PW SKILLS | DevOps and Cloud Computing

Ansible Inventory, Modules, and Playbook Essentials

Objective

- Differentiate between static and dynamic inventories and manage inventory files.
- Utilize common Ansible modules to automate package installation, file management, and services.
- Write and execute Ansible playbooks with YAML syntax, tasks, variables, and loops.
- Implement handlers and notifications to automate service restarts and actions.

PW SKILLS

Explaining the purpose of an inventory in Ansible (defining managed nodes).

Let’s see

In Ansible, an inventory is a fundamental component that defines the managed nodes (or "hosts") that Ansible automates. Its primary purpose is to organize and specify these nodes so that tasks can be executed efficiently across one or more systems.

Its purpose is to:

- Identify Managed Nodes
- Organize Hosts into Groups
- Support Variables and Patterns
- Enable Flexibility

PW SKILLS

Differentiate between: Static & Dynamic Inventory

Static v/s Dynamic

Aspect	Static Inventory	Dynamic Inventory
Definition	A predefined list of hosts written in a static file (e.g., INI or YAML format).	Automatically fetches host information from external sources like cloud providers or APIs.
Management	Requires manual updates whenever hosts are added, removed, or changed.	Updates automatically as infrastructure changes (e.g., adding/removing VMs).
Use Cases	Suitable for small, stable environments with minimal changes.	Ideal for dynamic environments like cloud-based setups or auto-scaling systems.
Format	Simple text files with hostnames/IPs grouped under categories.	Scripts or plugins that return inventory data in JSON format.
Examples	`[webservers]` followed by host IPs or names.	AWS EC2 plugin dynamically fetching all instances tagged as "webserver."
Advantages	Easy to set up and understand; no additional tools required.	Reduces human error, saves time, and scales well with large or dynamic infrastructures.

PW SKILLS

Discuss use cases for each approach

Let’s discuss

Static Inventory

Static inventories are predefined files listing hosts and groups. Common use cases include:

- Small, Stable Environments
- Testing and Development
- Isolated Environments
- Simple Configurations

Let’s discuss

Dynamic Inventory

Dynamic inventories are generated automatically by plugins or scripts. Key use cases include:

- Cloud-Based Infrastructure
- Auto-Scaling Systems
- Temporary Resources
- Large, Complex Setups

PW SKILLS

Explaining the structure of a static inventory file (/etc/ansible/hosts).

Static inventory file

The structure of a static inventory file in Ansible, typically located at /etc/ansible/hosts, is straightforward and organized into groups and hosts. Below is a concise explanation:

Structure of a Static Inventory File:

- **Group Names:** Defined in square brackets ([group name]) to categorize hosts.
- **Hosts:** Listed under each group using hostnames or IP addresses, one per line.
- **Variables:** Group or host-specific variables can be included directly or referenced from external files.

Static inventory file

Example:

```
# Individual host
mail.example.com

# Group of web servers
[webservers]
foo.example.com
bar.example.com

# Group of database servers
[dbservers]
one.example.com
two.example.com
three.example.com
```

PW SKILLS

Demonstrating defining groups and aliases

Let’s do it

A static inventory file can define groups and aliases for better organization and targeting of hosts. Below is an example:

```
# Individual host with an alias
web1 ansible_host=192.168.1.10

# Group of web servers
[webservers]
web1
web2 ansible_host=192.168.1.11

# Group of database servers
[dbservers]
db1 ansible_host=192.168.1.20
db2 ansible_host=192.168.1.21

# Nested group (parent group)
[production:children]
webservers
dbservers
```

Let’s do it

Explanation:

- **Aliases:** web1 and db1 are aliases for the actual IPs or hostnames defined using ansible host.
- **Groups:** [webservers] and [dbservers] categorize hosts into logical groups.
- **Nested Groups:** [production:children] combines webservers and dbservers into a parent group for easier management.

PW SKILLS

Showing how to test inventory connectivity

Let’s see

To test inventory connectivity in Ansible, you can use the ping module, which verifies whether Ansible can connect to the managed nodes. Here's how to do it:

Command to Test Connectivity:

```
ansible all -m ping -i /path/to/inventory
```

Explanation:

- **all:** Targets all hosts in the inventory.
- **-m ping:** Uses the ping module to check connectivity.
- **-i /path/to/inventory:** Specifies the inventory file (optional if using the default /etc/ansible/hosts).

Let’s see

Example Output:

```
host1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
host2 | UNREACHABLE! => {
    "msg": "Failed to connect to the host via ssh"
}
```

- This confirms which hosts are reachable and highlights any connectivity issues.

PW SKILLS

Explaining the need for dynamic inventories in cloud environments

Dynamic inventories

Dynamic inventories are essential in cloud environments due to the following reasons:

1. Real-Time Updates
2. Scalability
3. Integration with Cloud Providers
4. Efficiency and Automation

PW SKILLS

Demonstrating using cloud inventory scripts

Let’s do it

Dynamic inventory scripts fetch real-time data from cloud providers (AWS, GCP, Azure) and display the infrastructure graphically. Below are examples for each:

AWS:

```
ansible-inventory -i aws_ec2.yml --graph
```

- **Purpose:** Fetches EC2 instances dynamically using the aws ec2.yml plugin configuration.
- **Output:** Displays a tree structure of AWS hosts grouped by tags, regions, or other criteria.

GCP:

```
ansible-inventory -i gcp.yml --graph
```

- **Purpose:** Retrieves GCP instances dynamically using the gcp.yml plugin configuration.
- **Output:** Shows a graphical representation of GCP hosts grouped by predefined criteria like zones or labels.

Azure:

```
ansible-inventory -i azure_rm.yml --graph
```

- **Purpose:** Dynamically lists Azure resources using the azure rm.yml plugin configuration.
- **Output:** Visualizes Azure hosts in a hierarchical format based on resource groups, tags, or other parameters.

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Introducing key Ansible modules

Ansible modules

File Management

- **file:** Manages files, directories, and symlinks (e.g., create, delete, or modify permissions).
- **copy:** Copies files from the control node to managed nodes.
- **template:** Deploys configuration files using Jinja2 templates.

Package Management

- **apt:** Manages packages on Debian-based systems (e.g., Ubuntu).
- **yum:** Handles packages on RHEL-based systems (e.g., CentOS).
- **dnf:** Used for package management on newer RHEL-based systems.

Ansible modules

Service Control

- **service:** Manages services (start, stop, restart) abstracting init systems.
- **systemd:** Directly controls services using the systemd init system.

Shell Execution

- **command:** Executes commands on remote hosts without a shell.
- **shell:** Runs shell commands with access to shell features like pipes and redirects.

User Management

- **user:** Manages user accounts (create, modify, or delete users).
- **group:** Handles group creation and management.

PW SKILLS

Demonstrating running ad-hoc commands

Let’s do it

Ansible ad-hoc commands are simple, one-time commands used to automate tasks on managed nodes without the need for creating a playbook. These commands are ideal for quick operations like rebooting servers, managing services, or installing packages. Below is a demonstration of running ad-hoc commands:

Syntax

The basic syntax for an ad-hoc command is:

```
ansible <hosts> -m <module> -a "<arguments>"
```

Where:

<hosts>: Target hosts (e.g., all, webservers).

<module>: The Ansible module to use (e.g., ping, yum).

<arguments>: Parameters for the module.

Let’s do it

Examples

1. Ping all hosts:
To check connectivity with all hosts in the inventory:

```
ansible all -m ping
```

Output:

```
node1 | SUCCESS => {
    "ping": "pong"
}
```

Let’s do it

2. Gather system facts:
To collect information about hosts:

```
ansible all -m setup
```

3. Install a package:
To install wget on target nodes:

```
ansible webservers -m yum -a "name=wget state=present" --become
```

Let’s do it

4. Create a directory:

To create a directory with specific permissions:

```
ansible webservers -m file -a "dest=/path/to/dir
mode=755 state=directory"
```

5. Restart a service:

To restart the httpd service:

```
ansible webservers -m service -a "name=httpd state=restarted"
```

Let’s do it

6. Run commands as non-root user:

Use the --become flag for root privileges and -K to prompt for the password:

```
ansible all -m command -a "uptime" --become -K
```

Parallel Execution

To run commands on multiple servers concurrently, use the -f flag (default is 5 parallel executions):

```
ansible prod -m apt -a "name=wget state=present" --become -f 10
```

- Ad-hoc commands are efficient for troubleshooting, testing modules, or performing quick tasks across multiple systems.

PW SKILLS

Explaining YAML syntax and playbook structure

Let’s see

YAML Syntax

YAML (YAML Ain't Markup Language) is a human-readable data serialization language often used for configuration files. Below are its key syntax rules:

1. Key-Value Pairs:

- **Defined as key:** value.

Example:

```
name: John Doe
age: 30
```

Let's see

2. Lists:

- Represented with a dash (-) and space.

Example:

```
fruits:
  - Apple
  - Orange
  - Banana
```

3. Dictionaries (Maps):

- Nested key-value pairs.

Example:

```
address:
  city: New York
  zip: 10001
```

Let's see

4. Comments:

- Begin with #.

Example:

```
# This is a comment
version: 1.0
```

5. Multiline Strings:

- Use > for folded style (no newlines) or | for literal style (preserve newlines).

Example:

```
description: >
  This is a long description
  that spans multiple lines.
```

Let’s see

Ansible Playbook Structure

Ansible playbooks, written in YAML, automate tasks on managed nodes. Below is the structure:

1. Header:

Starts with --- (optional but recommended).

2. Play Definition:

Defines the hosts and tasks.

Example:

```
---
- name: Example Playbook
  hosts: all
  become: yes  # Run tasks as root
  tasks:
    - name: Install Apache
      yum:
        name: httpd
        state: present

    - name: Start Apache Service
      service:
        name: httpd
        state: started
```

Let’s see

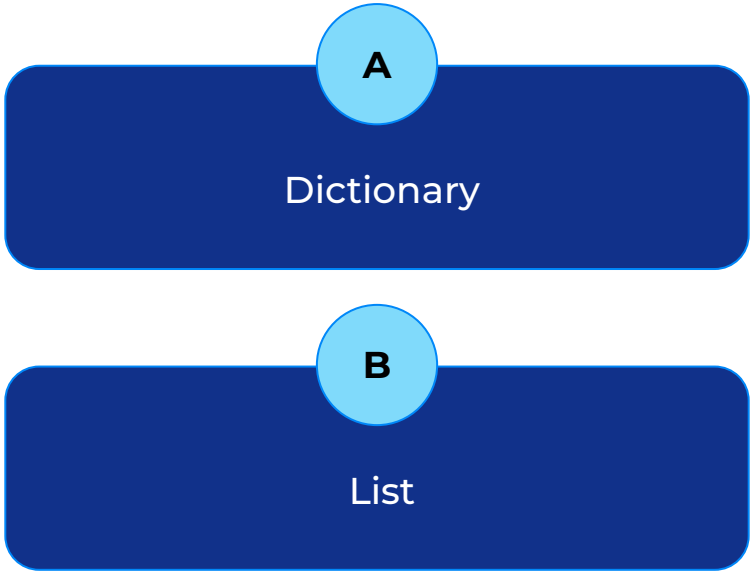
3. Key Sections:

- **name:** Describes the play or task.
- **hosts:** Specifies target hosts.
- **tasks:** A list of actions to perform.
- **vars:** Variables to use in the playbook.
- **handlers:** Define actions triggered by tasks.

This structure ensures readability and modularity in automation processes.

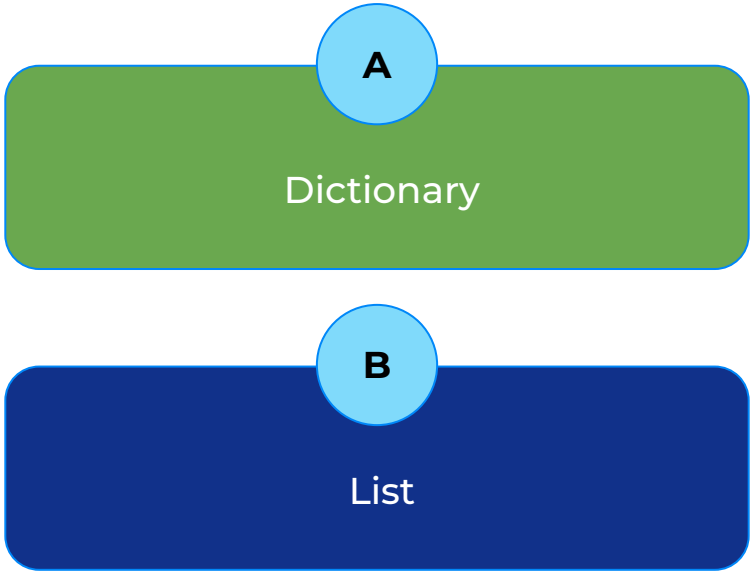
Pop Quiz

Q. Which type of data structure does YAML use for key-value pairs?



Pop Quiz

Q. Which type of data structure does YAML use for key-value pairs?



PW SKILLS

Define tasks, handlers, and plays in playbooks.

Let's see

1. Tasks:

- A task is the smallest unit of action in a playbook.
- It defines a specific operation (e.g., installing a package, starting a service) using Ansible modules.

Example:

```
- name: Install Apache
  apt:
    name: apache2
    state: present
```

Let's see

2. Handlers:

- Handlers are special tasks triggered by other tasks using the notify directive.
- They are used for operations that need to run only when there is a change (e.g., restarting a service after configuration changes).

Example:

```
tasks:
  - name: Update configuration
    copy:
      src: /path/to/config
      dest: /etc/config
    notify: Restart Apache

handlers:
  - name: Restart Apache
    service:
      name: apache2
      state: restarted
```

Let's see

3. Plays:

- A play is a collection of tasks mapped to specific hosts or groups of hosts.
- It defines what tasks to execute and where (on which hosts) they should run.
- Plays include parameters like hosts, become, and tasks.

Example:

```
- name: Configure Web Servers
  hosts: webservers
  become: yes
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
```

PW SKILLS

Demonstrating writing a basic playbook

Let’s do it

Basic Ansible Playbook Example

- Below is a demonstration of writing a simple Ansible playbook that installs and starts the Apache web server on a group of hosts.

Playbook Structure:

```
---
- name: Install and Start Apache Web Server
  hosts: webservers  # Target group of hosts
  become: yes        # Run tasks with elevated privileges (sudo)

  tasks:             # List of tasks to be executed
    - name: Install Apache
      apt:           # Using the apt module for package management
        name: apache2
        state: present  # Ensure Apache is installed

    - name: Start Apache Service
      service:     # Using the service module to manage services
        name: apache2
        state: started  # Ensure the service is running

    - name: Enable Apache to start on boot
      service:
        name: apache2
        enabled: yes  # Ensure the service is enabled at boot
```

Running the Playbook

To execute this playbook, save it as apache install.yml, then run the following command:

```
ansible-playbook apache_install.yml -i inventory_file
```

Replace ‘inventory file’ with your actual inventory file path. This command will apply the defined tasks to all hosts in the ‘webservers’ group.

PW SKILLS

Explaining how to run a playbook

How to run playbook

1. **Prepare Inventory:**
- Create an inventory file (e.g., inventory.ini) that lists the target hosts.

Example:

```
[webservers]
server1.example.com
server2.example.com
```

2. **Write the Playbook:**
- Create a YAML file (e.g., playbook.yml) containing plays and tasks.

Example:

```yaml
---
- name: Install Apache
  hosts: webservers
  become: yes
  tasks:
    - name: Install Apache package
      apt:
        name: apache2
        state: present
    - name: Start Apache service
      service:
        name: apache2
        state: started
```

3. Run the Playbook:

- Use the ansible-playbook command to execute the playbook.

Syntax:

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

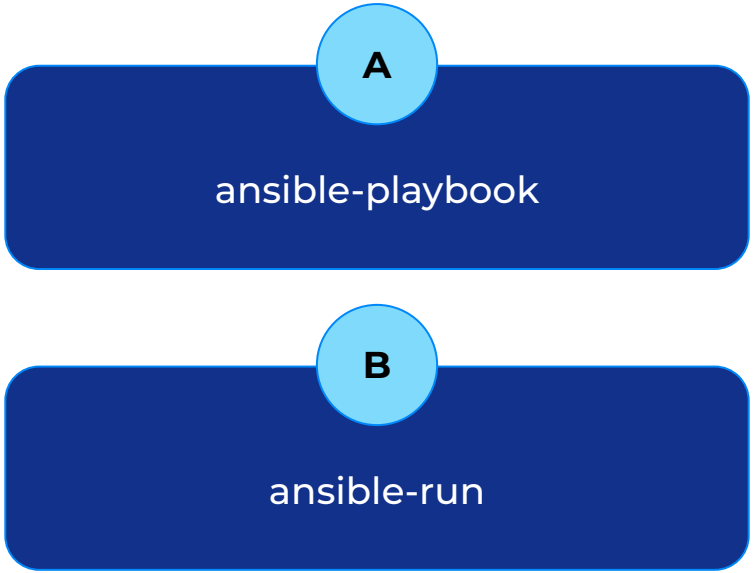
4. Example Command:

```
ansible-playbook -i inventory.ini playbook.yml -u admin --check
```

- The playbook will execute tasks sequentially on the hosts listed in the inventory file.

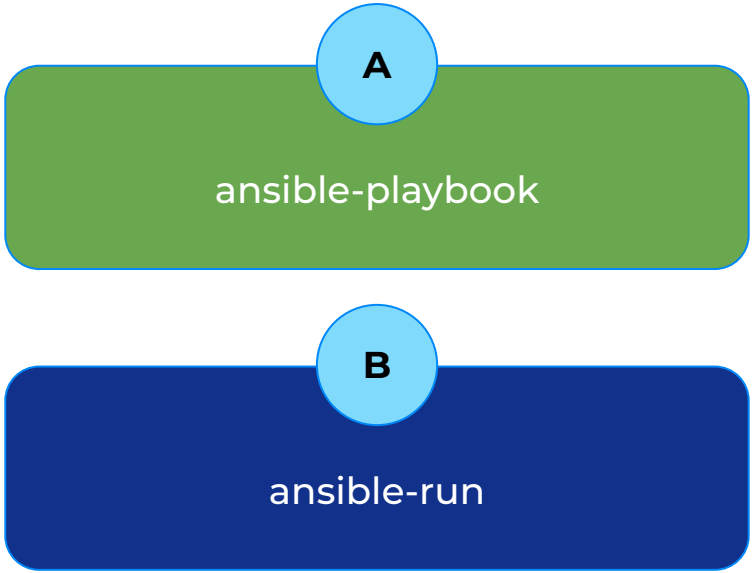
Pop Quiz

Q. Which command is used to execute an Ansible playbook?



Pop Quiz

Q. Which command is used to execute an Ansible playbook?



PW SKILLS

Explain the use of variables in playbooks

Variables in playbook

Variables in Ansible playbooks are key-value pairs used to make tasks dynamic and reusable. They allow customization for different environments, hosts, or runtime conditions.

Key Features of Variables:

1. Definition:
Variables can be defined in the vars section of a playbook, inventory files, external variable files, or passed at runtime.

Example:

```
vars:
  username: admin
  package_name: httpd
```

Variables in playbook

2. Usage:
Variables are referenced using {{ variable name }}.

Example:

```
tasks:
  - name: Install a package
    ansible.builtin.package:
      name: "{{ package_name }}"
      state: present
```

3. Sources:

- **Playbooks (vars):** Defined directly in the playbook.
- **Inventory:** Host-specific or group-specific variables.
- **External Files:** Imported using vars files.
- **Runtime:** Passed via --extra-vars (e.g., ansible-playbook playbook.yml --extra-vars "var=value")

Variables in playbook

4. Dynamic Behavior:
Variables can be conditionally loaded based on facts (e.g., OS type) or runtime inputs.

Example:

```
vars_files:
  - "vars/{{ ansible_facts['os_family'] }}.yml"
```

Variables in playbook

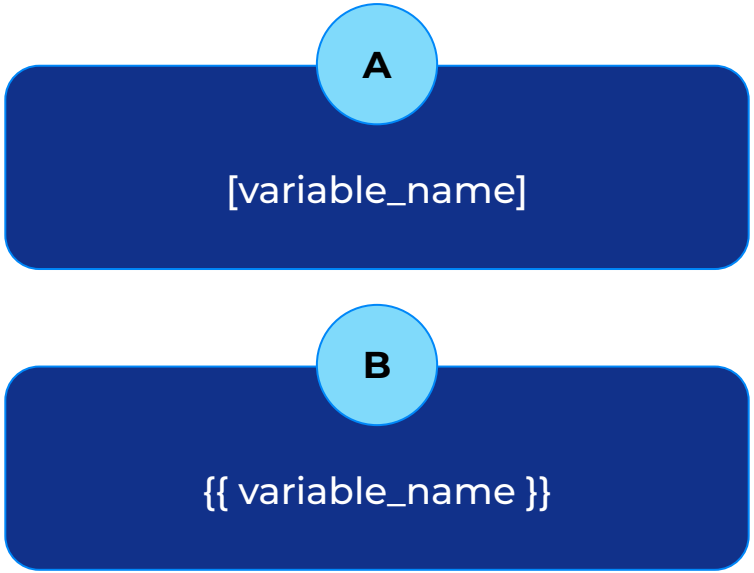
Example Playbook with Variables:

```
---
- name: Install and Start Web Server
  hosts: webservers
  become: yes
  vars:
    package_name: apache2
    service_name: apache2
  tasks:
    - name: Install Apache
      ansible.builtin.package:
        name: "{{ package_name }}"
        state: present

    - name: Start Apache Service
      ansible.builtin.service:
        name: "{{ service_name }}"
        state: started
        enabled: yes
```

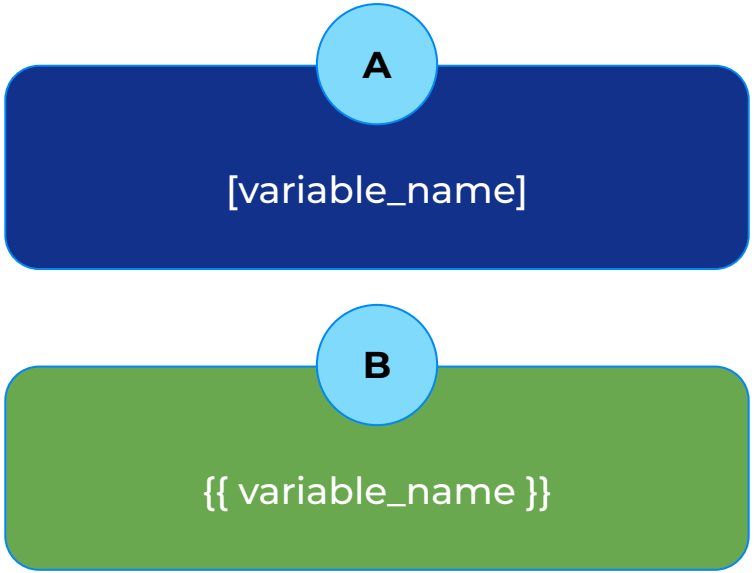
Pop Quiz

Q. Which syntax is used to reference a variable in an Ansible playbook?



Pop Quiz

Q. Which syntax is used to reference a variable in an Ansible playbook?



PW SKILLS

Introducing facts and ansible facts

Facts

Facts are system information gathered by Ansible about managed nodes (hosts). These facts include details such as IP addresses, operating systems, hardware specifications, and more. They are automatically collected using the setup module during playbook execution and stored in the ansible facts variable.

Key Points About Facts:

- Facts provide real-time data about hosts, enabling dynamic decision-making in playbooks.
- Facts are stored in JSON format within the ansible facts variable.
- Facts are collected automatically before playbook execution using the setup module.
- Facts enable conditional tasks based on the host's state or properties, improving flexibility and efficiency.

PW SKILLS

Demonstrating conditionals

Conditionals

Conditionals in Ansible allow you to execute tasks based on specific conditions, enhancing the flexibility of your playbooks. You can use the when directive to define these conditions.

Basic Syntax

The syntax for using conditionals is:

```
- name: Task description
  module_name:
    parameters
  when: condition
```

Conditionals

Example:

1. Using Facts:

Execute a task only if the host's operating system is Ubuntu.

```
- name: Install Apache on Ubuntu
  apt:
    name: apache2
    state: present
  when: ansible_facts['os_family'] == 'Debian'
```

Conditionals

2. Using Variables:

Run a task based on a variable value.

```
vars:
  install_nginx: true

tasks:
  - name: Install Nginx
    apt:
      name: nginx
      state: present
    when: install_nginx == true
```

Conditionals

3. Combining Conditions:

Use logical operators to combine multiple conditions.

```
- name: Install package if OS is RedHat and version is 7 or higher
  yum:
    name: httpd
    state: present
  when: ansible_facts['os_family'] == 'RedHat' and
ansible_facts['distribution_version'] | version_compare('7', '>=')
```

Conditionals

4. Negation:

Execute a task when a condition is false.

```
- name: Ensure Apache is not installed on non-Ubuntu systems
  apt:
    name: apache2
    state: absent
  when: ansible_facts['os_family'] != 'Debian'
```

PW SKILLS

Demonstrating loops

Let’s do it

Loops in Ansible allow you to execute a task multiple times with different values, making your playbooks more efficient and concise. You can use the loop directive to iterate over lists, dictionaries, or ranges.

Basic Syntax

The syntax for using loops is:

```
- name: Task description
  module_name:
    parameters
  loop: list_of_items
```

Let’s do it

Examples of Loops

1. Looping Over a List:
Install multiple packages using a list.

```
- name: Install multiple packages
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - git
    - vim
    - curl
```

Let’s do it

2. Looping Over a Dictionary:

Create users based on a dictionary of usernames and their shell.

```
vars:
  users:
    alice:
      shell: /bin/bash
    bob:
      shell: /bin/zsh

tasks:
  - name: Create users with specific shells
    user:
      name: "{{ item.key }}"
      shell: "{{ item.value.shell }}"
      state: present
    loop: "{{ users | dict2items }}"
```

Let’s do it

3. Looping with Index:

Access the index of the current iteration using loop.index.

```
- name: Create directories with index
  file:
    path: "/tmp/dir{{ item }}"
    state: directory
  loop: "{{ range(1, 6) | list }}"
```

Let’s do it

4. Nested Loops:

Loop through a list of items and perform actions for each item in another list.

```
vars:
  services:
    - webserver
    - database

tasks:
  - name: Start services on hosts
    service:
      name: "{{ item }}-service"
      state: started
    loop: "{{ services }}"
```

PW SKILLS

How handlers automate service restarts or actions after changes

Let’s see

In an Ansible playbook, handlers automate service restarts or actions by responding to changes triggered by specific tasks. Here's how they work:

1. Definition and Notification:

- Handlers are defined under the handlers section in a playbook and are triggered using the notify keyword in tasks.

2. Execution on Change:

- Handlers execute only when notified, ensuring that actions like service restarts happen only if a change occurs (e.g., a file modification). This prevents unnecessary restarts and maintains idempotence.

3. Single Execution:

- Even if multiple tasks notify the same handler during a playbook run, the handler executes only once at the end of the playbook to optimize performance and avoid redundant actions.

4. Examples:

A task updating a configuration file can notify a handler like this:

- In this example, the service restarts only if the configuration file changes.

```
tasks:
  - name: Update config
    template:
      src: myservice.conf.j2
      dest: /etc/myservice/myservice.conf
    notify: Restart myservice

handlers:
  - name: Restart myservice
    service:
      name: myservice
      state: restarted
```

PW SKILLS

Demonstrating a playbook with handlers

Let’s do it

Here is an example of an Ansible playbook that demonstrates the use of handlers to automate service restarts after configuration changes:

```
---
- name: Demonstrate Handlers in Ansible
  hosts: all
  become: yes

  tasks:
    - name: Update Nginx configuration
      copy:
        src: /path/to/nginx.conf
        dest: /etc/nginx/nginx.conf
      notify: Restart Nginx

    - name: Create a test file
      file:
        path: /tmp/testfile.txt
        state: touch

  handlers:
    - name: Restart Nginx
      service:
        name: nginx
        state: restarted
```

PW SKILLS

How notifications trigger handlers only when changes occur

Let’s see

In Ansible, notifications trigger handlers only when changes occur in the associated tasks. Here's how it works:

1. Notify Directive:
The notify keyword in a task links the task to a handler. A handler is triggered only if the task results in a "changed" state, meaning the task modified something on the target system.

2. Execution Control:
Handlers are executed at the end of the playbook run, ensuring efficient and controlled operations. Even if multiple tasks notify the same handler, it runs only once to avoid redundant actions.

Let's see

3. Example: In this example, the handler Restart MyApp will execute only if the configuration file changes.

```
tasks:
  - name: Update configuration file
    copy:
      src: /path/to/config.conf
      dest: /etc/myapp/config.conf
    notify: Restart MyApp

handlers:
  - name: Restart MyApp
    service:
      name: myapp
      state: restarted
```

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessio
- Review your lecture notes
- Note down questions and queries regarding this session ar
  consult the teaching assistants

Thanks !