or formats.

Other types include connection plugins, lookup plugins, callback plugins, etc.

Practical Examples

Become Plugin Use Case:

To run a task as a superuser, you might use the become keyword in a playbook, effectively utilizing the become plugin to gain root privileges for that task.

Filter Plugin Example:

When you need to manipulate strings within your playbook, you might use a filter plugin like {{ "Hello World"|lower }} to convert a string to lowercase.

Discovering and Utilizing Plugins

Listing Available Plugins:

Use the ansible-doc -l command to list all available modules. For plugins, a similar approach can be taken, tailored by the type of plugin you're interested in.

Plugin Naming Convention:

Both modules and plugins follow a similar naming convention: namespace.collection.plugin_or_module_name. This helps in identifying and organizing plugins within collections.

Finding Specific Plugins:

To list all filter plugins, for example, you can use the command ansible-doc -t filter -l

Similarly, for connection plugins, the command would be ansible-doc -t connection -l

Ansible Community Collections

Introduction to Ansible Collections

- **Ansible Core**: Starts with a minimal set of modules and plugins known as Ansible built-in modules and plugins.
- **Expansion**: For additional functionality, such as cloud integration, additional modules and plugins are necessary.

Identifying the Need for Additional Modules

Limitation: A direct search in Ansible Core for cloud-related modules (e.g., AWS) shows no results, indicating the core's limitations in managing cloud services.

Solution: To manage cloud services, one must extend Ansible's capabilities through either custom development or utilizing third-party collections.

Methods to Extend Ansible

Custom Development:

Involves creating custom modules and plugins using Python scripting.

Considered time-consuming and requires Python knowledge.

Using Third-party Collections:

Install needed modules and plugins from existing collections.

Collections are grouped under namespaces for organization.

Working with Ansible Collections

Finding Collections: For AWS management, search for "Ansible AWS collection" to find relevant collections.

Namespace and Collections: Collections are identified by a namespace (e.g., Amazon.AWS) which helps in distinguishing and organizing collections.

Installing a Collection

Installation Command: Use ansible-galaxy collection install namespace.collection_name to install a collection.
Example: To install the AWS collection from Amazon namespace, the command would be something like ansible-galaxy collection install amazon.aws.

Best Practice: Install collections within the project directory and specify the installation path using -p.
Post Installation Tasks

Module Availability: Post installation, the modules from the collection might not immediately be available.

Configuration File: To make new modules available, add the collection's path to the ansible.cfg file under collections_paths.
Verification: Verify the installation and availability of new modules by listing them or using them in tasks.

Commands to Run:

```
ansible --version
cd /usr/local/lib/python3.7/site-packages/ansible_collections
vi ansible.cfg
    COLLECTIONS_PATHS=/usr/local/lib/python3.7/site-packages/ansible_collections
ansible-config dump | grep COLLECTIONS_PATHS
ansible-doc -l | grep -i aws
ansible-galaxy collection list
cd amazon/
ls
cd aws/
vi requirements.txt
pip3 install botocore --user
pip3 install boto3 --user
```

Ansible Dynamic Inventory

```
touch aws_ec2.yml
ansible-doc -t inventory -l | grep aws

https://docs.ansible.com/ansible/latest/collections/amazon/aws/aws_ec2_inventory.html#ansible-collections-amazon-aws-aws-ec2-inventory

vi aws_ec2.yml
        ---
        plugin: amazon.aws.aws_ec2
        aws_access_key:
        aws_secret_key:
        regions:
          - us-east-1
          - us-east-2

ansible-inventory -i aws_ec2.yml --list
ansible-inventory -i aws_ec2.yml --graph
ansible aws_ec2 -m ping -i aws_ec2.yml
```

Login to EC2 -- change tags of one server to Env=dev and on Second Server Env=Prod

vi aws_ec2.yml

```
---
plugin: amazon.aws.aws_ec2
aws_access_key:
aws_secret_key:
regions:
  - us-east-1
  - us-east-2
keyed_groups:
  - key: tags.Env
```

```
ansible-inventory -i aws_ec2.yml --graph
ansible dev -m ping -i aws_ec2.yml
```

```
filters:
  # All instances with their state as `running`
  instance-state-name: running
```

Ansible Roles

Introduction to Ansible Roles

•**Definition:** Ansible roles are structures that facilitate the organization of playbooks by categorizing related tasks, variables, files, templates, and handlers into a recognizable directory structure.

•**Benefit:** Simplifies playbook management by modularizing components, making playbooks easier to understand, maintain, and reuse.

Why Use Ansible Roles?

•**Complexity Management:** Break down complex playbooks into manageable sections.

•**Reusability:** Roles can be reused across different projects and playbooks.

•**Organization:** Provides a standard structure for organizing playbook content.

Creating an Ansible Role

Role Creation Example: Installing and configuring the Apache HTTP server (httpd).
Ansible Galaxy: Use Ansible Galaxy's ansible-galaxy init command to create a role template.
Example: ansible-galaxy init httpd --offline to create an httpd role without fetching any online content.

Role Structure Overview:

defaults: Directory for default variables, lowest priority.
vars: Contains variables with the highest priority.
files: For static files to be copied to managed nodes. Files here can be referenced in tasks without specifying the full path.
templates: Holds Jinja2 template files. Like files, templates can be referenced without the full path.
handlers: Contains handlers, which are tasks triggered by other tasks.
tasks: The main directory where task definitions are stored.
meta: Metadata about the role, including dependencies.
README.md: A markdown file for documenting the role.

handlers

```
---
# handlers file for httpd
- name: Restart WebServer
  ansible.builtin.service:
    name: "{{pkgName.get(ansible_facts.os_family)}}"
    state: restarted
```

tasks

```
---
# tasks file for httpd
- name: Installing package on RHEL Family
  ansible.builtin.yum:
    name: "{{pkgName.get(ansible_facts.os_family)}}"
    state: present
    update_cache: true
  when: ansible_facts.os_family == "RedHat"
- name: Installing package on Debian Family
  ansible.builtin.apt:
    name: "{{pkgName.get(ansible_facts.os_family)}}"
    state: present
    update_cache: true
  when: ansible_facts.os_family == "Debian"
- name: Copying the index.html file to
/var/www/html/index.html
  ansible.builtin.template:
    src: index.j2
    dest: /var/www/html/index.html
  notify: Restart WebServer
```

Templates - Index.j2

```
This is httpd webserver from {{inventory_hostname}} using
roles concept
```

vars

```
---
# vars file for httpd
pkgName:
  RedHat: httpd
  Debian: apache2
```

PlayBook

```
---
- name: Install httpd/apache2
  hosts: all
  gather_facts: true
  vars:
    pkgName:
      RedHat: httpd
      Debian: apache2
  become: yes
  roles:
   - httpd
```

Ansible Vault

Introduction to Ansible Vault

•**Purpose of Ansible Vault**: Encrypts files to securely mask secrets like passwords, keys, or any sensitive data within Ansible playbooks or YAML files.

•**Why Encrypt?**: Direct inclusion of sensitive data (e.g., AWS credentials) in playbooks can lead to security risks, especially if the code is stored in public repositories.

Key Features and Commands of Ansible Vault

Encrypting Files: Use ansible-vault encrypt file.yml to encrypt an entire playbook or YAML file. This requires a password to access the encrypted content.

Decrypting Files: With ansible-vault decrypt file.yml, you can decrypt files if you have the password.

Creating Encrypted Files: ansible-vault create file.yml allows for the creation of a new encrypted file from scratch.

Editing Encrypted Files: ansible-vault edit file.yml enables you to edit an encrypted file without manually decrypting it first.

Viewing Encrypted Files: ansible-vault view file.yml lets you view the contents of an encrypted file.

Rekeying: ansible-vault rekey file.yml is used to change the password of an encrypted file.

Encrypting an AWS Credentials Playbook

Scenario: An AWS EC2 instance launch playbook contains plaintext AWS credentials.
Solution: Encrypt the playbook using ansible-vault encrypt aws_playbook.yml, thereby securing the credentials.

Running an Encrypted Playbook

To execute an encrypted playbook, use ansible-playbook aws_playbook.yml --ask-vault-pass to prompt for the vault password, decrypting the playbook at runtime.

Encrypting Specific Strings

Instead of encrypting the entire playbook, sensitive values (e.g., aws_access_key and aws_secret_key) can be encrypted individually with ansible-vault encrypt_string.
The encrypted strings are then used directly within the playbook, providing a more granular level of security.

Best Practices for Using Variables

Store sensitive data in variable files.
Encrypt only the variable files containing secrets using ansible-vault encrypt vars.yml, leaving the main playbook readable.
Use --ask-vault-pass or a vault password file during playbook execution to decrypt the encrypted variable file on-the-fly.

Encrypting an AWS Credentials Playbook [aws_playbook.yml]

```
---
- name: Launch AWS EC2 Instance
  hosts: localhost
  vars:
    aws_access_key: YOUR_ACCESS_KEY
    aws_secret_key: YOUR_SECRET_KEY
  tasks:
    - name: Launch an instance
      amazon.aws.ec2_instance:
        aws_access_key: "{{ aws_access_key }}"
        aws_secret_key: "{{ aws_secret_key }}"
        region: us-east-1
        instance_type: t2.micro
        image: ami-0c55b159cbfafe1f0
```

```
ansible-vault encrypt aws_playbook.yml
```

Running an Encrypted Playbook

```
ansible-playbook aws_playbook.yml --ask-vault-pass
```

Encrypting Specific Strings [Like AWS Keys]

```
ansible-vault encrypt_string 'YOUR_ACCESS_KEY'
ansible-vault encrypt_string 'YOUR_SECRET_KEY'
```

Insert the output encrypted strings into your playbook:

```
---
- name: Launch AWS EC2 Instance with Encrypted Credentials
  hosts: localhost
  vars:
    aws_access_key: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          663864396532363739643732623231663937...
    aws_secret_key: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          346239383164396462663736343761626639...
  tasks:
    - name: Launch an instance
      amazon.aws.ec2_instance :
        aws_access_key: "{{ aws_access_key }}"
        aws_secret_key: "{{ aws_secret_key }}"
        region: us-east-1
        instance_type: t2.micro
        image: ami-0c55b159cbfafe1f0
```

ansible-playbook aws_playbook.yml --ask-vault-pass

Best Practices for Using Variables

vars.yml

```
aws_access_key: YOUR_ACCESS_KEY
aws_secret_key: YOUR_SECRET_KEY
```

```
ansible-vault encrypt vars.yml
```

```
---
- name: Launch AWS EC2 Instance Using Encrypted Vars
  hosts: localhost
  vars_files:
    - vars.yml
  tasks:
    - name: Launch an instance
      amazon.aws.ec2_instance :
       aws_access_key: "{{ aws_access_key }}"
       aws_secret_key: "{{ aws_secret_key }}"
       region: us-east-1
       instance_type: t2.micro
       image: ami-0c55b159cbfafe1f0
```

```
ansible-playbook ec2_launch.yml --ask-vault-pass
```

Raw Module

Understanding Ansible's Raw Module

Introduction to the Raw Module

- **Purpose:** Executes low-level commands on managed nodes.
- **Utility:** Particularly useful when Python is not installed on managed nodes.
- **Ansible and Python:** Ansible is built with Python and typically requires Python on managed nodes to execute modules.

When to Use the Raw Module

- **No Python Scenario:** If managed nodes don't have Python installed, the raw module allows you to perform simple operations.
- **Fallback Option:** Acts as a fallback for executing commands when standard modules like shell or command cannot be used due to the absence of Python.

```
ansible test_group -m ansible.builtin.raw -a 'uname -a'
```

```
---
- name: Execute commands on nodes without Python using FQCN
  hosts: test_group
  gather_facts: false  # Disabling as gathering facts requires Python

  tasks:
    - name: Check Java version using raw module
      ansible.builtin.raw: java -version
      register: java_version

    - name: Display Java version using debug module
      ansible.builtin.debug:
        var: java_version.stdout_lines
```

🙏 🙏 Namaskaram 🙏 🙏