**1) Ansible Basic:**

* **What is Ansible?**

**Ansible** is an open-source **automation tool** used for **configuration management, application deployment, and infrastructure orchestration**. It helps **automate repetitive IT tasks** such as software installation, system updates, and cloud provisioning.

### **Key Features of Ansible:**

✅ **Agentless:** No need to install agents on remote systems; it uses SSH for communication.  
✅ **Declarative Language:** Uses YAML-based **playbooks** to define automation tasks.  
✅ **Idempotent:** Ensures tasks execute only when needed, avoiding unnecessary changes.  
✅ **Scalable:** Can manage thousands of servers efficiently.  
✅ **Extensible:** Supports **modules, roles, and plugins** for various automation needs.

### **Common Use Cases:**

* **Server Configuration Management** (e.g., setting up web servers, databases)
* **CI/CD Automation** (integrating with Jenkins, Git, etc.)
* **Cloud Provisioning** (AWS, Azure, GCP automation)
* **Security & Compliance** (patching, enforcing policies)
* **Application Deployment** (managing microservices, Kubernetes automation)
* **Ansible architecture (Control Node, Managed Nodes, Inventory)**

Ansible follows a **simple and agentless** architecture that consists of the following key components:

**1️⃣ Control Node (Master)**

* The **main machine** where Ansible is installed.
* Runs Ansible **commands, playbooks, and modules** to automate tasks.
* Uses **SSH (Linux) or WinRM (Windows)** to communicate with remote systems.
* Only **one** control Node is required to manage multiple machines.

✅ **Example:** Your laptop or a dedicated server running Ansible.

### **2️⃣ Managed Nodes (Target Machines)**

* The **remote machines** (Linux, Windows, Network devices, Cloud, etc.) controlled by Ansible.
* No need to install **Ansible agents** on these machines (Agentless).
* Ansible pushes configurations to these nodes and ensures compliance.

✅ **Example:** Web servers, databases, cloud instances, networking devices.

### **3️⃣ Inventory (Hosts File)**

* A file that lists the **IP addresses or hostnames** of managed nodes.
* Can be in **INI, YAML, or dynamic inventory** formats.
* Defines **groups** of servers (e.g., web\_servers, db\_servers).

✅ **Example of a simple inventory file (inventory.ini):**

ini

[web\_servers]  
192.168.1.10  
192.168.1.11  
[db\_servers]  
db1.example.com  
db2.example.com

### **Other Key Components:**

🔹 **Modules:** predefined scripts to perform tasks (e.g., install software, manage users).  
🔹 **Playbooks:** YAML files containing automation instructions (tasks).  
🔹 **Plugins:** Extend Ansible’s functionality (logging, authentication, etc.).  
🔹 **Facts:** Gather system information about managed nodes.  
🔹 **Ad-Hoc Commands:** One-time quick commands without playbooks

### **How Ansible Works?**

1️⃣ The control **node** reads the **inventory** to find managed nodes.  
2️⃣ It establishes a connection using **SSH** (Linux) or **WinRM** (Windows).  
3️⃣ The **Playbook** or **ad-hoc command** runs on the managed nodes.  
4️⃣ The modules execute tasks and return results.  
5️⃣ Ansible ensures **idempotency**, meaning changes happen only when necessary.

* **YAML syntax**

YAML (**Yet Another Markup Language**) is a human-readable format used for writing **Ansible Playbooks, inventory files, and configurations**. It follows simple indentation rules.

## **1️⃣ Basic YAML Rules ✅**

* Use **spaces for indentation** (not tabs).
* Start with --- (optional but recommended).
* Keys and values are separated by a **colon (:) and a space**.
* Lists start with a **dash (-)**.
* Strings can be **quoted or unquoted**.

📌 **Example YAML Structure:**

yaml

---  
name: "John Doe"  
age: 30  
skills:  
 - Ansible  
 - Docker  
 - Kubernetes

## **2️⃣ Lists in YAML 📋**

Use **dashes (-)** for lists.

yaml

fruits:  
 - Apple  
 - Banana  
 - Orange

Equivalent JSON:

json

{  
 "fruits": ["Apple", "Banana", "Orange"]  
}

## **3️⃣ Dictionaries (Key-Value Pairs) 🔑**

Dictionaries use key-value pairs.

yaml

person:  
 name: "Alice"  
 age: 25  
 city: "New York"

Equivalent JSON:

json

{  
 "person": {  
 "name": "Alice",  
 "age": 25,  
 "city": "New York"  
 }  
}

## **4️⃣ Nested Structures (Lists Inside Dictionaries) 🏗️**

You can nest lists inside dictionaries.

yaml

servers:  
 web:  
 - server1  
 - server2  
 database:  
 - db1  
 - db2

## **5️⃣ Boolean, Null, and Numbers 🔢**

yaml

is\_enabled: true  
is\_disabled: false  
count: 10  
price: 99.99  
value: null # Represents empty or None

## **6️⃣ Multi-Line Strings 📜**

Use | for multi-line blocks (**preserves line breaks**).

yaml

description: |  
 This is a long   
 multi-line description.

Use > for multi-line blocks (**removes line breaks**).

yaml

description: >  
 This is a long   
 multi-line description.

Becomes: "This is a long multi-line description."

## **7️⃣ Ansible Playbook Example 🚀**

---  
- name: Install Nginx on Web Servers  
 hosts: web\_servers  
 become: yes  
 tasks:  
 - name: Install Nginx  
 apt:  
 name: nginx  
 state: present

**2) Inventory Management:**

* **Static and dynamic inventory (/etc/ansible/hosts, inventory scripts)**

### **Ansible Inventory: Static vs. Dynamic 📂**

Ansible uses an **inventory file** to define the **managed nodes (hosts)** it controls. There are two types of inventories:

## **1️⃣ Static Inventory (Manual Configuration) 📄**

A **static inventory** is a **fixed list of hosts** written in a file (e.g., /etc/ansible/hosts). It does not change unless manually updated.

### **Example: Static Inventory (/etc/ansible/hosts or inventory.ini)**

ini

[web\_servers]  
192.168.1.10  
192.168.1.11  
  
[db\_servers]  
db1.example.com  
db2.example.com  
  
[load\_balancers]  
lb1 ansible\_host=192.168.1.100 ansible\_user=ubuntu ansible\_port=22

✅ **Features of Static Inventory:**  
✔ Simple and easy to configure  
✔ Good for **small environments**  
✔ Hosts and groups are **manually managed**

## **2️⃣ Dynamic Inventory (Automated Discovery) 🤖**

A **dynamic inventory** fetches **real-time** data from **cloud providers, databases, or external APIs**. This is useful when managing **cloud instances** (AWS, Azure, GCP, etc.) that frequently change.

### **Example: Dynamic Inventory Using a Script**

bash

#!/usr/bin/env python3  
  
import json  
  
inventory = {  
 "\_meta": {  
 "hostvars": {}  
 },  
 "web\_servers": {  
 "hosts": ["web1.example.com", "web2.example.com"]  
 },

"db\_servers": {  
 "hosts": ["db1.example.com", "db2.example.com"]  
 }  
}  
  
print(json.dumps(inventory))

✅ **Features of Dynamic Inventory:**  
✔ Automatically **discovers** new and removed hosts  
✔ Fetches **real-time host details** (IP, OS, tags, etc.)  
✔ Used in **cloud environments** with **AWS, Azure, GCP, VMware, etc.**

## **3️⃣ Using Built-in Dynamic Inventory Plugins 🌐**

Ansible provides **pre-built inventory plugins** for cloud providers.

### **Example: AWS Dynamic Inventory (aws\_ec2)**

bash

ansible-inventory -i aws\_ec2.yml --list

📌 **AWS EC2 Inventory Example (aws\_ec2.yml):**

yaml

plugin: amazon.aws.aws\_ec2  
regions:  
 - us-east-1  
keyed\_groups:  
 - key: tags.Name  
 prefix: "instance\_"

## **4️⃣ Running Inventory Commands 🖥️**

📌 **List Inventory Hosts:**

bash

ansible-inventory -i inventory.ini --list

📌 **Ping All Hosts:**

bash

ansible -i inventory.ini all -m ping

## **🔎 Static vs. Dynamic Inventory Comparison**

| **Feature** | **Static Inventory** | **Dynamic Inventory** |
| --- | --- | --- |
| **Setup** | Manual | Automated |
| **Flexibility** | Fixed list | Real-time discovery |
| **Best For** | Small setups | Cloud environments |
| **Scalability** | Low | High |
| **Data Source** | Local file (/etc/ansible/hosts) | External sources (API, Cloud, DB) |

### **🔹 When to Use Which?**

✔ **Use Static Inventory** for small, stable environments.  
✔ **Use Dynamic Inventory** for large, cloud-based, or dynamic environments.

* **Grouping hosts, patterns ([webservers], [databases])**

Ansible allows you to **group hosts** into categories, making it easier to manage and execute tasks on specific sets of machines.

## **1️⃣ Defining Groups in Inventory (inventory.ini)**

You can group hosts using **square brackets [ ]** in a **static inventory file**.

📌 **Example: Grouping Web and Database Servers**

ini

[web\_servers]  
192.168.1.10  
192.168.1.11  
web1.example.com  
web2.example.com  
  
[db\_servers]  
db1.example.com  
db2.example.com  
  
[load\_balancers]  
lb1 ansible\_host=192.168.1.100 ansible\_user=ubuntu ansible\_port=22

🔹 **Use Case:** Run Ansible tasks only on web\_servers, db\_servers, or load\_balancers groups.

## **2️⃣ Grouping Inside Another Group (Children Groups) 👨‍👩‍👦**

You can create **parent groups** using [groupname:children].

📌 **Example: Defining an "Application Servers" Group**

ini

[app\_servers:children]  
web\_servers  
db\_servers

🔹 **Now, running Ansible on app\_servers will include both web and database servers.**

## **3️⃣ Setting Variables for Groups (group\_vars/ Directory) 📝**

You can assign **variables to groups** inside group\_vars/ for better management.

📌 **Example: Defining Group Variables for Web Servers (group\_vars/web\_servers.yml)**

yaml

---  
ansible\_user: ubuntu  
ansible\_ssh\_private\_key\_file: ~/.ssh/web\_key.pem

🔹 **All hosts in web\_servers will use ubuntu as the SSH user.**

## **4️⃣ Using Patterns to Target Groups & Hosts 🎯**

Ansible supports **patterns** to target specific hosts or groups dynamically.

### **Common Patterns:**

| **Pattern** | **Description** |
| --- | --- |
| all | All hosts in the inventory |
| web\_servers | Only the web\_servers group |
| web1.example.com | A specific host |
| web\_servers:!web1.example.com | All web\_servers except web1.example.com |
| web\_servers:&databases | Hosts that are in both groups |
| web\_servers:children | Lists all child groups under web\_servers |

### **Example: Running Commands on Specific Groups**

bash

# Ping only web servers  
ansible -i inventory.ini web\_servers -m ping  
  
# Run tasks on all servers except database servers  
ansible -i inventory.ini all:!db\_servers -m ping  
  
# Run tasks only on common hosts in web\_servers & db\_servers  
ansible -i inventory.ini "web\_servers:&db\_servers" -m ping

## **🔹 Summary**

✔ Use [group\_name] to define groups.  
✔ Use [group\_name:children] to create **parent groups**.  
✔ Store **variables** in group\_vars/ for better management.  
✔ Use **patterns** to run Ansible tasks efficiently.

3) **Modules & Ad-hoc Commands**

| Module | Description | Example Ad-hoc Command |
| --- | --- | --- |
| ping | Checks connectivity | ansible all -m ping |
| command | Runs shell commands (no variables, pipes, or redirection) | ansible web\_servers -m command -a "uptime" |
| shell | Runs shell commands (allows pipes, redirection, variables) | ansible db\_servers -m shell -a "df -h /" |
| copy | Copies files to remote hosts | ansible all -m copy -a "src=/etc/hosts dest=/tmp/hosts" |
| fetch | Fetches files from remote hosts | ansible all -m fetch -a "src=/var/log/syslog dest=./logs/" |
| file | Manages file permissions, directories, and symlinks | ansible all -m file -a "path=/opt/data state=directory mode=0755" |
| yum / apt | Installs or removes packages (YUM for RHEL, APT for Ubuntu) | ansible all -m yum -a "name=httpd state=present" |
| service | Manages system services | ansible all -m service -a "name=nginx state=restarted" |
| lineinfile | Modifies lines in a file | ansible all -m lineinfile -a "path=/etc/ssh/sshd\_config line='PermitRootLogin no' state=present" |

* Running ad-hoc commands (ansible all -m ping, ansible webservers -m command -a "uptime")

📌 **1. Ping All Hosts to Check Connectivity**

ansible all -m ping

📌 **2. Check Uptime of Web Servers**

ansible web\_servers -m command -a "uptime"

📌 **3. Run a Shell Command on All Hosts**

ansible all -m shell -a "df -h /"

📌 **4. Copy a File to Remote Hosts**

ansible all -m copy -a "src=/tmp/config.conf dest=/etc/config.conf owner=root mode=0644"

📌 **5. Fetch a File from Remote Hosts to Local Machine**

ansible all -m fetch -a "src=/var/log/messages dest=./logs/messages flat=yes"

📌 **6. Create a Directory on All Hosts**

ansible all -m file -a "path=/opt/data state=directory mode=0755"

📌 **7. Install Nginx using YUM on RHEL-based Servers**

ansible all -m yum -a "name=nginx state=present"

📌 **8. Start & Enable a Service (Nginx)**

ansible all -m service -a "name=nginx state=started enabled=yes"

📌 **9. Modify SSH Config to Disable Root Login**

ansible all -m lineinfile -a "path=/etc/ssh/sshd\_config regexp='^PermitRootLogin' line='PermitRootLogin no'"

📌 **10. Restart SSH Service After Modifying Config**

ansible all -m service -a "name=sshd state=restarted"

## **🔹 Summary**

✔ **Use ad-hoc commands** for quick tasks **without writing a Playbook**.  
✔ **Test connectivity** using ansible **all -m ping**.  
✔ **Modify configurations** with **lineinfile**.

**4) Playbooks**

An Ansible Playbook consists of:

🔹 **Hosts** – Define target machines to run tasks on  
🔹 **Tasks** – The actions to be performed (e.g., install a package, restart a service)  
🔹 **Vars** – Variables for dynamic values  
🔹 **Handlers** – Triggered when a change occurs (e.g., restart a service after an update)  
🔹 **Roles** – Modularized collections of tasks, handlers, templates, and variables

📌 **Basic Playbook Structure:**

---  
- name: Configure Web Servers  
 hosts: web\_servers  
 become: yes # Run tasks as root  
 vars:  
 http\_port: 80  
 document\_root: /var/www/html  
  
 tasks:  
 - name: Install Apache  
 yum:  
 name: httpd  
 state: present  
  
 - name: Start Apache service  
 service:  
 name: httpd  
 state: started  
 enabled: yes

## **2️⃣ Writing Simple & Complex Playbooks ✍️**

### **✅ Simple Playbook Example: Installing Nginx**

---  
- name: Install and Start Nginx  
 hosts: web\_servers  
 become: yes  
 tasks:  
 - name: Install Nginx  
 yum:  
 name: nginx  
 state: present  
  
 - name: Start and Enable Nginx  
 service:  
 name: nginx  
 state: started  
 enabled: yes

### **✅ Complex Playbook Example: Install & Configure Apache**

---  
- name: Install and Configure Apache  
 hosts: web\_servers  
 become: yes  
  
 vars:  
 http\_port: 8080  
  
 tasks:  
 - name: Install Apache  
 yum:  
 name: httpd  
 state: present  
  
 - name: Start and Enable Apache  
 service:  
 name: httpd  
 state: started  
 enabled: yes  
  
 - name: Change Apache Port  
 lineinfile:  
 path: /etc/httpd/conf/httpd.conf  
 regexp: "^Listen "  
 line: "Listen {{ http\_port }}"  
 notify: Restart Apache  
  
 handlers:  
 - name: Restart Apache  
 service:  
 name: httpd  
 state: restarted

## **3️⃣ Conditionals (when:) 🧐**

You can use when: to execute tasks **conditionally** based on variables or system facts.

📌 **Example: Install Apache only on RHEL-based systems**

---  
- name: Install Apache on RHEL  
 hosts: web\_servers  
 become: yes  
 tasks:  
 - name: Install Apache  
 yum:  
 name: httpd  
 state: present  
 when: ansible\_os\_family == "RedHat"

📌 **Example: Create a User Only if It Does Not Exist**

---  
- name: Add User Only If It Doesn't Exist  
 hosts: all  
 become: yes  
 tasks:  
 - name: Create a User  
 user:  
 name: devuser  
 state: present  
 when: "'devuser' not in ansible\_facts['getent\_passwd']"

## **4️⃣ Loops (loop, with\_items) 🔄**

Loops allow you to run a task **multiple times** with different inputs.

📌 **Example: Install Multiple Packages Using loop**

---  
- name: Install Multiple Packages  
 hosts: all  
 become: yes  
 tasks:  
 - name: Install packages  
 yum:  
 name: "{{ item }}"  
 state: present  
 loop:  
 - httpd  
 - vim  
 - git

📌 **Example: Create Multiple Users Using with\_items**

yaml

---  
- name: Create Users  
 hosts: all  
 become: yes  
 tasks:  
 - name: Add Users  
 user:  
 name: "{{ item }}"  
 state: present  
 with\_items:  
 - user1  
 - user2  
 - user3

## **5️⃣ Handlers (notify, listen) 🔁**

Handlers **run only when notified** by a task.

📌 **Example: Restart Apache Only If Configuration Changes**

---  
- name: Configure Apache  
 hosts: web\_servers  
 become: yes  
 tasks:  
 - name: Change Apache Config  
 lineinfile:  
 path: /etc/httpd/conf/httpd.conf  
 regexp: "^Listen "  
 line: "Listen 8080"  
 notify: Restart Apache  
  
 handlers:  
 - name: Restart Apache  
 service:  
 name: httpd  
 state: restarted

📌 **Example: Using listen to Group Handlers**

---  
- name: Configure Web Server  
 hosts: web\_servers  
 become: yes  
 tasks:  
 - name: Update Apache Config  
 lineinfile:  
 path: /etc/httpd/conf/httpd.conf  
 regexp: "^Listen "  
 line: "Listen 8080"  
 notify: Restart Web Server  
  
 - name: Update Firewall Rules  
 lineinfile:  
 path: /etc/firewalld/services/http.xml  
 line: '<port protocol="tcp" port="8080"/>'  
 notify: Restart Web Server  
  
 handlers:  
 - name: Restart Apache  
 listen: Restart Web Server  
 service:  
 name: httpd  
 state: restarted

**5) Variables and Templates**

* Defining variables (vars, vars\_files, host\_vars, group\_vars)

Ansible provides multiple ways to define and manage variables for better **scalability and reusability**.

## **1️⃣ Ways to Define Variables in Ansible**

### **✅ 1. Inline Variables (vars) in Playbook**

You can define variables inside a playbook under the vars section.

📌 **Example: Using Inline Variables**

---  
- name: Deploy Web Server  
 hosts: web\_servers  
 become: yes  
 vars:  
 http\_port: 8080  
 document\_root: /var/www/html  
  
 tasks:  
 - name: Configure Apache Port  
 lineinfile:  
 path: /etc/httpd/conf/httpd.conf  
 regexp: "^Listen "  
 line: "Listen {{ http\_port }}"

### **✅ 2. Using External Variable Files (vars\_files)**

Instead of defining variables directly in the playbook, you can store them in a separate file and reference them.

📌 **Step 1: Create a Variables File (webserver\_vars.yml)**

http\_port: 8080  
document\_root: /var/www/html

📌 **Step 2: Reference the File in the Playbook**

---  
- name: Deploy Web Server  
 hosts: web\_servers  
 become: yes  
 vars\_files:  
 - webserver\_vars.yml

**✅ 3. Using Host-Specific Variables (host\_vars/)**

Host-specific variables allow **per-host** customization.

📌 **Step 1: Create a host\_vars/ directory**

host\_vars/  
 ├── server1.yml  
 ├── server2.yml

📌 **Step 2: Define Variables in host\_vars/server1.yml**

yaml

http\_port: 8081  
server\_name: "server1.example.com"

📌 **Step 3: Use It in the Playbook**

yaml

---  
- name: Configure Web Server  
 hosts: web\_servers  
 tasks:  
 - name: Print Server Name  
 debug:  
 msg: "Deploying on {{ server\_name }}"

### **✅ 4. Using Group-Specific Variables (group\_vars/)**

Group variables apply **to all hosts** in a specific group.

📌 **Step 1: Create a group\_vars/ directory**

group\_vars/  
 ├── web\_servers.yml  
 ├── db\_servers.yml

📌 **Step 2: Define Variables in group\_vars/web\_servers.yml**

yaml

http\_port: 8080  
web\_user: apache

📌 **Step 3: Use It in the Playbook**

yaml

---  
- name: Configure Web Server  
 hosts: web\_servers  
 tasks:  
 - name: Print Web User  
 debug:  
 msg: "Running web server as {{ web\_user }}"

* **Jinja2 templates ({{ variable\_name }}, {{ ansible\_facts['hostname'] }})**

## **2️⃣ Jinja2 Templates in Ansible 🏗️**

Jinja2 is a **templating engine** used in Ansible to dynamically generate configuration files.

### **✅ 1. Using Jinja2 Variables in Playbooks**

Jinja2 allows **embedding variables inside templates**.

📌 **Example: Print System Hostname and IP Address**

yaml

---  
- name: Print System Info  
 hosts: all  
 tasks:  
 - name: Show Hostname and IP  
 debug:  
 msg: "Hostname: {{ ansible\_facts['hostname'] }}, IP: {{ ansible\_facts['default\_ipv4']['address'] }}"

### **✅ 2. Creating a Jinja2 Template**

Templates help **generate dynamic configuration files**.

📌 **Step 1: Create a Jinja2 Template (httpd.conf.j2)**

jinja

Listen {{ http\_port }}  
  
<VirtualHost \*:{{ http\_port }}>  
 DocumentRoot {{ document\_root }}  
 ServerName {{ server\_name }}  
</VirtualHost>

📌 **Step 2: Use It in an Ansible Playbook**

yaml

---  
- name: Deploy Web Server  
 hosts: web\_servers  
 become: yes  
 vars:  
 http\_port: 8080  
 document\_root: /var/www/html  
 server\_name: mywebsite.com  
  
 tasks:  
 - name: Deploy Apache Config  
 template:  
 src: httpd.conf.j2  
 dest: /etc/httpd/conf/httpd.conf  
 notify: Restart Apache  
  
 handlers:  
 - name: Restart Apache  
 service:  
 name: httpd  
 state: restarted

### **✅ 3. Using Conditionals in Jinja2**

You can use **if-else conditions** in templates.

📌 **Example: Set a Default Document Root If Not Defined**

jinja

DocumentRoot {% if document\_root is defined %}{{ document\_root }}{% else %}/var/www/html{% endif %}

**6) Roles & Best Practices**

* Creating roles (ansible-galaxy init myrole)

Ansible **roles** help organize playbooks into **modular**, **reusable**, and **structured** components. Roles are useful for managing **large-scale automation** by separating tasks, handlers, variables, and templates.

## **1️⃣ Creating a Role Using Ansible Galaxy 🚀**

Ansible provides the ansible-galaxy init command to generate a **standard directory structure** for a role.

📌 **Command to Create a Role:**

ansible-galaxy init myrole

📌 **Example: Creating a Role for Apache Web Server**

ansible-galaxy init apache

* Directory structure (tasks, handlers, defaults, vars, files, templates)

## **2️⃣ Directory Structure of an Ansible Role 📂**

When you run ansible-galaxy init apache, it creates the following structure:

apache/  
├── defaults/ # Default variables  
│ ├── main.yml  
├── files/ # Static files (e.g., config files, scripts)  
├── handlers/ # Handlers (e.g., restart services)  
│ ├── main.yml  
├── meta/ # Metadata about the role  
│ ├── main.yml  
├── tasks/ # Main tasks (actual automation logic)  
│ ├── main.yml  
├── templates/ # Jinja2 templates for dynamic configs  
├── tests/ # Test playbooks for role validation  
│ ├── test.yml  
├── vars/ # Role-specific variables  
│ ├── main.yml

## **3️⃣ Explanation of Each Directory 📁**

📌 **1. tasks/ (Main Automation Logic)**  
This is where you define the **actual tasks** that the role performs.

📌 **Example: tasks/main.yml (Install & Start Apache)**

---  
- name: Install Apache  
 yum:  
 name: httpd  
 state: present  
  
- name: Start Apache Service  
 service:  
 name: httpd  
 state: started  
 enabled: yes

📌 **2. handlers/ (Triggered on Changes)** Handlers are like tasks but run **only when notified**.

📌 **Example: handlers/main.yml (Restart Apache)**

---  
- name: Restart Apache  
 service:  
 name: httpd  
 state: restarted

📌 **3. defaults/ (Default Variables for Role)**  
These variables have the **lowest precedence** and can be overridden.

📌 **Example: defaults/main.yml**

---  
http\_port: 80  
document\_root: /var/www/html

📌 **4. vars/ (Role-Specific Variables with Higher Priority)**  
Variables defined here **override** those in defaults/.

📌 **Example: vars/main.yml**

---  
http\_port: 8080

📌 **5. files/ (Static Files to Copy to Managed Nodes)**  
Used for **static files** like configuration files or custom scripts.

📌 **Example: Copy a Static HTML File**

---  
- name: Copy index.html  
 copy:  
 src: index.html  
 dest: /var/www/html/index.html

📌 **6. templates/ (Dynamic Config Files Using Jinja2)**  
Used for **templated configuration files**.

📌 **Example: templates/httpd.conf.j2**

Listen {{ http\_port }}  
  
<VirtualHost \*:{{ http\_port }}>  
 DocumentRoot {{ document\_root }}  
</VirtualHost>

📌 **Example: Use the Template in a Task (tasks/main.yml)**

---  
- name: Deploy Apache Config  
 template:  
 src: httpd.conf.j2  
 dest: /etc/httpd/conf/httpd.conf  
 notify: Restart Apache

## **4️⃣ Using the Role in a Playbook 🎭**

Once the role is created, you can use it in a playbook.

📌 **Example: site.yml**

---  
- name: Deploy Web Server  
 hosts: web\_servers  
 roles:  
 - apache

📌 **Run the Playbook**

ansible-playbook site.yml

**7) Ansible Vault**

* Encrypting secrets (ansible-vault encrypt myfile.yml)

## **Encrypting Secrets in Ansible with Ansible Vault 🔐**

Ansible Vault helps **secure sensitive data** like passwords, API keys, and SSH credentials by encrypting them.

## **1️⃣ Encrypting a File Using Ansible Vault**

To **encrypt a file**, use:

bash

ansible-vault encrypt myfile.yml

🔹 You will be prompted to enter a password.

📌 **Example: Encrypting a Secrets File**

bash

ansible-vault encrypt secrets.yml

## **2️⃣ Editing an Encrypted File**

To **edit** the encrypted file securely:

bash

ansible-vault edit secrets.yml

## **3️⃣ Decrypting a File**

To **permanently decrypt** the file:

bash

ansible-vault decrypt secrets.yml

If you only want to **view** the file without decrypting it permanently:

bash

ansible-vault view secrets.yml

* Using Vault in playbooks (vars\_files: secrets.yml)

## **4️⃣ Using Vault in Playbooks**

You can **reference an encrypted file** inside your playbook under vars\_files:

📌 **Example: Using an Encrypted Vault File in Playbook**

yaml

---  
- name: Deploy Secure Application  
 hosts: all  
 vars\_files:  
 - secrets.yml # This file is encrypted with Ansible Vault  
 tasks:  
 - name: Print Secret Key  
 debug:  
 msg: "The secret key is {{ secret\_key }}"

## **5️⃣ Running a Playbook with Vault**

When running a playbook with an encrypted file, you must **provide the vault password**.

🔹 **Manually enter the password:**

bash

ansible-playbook playbook.yml --ask-vault-pass

🔹 **Using a vault password file (not recommended for security reasons):**

bash

ansible-playbook playbook.yml --vault-password-file vault-pass.txt

(The file vault-pass.txt should contain only the vault password.)

## **6️⃣ Encrypting Specific Variables in a YAML File**

Instead of encrypting the entire file, you can encrypt only **specific variables**.

📌 **Example: Encrypting a Password Variable in secrets.yml**

db\_password: !vault |  
 $ANSIBLE\_VAULT;1.1;AES256  
 6237626338326538653136626134...

To encrypt a single variable, run:

ansible-vault encrypt\_string 'MySecretPassword' --name 'db\_password'

**8) Error Handling & Debugging**

## **Advanced Ansible Execution & Error Handling 🚀**

### **1️⃣ Dry Run Mode (--check) 🛠️**

The --check flag is used to **simulate** playbook execution **without making actual changes**.

📌 **Example: Running a Playbook in Dry-Run Mode**

ansible-playbook playbook.yml --check

🔹 Useful for testing what would happen **before** making changes.  
🔹 **Limitations:** Some modules may fail because they depend on real execution (e.g., shell, command).

### **2️⃣ Debugging with the debug Module 🧐**

The debug module helps print variables or messages for **troubleshooting**.

📌 **Example: Print a Message**

- name: Print a message  
 debug:  
 msg: "Hello, Ansible!"

📌 **Example: Print a Variable**

- name: Show OS Information  
 debug:  
 var: ansible\_facts['distribution']

📌 **Example: Print JSON Output for Better Formatting**

- name: Show All Variables  
 debug:  
 var: ansible\_facts  
 verbosity: 2

🔹 **Use verbosity: 2 to display debug output only when running with -vv.**

### **3️⃣ Handling Failures with failed\_when ❌**

The failed\_when condition is used to **customize failure conditions**.

📌 **Example: Fail the Task if the HTTP Response Code is Not 200**

- name: Check if Website is Up  
 uri:  
 url: <http://example.com> return\_content: no  
 register: result  
 failed\_when: result.status != 200

🔹 **Prevents false positives** where a command might exit successfully but return an unexpected result.

### **4️⃣ Ignoring Errors with ignore\_errors 🚨**

The ignore\_errors option **allows playbook execution to continue even if a task fails**.

📌 **Example: Ignore Failure of a Command**

- name: Try to Restart a Non-Existent Service  
 service:  
 name: nonexistent\_service  
 state: restarted  
 ignore\_errors: yes

🔹 Useful when **non-critical tasks** should not stop the playbook execution.  
🔹 Combine with failed\_when for **better error handling**.

### **5️⃣ Combining failed\_when and ignore\_errors 🎯**

📌 **Example: Log an Error Without Stopping Execution**

- name: Check Disk Space  
 shell: df -h | grep "/dev/sda1"  
 register: disk\_space  
 failed\_when: "'100%' in disk\_space.stdout"  
 ignore\_errors: yes

🔹 This **logs an error** if the disk is full but **does not stop execution**.