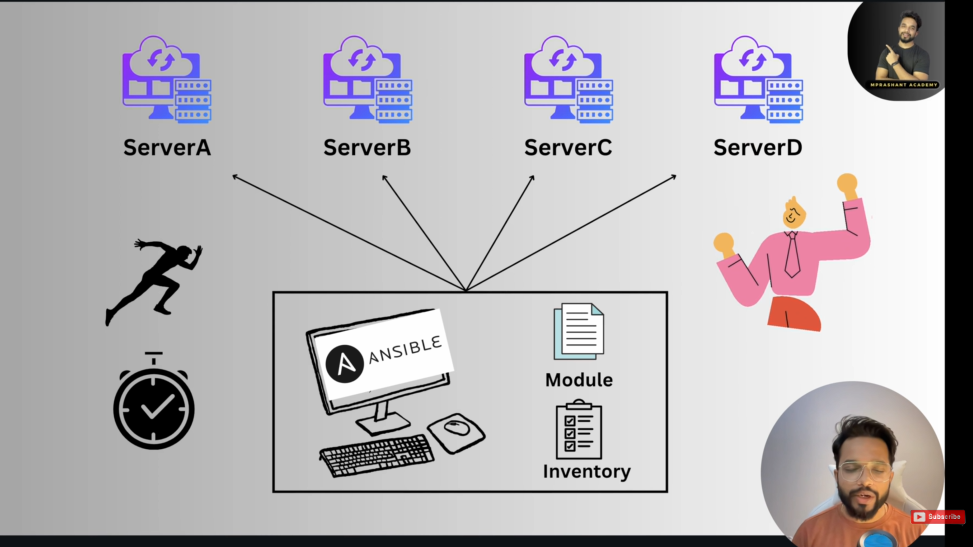
Ansible (Udemy)



**What is Ansible?**

Ansible is an IT automation tool that helps you automate tasks on your computers and servers. Think of it like a remote control for your IT infrastructure.

**Key Features**

1. **No Need for Extra Software**: [Ansible operates without the need for agents on managed nodes](https://www.ansible.com/how-ansible-works/). You don’t need to install any special software on the computers you want to control. Ansible uses SSH (a secure way to connect to other computers) to do its job.
2. **Easy to Read and Write**: Ansible uses YAML, a simple language that looks like plain English, to write instructions. This makes it easy for anyone to understand and use.
3. **Works Everywhere**: Ansible can manage different types of systems, like Linux, Windows, cloud services, and even network devices.
4. **Handles Big and Small Setups**: Whether you have a few computers or thousands, Ansible can manage them all.
5. **Consistent Results**: Ansible ensures that running the same instructions multiple times won’t change the system after the first successful run.
6. **Open Source**: It is free and has a large community.
7. **Integration**: It can be integrated with other tools such as docker, Kubernetes, Azure, AWS

**How Ansible Works**

1. **Control Node**: This is your main computer where you run Ansible. You write your instructions (playbooks) here.
2. **Managed Nodes**: These are the computers or servers you want to control. Ansible connects to them and runs the tasks you define.
3. **Modules** Smallest Program to do a task, like installing software or creating files.
4. **Playbooks**: Playbooks are like recipes. They contain a list of tasks you want to perform on your managed nodes. Like creating a web server on the remote system
5. **Inventory**: This is a list of all the computers or servers you want to manage. It can be a simple text file with their names or IP addresses.

**Example**: Let’s say you want to install a web server on multiple computers. Here’s how you can do it with Ansible:

1. **Write a Playbook**:

- name: Install web server

hosts: webservers

tasks:

- name: Install nginx

apt:

name: nginx

state: present

This playbook tells Ansible to install nginx on all computers listed under “webservers”.

1. **Create an Inventory File**:

[webservers]

server1.example.com

server2.example.com

This file lists the computers where you want to install the web server.

1. **Run the Playbook**: On your control node, you run the command:

ansible-playbook -i inventory playbook.yml

Ansible will connect to the computers listed in the inventory and install Apache on them.

**YAML (Udemy)**

YAML, which stands for “Yet Another Markup Language,” is a human-readable data serialization format often used for configuration files and data exchange between different programming languages.

YAML is designed to be easy to read and write, making it a popular choice for configuration files and data serialization. It is similar to JSON and XML but aims to be more human-friendly.

**Key Features of YAML**

1. **Human-Readable**: YAML’s syntax is designed to be easy for humans to read and write. It uses indentation to represent structure, similar to Python.
2. **Data Serialization**: YAML is used to serialize data, which means converting data structures into a format that can be easily stored and transmitted.
3. **Support for Complex Data Types**: YAML supports various data types, including scalars (strings, numbers), lists, and dictionaries (key-value pairs).
4. **Compatibility with JSON**: [YAML is a superset of JSON, meaning any valid JSON file is also a valid YAML file1](https://www.freecodecamp.org/news/what-is-yaml-the-yml-file-format/).

**Basic Syntax**: YAML files typically have a .yaml or .yml extension. Here are some basic syntax rules:

1. **Start of a Document**: YAML documents start with --- and end with ... (optional).

---

1. **Key-Value Pairs**: Key-value pairs are written as key: value.

name: John Doe

age: 30

1. **Lists**: Lists are represented with a dash (-) followed by a space.

hobbies:

- reading

- hiking

- coding

1. **Dictionaries**: Dictionaries (or maps) are collections of key-value pairs.

address:

street: 123 Main St

city: Anytown

state: CA

1. **Comments**: Comments start with # and are ignored by the parser.

# This is a comment

**Example YAML File:** Here’s an example of a YAML file that might be used for configuring a web application:

---

app\_name: MyWebApp

version: 1.0

maintainer: Jane Doe

database:

host: localhost

port: 5432

name: mywebapp\_db

user: dbuser

password: secret

features:

- user\_authentication

- data\_encryption

- api\_support

logging:

level: debug

file: /var/log/mywebapp.log

**Ansible Inventory**

Ansible inventory is like a list of all the computers and servers you want to manage with Ansible. It tells Ansible where these machines are and how to connect to them. The inventory file is where you list all the systems (hosts) you want Ansible to manage. If you don’t create a new inventory file, Ansible uses the default one located at /etc/ansible/hosts

**Types of Inventory**

1. **Static Inventory**: This is a fixed list of machines, written in a file.
   * Example in INI format:

**[webservers]**

web1.example.com

web2.example.com

**[dbservers]**

db1.example.com

db2.example.com

**Groups**: You can group servers together using square brackets. For example, [webservers] and [dbservers] are groups.

* + Example in YAML format:

all:

hosts:

web1.example.com:

web2.example.com:

children:

dbservers:

hosts:

db1.example.com:

db2.example.com:

1. **Dynamic Inventory**: This is generated automatically by scripts or plugins, useful for environments where machines come and go frequently, like in the cloud.
   * Example: Using a script to list all running AWS EC2 instances.

**Simple Example:** Let’s say you have three servers: two web servers and one database server. You want to manage these servers using Ansible.

**Inventory File in INI Format**

**[webservers]**

web1.example.com

web2.example.com

**[dbservers]**

db1.example.com

* **Groups**: [webservers] and [dbservers] are groups. You can group servers together to manage them more easily.
* **Hosts**: web1.example.com, web2.example.com, and db1.example.com are the servers you want to manage.

**Adding Aliases and Parameters**: You can also give servers aliases (nicknames) and specify additional parameters like the IP address, user, and port.

**[webservers]**

web1 ansible\_host=192.168.1.10 ansible\_user=admin ansible\_port=2222

web2 ansible\_host=192.168.1.11 ansible\_user=admin ansible\_port=2222

**[dbservers]**

db1 ansible\_host=192.168.1.20 ansible\_user=admin ansible\_port=2222

* **Aliases**: web1 and web2 are aliases for the servers.
* **ansible\_host**: Specifies the actual IP address of the server.
* **ansible\_user**: Specifies the user to connect as.
* **ansible\_port**: Specifies the port to connect to (default is 22 for SSH).
* **Ansible\_ssh\_pass**: password for linux Ssh
* **Ansible\_password**: for Windows password

**Inventory File in YAML Format**

Here’s the same example in YAML format:

all:

children:

webservers:

hosts:

web1:

ansible\_host: 192.168.1.10

ansible\_user: admin

ansible\_port: 2222

web2:

ansible\_host: 192.168.1.11

ansible\_user: admin

ansible\_port: 2222

dbservers:

hosts:

db1:

ansible\_host: 192.168.1.20

ansible\_user: admin

ansible\_port: 2222

**Variables**

­**Ansible Variables: Variables** in Ansible are like placeholders that store values. You can use them to make your playbooks more flexible and reusable.

* **Types of Variables**

1. **Simple Variables**: These are just single values.

greeting: "Hello, World!"

1. **List Variables**: These are lists of items.

fruits:

- apple

- banana

- cherry

1. **Dictionary Variables**: These are like a list of key-value pairs.

person:

name: John

age: 30

* **Registering Variables:** You can **register** the result of a task as a variable. This means you can use the output of a command later in your playbook.

- name: Get the current date

command: date

register: current\_date

- debug:

msg: "The current date is {{ current\_date.stdout }}"

In this example, the date command’s output is stored in current\_date, and then we print it.

* **Variable Precedence:** When you have multiple variables with the same name, Ansible decides which one to use based on a specific order. Think of it as a hierarchy:

1. Default values in roles
2. Variables in inventory files
3. Variables in playbooks
4. Variables set during runtime (extra vars)

The last one in the list has the highest priority.

* **Variable Scoping:** Variables can be defined in different places, and their scope determines where they can be used:
* **Global Scope**: Available everywhere.
* **Play Scope**: Available within a playbook.
* **Host Scope**: Available for a specific host.
* **Role Scope**: Available within a role.
* **Magic Variables: Magic variables** are special variables that Ansible provides automatically. They give you information about the environment and the playbook execution.
* hostvars: Access variables of other hosts.
* group\_names: List of groups the current host belongs to.
* inventory\_hostname: The name of the current host.
* ansible\_playbook\_dir: The directory where the playbook is located.
* **Ansible Facts: Ansible facts** are pieces of information that Ansible gathers about the managed hosts. These facts include details like the operating system, IP addresses, and more.

- name: Gather facts

ansible.builtin.setup:

- debug:

msg: "The OS type is {{ ansible\_facts['os\_family'] }}"

In this example, Ansible collects facts about the host, and then we print the operating system type.

**Playbook**

Think of Ansible playbooks as a set of instructions you give to a computer to perform tasks. These instructions are written in a simple, human-readable format called YAML. Here’s a breakdown:

* **Structure**: A playbook is like a recipe book. Each “recipe” (called a play) tells which computers (hosts) to work on and what steps (tasks) to perform.
* **Tasks**: Each task is a specific action, like installing software or copying a file.
* **Idempotency**: This means if you run the playbook multiple times, it won’t make unnecessary changes if everything is already set up correctly.

**Example**:

---

- name: Update web servers

hosts: webservers

tasks:

- name: Ensure apache is at the latest version

ansible.builtin.yum:

name: httpd

state: latest

- name: Write the apache config file

ansible.builtin.template:

src: /srv/httpd.j2

dest: /etc/httpd.conf

In this example:

* The playbook updates web servers.
* It ensures Apache (a web server software) is up-to-date.
* It copies a configuration file to the correct location.

**Verifying Playbooks**

Before running a playbook, it’s good to check it for errors and see what it will do. Here are some ways to verify:

* **Syntax Check**: This checks if the playbook is written correctly.

ansible-playbook --syntax-check my\_playbook.yml

* **Check Mode**: This simulates running the playbook without making any changes.

ansible-playbook --check my\_playbook.yml

* **Diff Mode**: This shows the differences between the current state and what the playbook will change.

ansible-playbook --check --diff my\_playbook.yml

* **List Hosts**: This shows which computers the playbook will affect.

ansible-playbook --list-hosts my\_playbook.yml

* **List Tasks**: This shows the tasks that will be performed.

ansible-playbook --list-tasks my\_playbook.yml

**Ansible-Init:** ansible-init is a tool that helps you start a new Ansible project quickly. It sets up a standard directory structure so you can organize your files easily.

* **Usage**: Run ansible-init followed by your project name.

ansible-init my\_project

This command creates a directory named my\_project with folders for roles, inventories, playbooks, etc.

**Ansible conditionals, Ansible Conditionals based on facts, variables, re-use and Loops**

* **Ansible Conditionals:** Conditionals let you run tasks only if certain conditions are true. Think of it like saying, “Do this, but only if that is true.”

**Example**:

- name: Install Apache on Debian

apt:

name: apache2

state: present

when: ansible\_facts['os\_family'] == "Debian"

This task installs Apache only if the operating system is Debian.

* **Conditionals Based on Facts:** Facts are details about the system, like its OS type or IP address. You can use these facts to decide whether to run a task.

**Example**:

- name: Check SELinux status

ansible.builtin.command: getenforce

register: selinux\_status

- name: Enable SELinux

ansible.builtin.command: setenforce 1

when: selinux\_status.stdout != "Enforcing"

Here, the second task enables SELinux only if it’s not already enabled.

* **Conditionals Based on Variables:** You can also use variables to control whether a task runs.

**Example**:

- name: Install a package

yum:

name: "{{ package\_name }}"

state: present

when: package\_name is defined

This task installs a package only if the package\_name variable is set.

* **Re-use with Conditionals:** You can reuse tasks by including them based on conditions. This helps avoid repeating the same tasks in different places.

**Example**:

- name: Include Debian tasks

include\_tasks: debian.yml

when: ansible\_facts['os\_family'] == "Debian"

- name: Include RedHat tasks

include\_tasks: redhat.yml

when: ansible\_facts['os\_family'] == "RedHat"

This includes different sets of tasks depending on whether the system is Debian or RedHat.

* **Loops in Ansible:** Loops let you run the same task multiple times with different items.

**Example**:

- name: Install multiple packages

apt:

name: "{{ item }}"

state: present

loop:

- apache2

- mysql-server

- php

This task installs Apache, MySQL, and PHP by looping through the list of packages.

* **Combining Loops and Conditionals:** You can combine loops and conditionals to create more complex logic.

**Example**:

- name: Install packages on Debian

apt:

name: "{{ item }}"

state: present

loop:

- apache2

- mysql-server

- php

when: ansible\_facts['os\_family'] == "Debian"

This task installs the packages only if the system is Debian.

* **With directive:** These directives are used to create loops in playbooks, allowing you to iterate over a list of items and perform tasks on each item. Here are some common with\_ directives and how they work:
* **with\_file:** This directive iterates over a list of file paths.

Example:

- name: Read multiple files

command: cat "{{ item }}"

with\_file:

- /path/to/file1

- /path/to/file2

- /path/to/file3

Here, the command task reads the contents of each file in the list.

* **with\_items**: This is the most basic and commonly used directive. It iterates over a list of items.
* **With\_url**: This directive iterates over a list of URLs.
* **With\_dict**: This directive iterates over the key-value pairs of a dictionary.
* **With\_together**: This directive iterates over multiple lists in parallel.
* **With\_nested**: This directive iterates over all combinations of items from multiple lists.
* **With\_sequence**: This directive generates a sequence of numbers.
* **With\_lines**: This directive iterates over the lines of a command output.

**Ansible modules, ansible plugins**

**Ansible Modules (Study form chatgpt)**

Ansible modules are the building blocks of Ansible’s functionality. They are small programs that perform specific tasks on remote hosts. Modules can manage system resources, execute commands, and handle various configurations.

**Types of Modules**:

* **System Modules:** These modules manage system resources and configurations.

**User Module:** Manages user accounts

**Group Module:** Manages user groups

**File Module:** Manages files and directories

* **Package Management Modules:** These modules manage software packages on different operating systems.

**Yum Module**: Manages packages on Red Hat-based systems

**Apt Module**: Manages packages on Debian-based systems

* **Service Management Modules:** These modules manage services on remote hosts.
* **Command and Shell Modules:** These modules execute commands on remote hosts.
* **Networking Modules:** These modules manage network configurations and devices.

**Ios\_config Module:** Manages Cisco IOS configurations.

**Netconf Module:** Manages network devices using NETCONF

* **Cloud Modules:** These modules manage cloud resources.

**Ec2 Module:** Manages AWS EC2 instances

**Azure\_rm Module:** Manages Azure resources

* **Database Modules:** These modules manage databases and database users.

**Mysql\_user Module**: Manages MySQL users

**Postgresql\_db Module:** Manages PostgreSQL databases

* **Files and Templates Modules:** These modules manage files and templates on remote hosts.

**Copy Module**: Copies files to remote hosts.

**Template Module:** Manages Jinja2 templates

**Ansible Plugins (Study from chatgpt)**

Plugins are like add-ons that extend Ansible’s capabilities. They help Ansible do more things or do things differently.

**Types of Plugins**:

* **Action Plugins**: Change how modules behave.
* **Callback Plugins**: Customize the output you see when Ansible runs.
* **Connection Plugins**: Manage how Ansible connects to remote computers.
* **Lookup Plugins**: Fetch data from external sources.
* **Filter Plugins**: Modify data within templates.
* **Inventory Plugins**: Handle dynamic inventories of hosts.
* **Vars Plugins**: Provide variables from external sources.

**Example of a Callback Plugin**: Imagine you want to print a message every time a playbook starts. You can write a callback plugin to do that.

**Ansible handlers, ansible roles and ansible collections**

* **Ansible Handlers (tasks triggered by notify)**

**Handlers** are special tasks in Ansible that are triggered by other tasks using the notify directive. They are typically used to restart services or perform other actions that should only happen once, even if multiple tasks notify them. Handlers run at the end of a playbook execution, ensuring that the necessary changes are applied only after all tasks have been executed.

**Example**: Imagine you manage several web servers. You often change the web server’s configuration file to adjust settings. However, just changing the file doesn’t apply the changes immediately. You need to restart the web server for the changes to take effect. Doing this manually every time can be tedious and error-prone.

**Handlers** in Ansible help automate this. You can define a handler to restart the web server and link it to the task that modifies the configuration file. This way, whenever the configuration file is changed, the handler is triggered to restart the web server automatically. Example: installing Apache

- name: Install Apache

apt:

name: apache2

state: present

notify: Restart Apache

handlers:

- name: Restart Apache

service:

name: apache2

state: restarted

* **Ansible Roles: (Chatgpt)**

**Roles** are a way to organize Ansible playbooks into reusable components. They help you keep everything neat and reusable. Think of a role as a package that contains everything needed to set up a part of your system, like installing and configuring a web server.

A role might look like this:

roles/

└── webserver/

├── tasks/

│ └── main.yml

├── handlers/

│ └── main.yml

├── templates/

├── files/

├── vars/

│ └── main.yml

├── defaults/

│ └── main.yml

├── meta/

│ └── main.yml

* **tasks/main.yml**: Contains the tasks to set up the web server.
* **handlers/main.yml**: Defines Handlers (tasks triggered by notify)
* **templates/**: Contains template files that can be customized.
* **files/**: Contains files that need to be copied to the server.
* **vars/main.yml**: Contains variables specific to the role.
* **defaults/main.yml**: Contains default values for variables.
* **meta/main.yml**: Contains metadata about the role, like dependencies on other roles.

Example:

# site.yml

- hosts: webservers

roles:

- webserver

Here, the webserver role contains all the tasks and files needed to set up web servers.

* **Ansible Galaxy (Chatgpt)**

**Ansible Galaxy** is a hub for finding, sharing, and downloading Ansible roles. It is a community-driven platform where users can publish their roles and use roles created by others

**Using Ansible Galaxy:** You can install roles from Ansible Galaxy using the ansible-galaxy command.

**Installing a Role**

To install a role, use this command:

ansible-galaxy install username.role\_name

For example, to install a role for setting up Nginx by a user named geerlingguy, you would run:

ansible-galaxy install geerlingguy.nginx

**Using Installed Roles**

Once installed, you can use the role in your playbook:

- hosts: webservers

roles:

- geerlingguy.nginx

**Creating and Publishing Roles:** You can also create your own roles and share them on Ansible Galaxy.

1. **Create a Role**: Use the ansible-galaxy init command to create a new role.
2. ansible-galaxy init my\_role
3. **Develop the Role**: Add tasks, handlers, templates, files, and variables to your role.
4. **Publish the Role**: Use the ansible-galaxy command to publish your role.
5. ansible-galaxy import username my\_role

* **Ansible Collections: Collections** are like big bundles that can include roles, modules, and plugins. They make it easy to share and reuse Ansible content. Think of a collection as a library that you can use in your projects.

A collection might look like this:

collections/

└── ansible\_collections/

└── my\_namespace/

└── my\_collection/

├── roles/

├── plugins/

├── modules/

├── playbooks/

├── docs/

├── tests/

├── meta/

└── README.md

**Using Collections:** Imagine you need to manage network devices from different vendors like Cisco, Juniper, and Arista. You can use collections to access specialized modules and roles for each vendor

**Templating in ansible and Jinja2 Templates for Dynamic Configs**

Templating in Ansible allows you to create files that can change based on variables and conditions. This is useful when you need to generate configuration files that are different for each server or environment.

**What is Jinja2?**

Jinja2 is a templating engine used by Ansible to create these dynamic files. It allows you to insert variables, use loops, and apply conditions in your templates.

Basic Concepts

**Variables**: Placeholders that get replaced with actual values.

**Loops**: Repeat a section of the template for each item in a list.

**Conditions**: Include or exclude parts of the template based on certain conditions.

**Filters**: Modify the values of variables.

Example 1: Using Variables

Let’s say you want to create a configuration file for a web server, and you want to include the server’s name.

**Template File (nginx.conf.j2):**

server {

listen 80;

server\_name {{ server\_name }};

location / {

proxy\_pass http://localhost:8080;

}

}

**Playbook (site.yml):**

- hosts: webservers

vars:

server\_name: example.com

tasks:

- name: Create Nginx configuration file

template:

src: nginx.conf.j2

dest: /etc/nginx/sites-available/default

In this example, {{ server\_name }} is a variable that gets replaced with example.com

**Ansible Course 2: Udemy**

**Setting Up a Basic Web Application**

**Overview:** You’re going to set up a simple web application using Python’s Flask framework and a MySQL database. This setup will help you understand the basics of web applications, which will be useful for deploying servers and systems later in the course.

**Steps to Set Up the Application**

1. **Identify the Server**: Choose a server where you will install and run your web application. This could be a virtual machine or a physical server.
2. **Install Python and Dependencies**
   * Since the web application is written in Python, you need to install Python and some additional packages.
   * Use the command: sudo apt-get install python3 python3-pip
3. **Install MySQL Database**
   * Install the MySQL server and client packages to manage your database.
   * Use the command: sudo apt-get install mysql-server mysql-client
   * During installation, set a root password for MySQL. This password will be used later to connect your application to the database.
4. **Configure and Start MySQL**
   * Start the MySQL service using: sudo service mysql start
   * Verify that MySQL is running with: sudo service mysql status
   * Connect to the MySQL database: mysql -u root -p
   * Create a new database: CREATE DATABASE employee\_db;
   * Create a new user and grant permissions:
   * CREATE USER 'youruser'@'localhost' IDENTIFIED BY 'yourpassword';
   * GRANT ALL PRIVILEGES ON employee\_db.\* TO 'youruser'@'localhost';
   * FLUSH PRIVILEGES;
5. **Install Flask**
   * Flask is a lightweight web framework for Python.
   * Install Flask using pip: pip3 install flask
6. **Download or Write the Source Code**
   * If you have the source code on GitHub, clone it to your server:
   * git clone https://github.com/yourusername/yourrepository.git
   * cd yourrepository
   * If you don’t have the source code, you can write a simple Flask application. Here’s an example:
   * from flask import Flask
   * app = Flask(\_\_name\_\_)
   * @app.route('/')
   * def hello():
   * return "Hello, World!"
   * if \_\_name\_\_ == '\_\_main\_\_':
   * app.run(host='0.0.0.0')
7. **Run the Web Application**
   * Start your Flask application:
   * python3 app.py
   * Your web application should now be running and accessible via the server’s IP address.

**(See the udemy video of developing playbook for simple web application and practice the codes)**

**(See the udemy video of file seperation and practice the codes)**

**Asynchronous Actions**

Normally, Ansible waits for each task to finish before moving on to the next one. This is called synchronous execution. But sometimes, tasks take a long time to complete, like installing software or transferring large files. Asynchronous actions let these tasks run in the background so Ansible can continue with other tasks without waiting.

**Why Use Asynchronous Actions?**

* **Avoid Timeouts**: Prevent tasks from failing because they take too long.
* **Improve Efficiency**Keep your playbook running smoothly by not waiting for long tasks.
* **Handle Long Tasks**: Manage tasks that need a lot of time without blocking other tasks.

**How to Use Asynchronous Actions**

You use two special keywords in your playbook:

* async: Sets the maximum time (in seconds) the task can run.
* poll: Tells Ansible how often (in seconds) to check if the task is done.

**Example 1: Basic Asynchronous Task**

Let’s say you have a task that takes 5 minutes (300 seconds) to complete. You want to allow it to run for up to 10 minutes (600 seconds) and check its status every 5 seconds.

- hosts: all

tasks:

- name: Simulate a long-running operation

command: /bin/sleep 300

async: 600

poll: 5

In this example:

* The task runs the sleep command for 300 seconds.
* async: 600 means the task can run for up to 600 seconds.
* poll: 5 means Ansible will check the task’s status every 5 seconds.

**Example 2: Fire and Forget**

If you don’t need to wait for the task to finish, you can set poll to 0. This tells Ansible to start the task and move on immediately.

- hosts: all

tasks:

- name: Run a long-running operation and don't wait

command: /bin/sleep 300

async: 600

poll: 0

In this example, Ansible starts the sleep command and immediately moves on to the next task without waiting for it to finish.

**Strategy in Ansible**

In Ansible, **strategy** refers to the method used to execute tasks across multiple hosts. Different strategies can be chosen based on your needs, and they can significantly impact the performance and efficiency of your playbooks.

**Types of Strategies in Ansible**

1. **Linear Strategy (Default)**
   * **How it works**: Ansible runs each task on all hosts in the play before moving on to the next task.
   * **Example**: If you have three tasks and three hosts, Ansible will run Task 1 on all hosts, then Task 2 on all hosts, and finally Task 3 on all hosts.
   * **Use case**: This is the default and works well for most scenarios.

- hosts: all

tasks:

- name: Task 1

command: echo "Task 1"

- name: Task 2

command: echo "Task 2"

- name: Task 3

command: echo "Task 3"

1. **Free Strategy**
   * **How it works**: Each host runs tasks as fast as it can, independently of other hosts.
   * **Example**: If one host finishes Task 1 quickly, it will immediately start Task 2 without waiting for other hosts.
   * **Use case**: Useful when tasks on different hosts do not depend on each other and can run independently.

- hosts: all

strategy: free

tasks:

- name: Task 1

command: echo "Task 1"

- name: Task 2

command: echo "Task 2"

- name: Task 3

command: echo "Task 3"

**Batch Staergies**

**Error Handling in Ansible**

Error handling in Ansible is crucial for ensuring your playbooks run smoothly and can recover from failures. Here are some key techniques and best practices for handling errors in Ansible:

**1. Ignoring Errors:** Sometimes, you might want to continue executing tasks even if one fails. You can use the ignore\_errors keyword to achieve this.

**2. Handling Unreachable Hosts:** If a host becomes unreachable, Ansible marks it as such and stops running tasks on it. You can use ignore\_unreachable to continue running tasks on other hosts.

**3. Using Blocks and Rescue:** Blocks allow you to group tasks together and handle errors within that block using rescue. This is similar to try-catch blocks in programming.

**4. Always Running Tasks:** You can use the always keyword to ensure certain tasks run regardless of whether previous tasks failed or succeeded.

**5. Defining Failure Conditions:** You can customize what constitutes a failure using the failed\_when keyword. This is useful when a task might return a non-zero exit code that you don’t want to treat as a failure.

**6. Force Handlers:** Handlers are tasks that run when notified by other tasks. By default, if a task fails, handlers won’t run. You can force handlers to run even if there are failures using force\_handlers.

**JINJA2 in Ansible**

**Jinja2** is a tool that helps you create dynamic and flexible configurations in Ansible. It allows you to use variables, filters, loops, and conditionals to make your templates more powerful.

**Key Features of Jinja2**

1. **Variables**: Insert dynamic content into your templates.
2. **Filters**: Modify the output of variables.
3. **Control Structures**: Use loops and conditionals to control the flow of your templates.

**Using Jinja2 in Ansible**

**1. Variables**: You can use variables to insert dynamic content into your templates. Variables are written inside double curly braces {{ }}.

**Example:**

- name: Print a message

hosts: localhost

vars:

my\_name: "Deepak"

tasks:

- name: Show my name

debug:

msg: "My name is {{ my\_name }}"

In this example, {{ my\_name }} will be replaced with “Deepak”.

**2. Filters:** Filters change how the output of a variable looks. You use the pipe symbol | to apply a filter.

**Example:**

- name: Print a message in uppercase

hosts: localhost

vars:

my\_name: "Deepak"

tasks:

- name: Show my name in uppercase

debug:

msg: "My name is {{ my\_name | upper }}"

Here, {{ my\_name | upper }} will convert “Deepak” to “DEEPAK”.

**3. Control Structures:** You can use loops and conditionals to control what happens in your templates.

**Example of a Loop:**

- name: Loop through a list

hosts: localhost

vars:

my\_list:

- item1

- item2

- item3

tasks:

- name: Show each item

debug:

msg: "Item: {{ item }}"

loop: "{{ my\_list }}"

This will print each item in my\_list.

**Example of a Conditional:**

- name: Conditional example

hosts: localhost

vars:

my\_var: true

tasks:

- name: Show a message if my\_var is true

debug:

msg: "my\_var is true"

when: my\_var

This will print “my\_var is true” only if my\_var is true.

**Lookups in Ansible**

**Lookups** in Ansible are a way to get data from outside sources and use it in your playbooks. This data can come from files, environment variables, URLs, and more. Lookups are run on the machine where Ansible is running, not on the remote servers.

**Key Features of Lookups**

1. **Fetching Data**: Get data from different sources.
2. **Integration with Jinja2**: Use lookups inside Jinja2 templates.
3. **Flexibility**: Combine lookups with filters, loops, and conditionals.

**Common Lookup Plugins**

Here are some common lookup plugins you might use:

1. **File Lookup**: Reads the contents of a file.
2. **Env Lookup**: Gets the value of an environment variable.
3. **Password Lookup**: Retrieves passwords from a file.
4. **URL Lookup**: Fetches data from a URL.

**Ansible Vault**

**Ansible Vault** is a feature in Ansible that allows you to keep sensitive data, like passwords and API keys, secure by encrypting them. This means you can store these sensitive details in your playbooks or roles without worrying about them being exposed.

**Key Features of Ansible Vault**

1. **Encrypting Files**: You can encrypt entire files.
2. **Encrypting Variables**: You can encrypt specific variables within your playbooks.
3. **Managing Passwords**: You can manage passwords used for encryption.

**How to Use Ansible Vault**

**1. Encrypting a File**: You can encrypt a file using the ansible-vault encrypt command.

**Example:**

ansible-vault encrypt secrets.yml

This command will prompt you to enter a password and then encrypt the secrets.yml file.

**2. Decrypting a File:** To decrypt a file, use the ansible-vault decrypt command.

**Example:**

ansible-vault decrypt secrets.yml

This command will prompt you for the password and then decrypt the secrets.yml file.

**Dynamic Inventory**

**Dynamic Inventory** in Ansible is a way to automatically generate and manage your list of hosts (servers) from external sources like cloud providers, databases, or other services. Unlike static inventory files, which you have to update manually, dynamic inventory fetches the latest information about your hosts in real-time.

**Why Use Dynamic Inventory?**

* **Automatic Updates**: No need to manually update your inventory file when hosts are added or removed.
* **Scalability**: Easily manage large and changing environments.
* **Integration**: Connects with various data sources like AWS, Google Cloud, databases, etc.

**How to Use Dynamic Inventory**

There are two main ways to set up dynamic inventory in Ansible:

**1. Inventory Scripts**: Inventory scripts are custom scripts that fetch host information from external sources. These scripts can be written in any programming language.

**2. Inventory Plugins:** Inventory plugins are built-in or third-party plugins that fetch host information from various sources. They are more powerful and flexible than scripts.

**Custom Modules**

**Custom modules** in Ansible are scripts that you write to perform specific tasks that are not covered by the built-in modules. They allow you to extend Ansible’s functionality to meet your unique needs.

**Why Use Custom Modules?**

* **Specialized Tasks**: Perform tasks that are specific to your environment or application.
* **Integration**: Interact with systems or APIs that are not supported by existing modules.
* **Customization**: Tailor the behavior of Ansible to fit your requirements.

**How to Create a Custom Module:** Creating a custom module involves writing a script, usually in Python, that Ansible can execute. Here’s a step-by-step guide to creating a simple custom module.

**Step 1: Set Up Your Development Environment:** Create a directory structure for your Ansible project

**Step 2: Write Your Custom Module:** Create a new Python file in the library directory

**Step 3: Make the Module Executable**

**Step 4: Use the Custom Module in a Playbook:** Create a playbook that uses your custom module.

**Step 5: Run the Playbook**

**Plugins**

**Plugins** in Ansible are pieces of code that extend Ansible’s core functionality. They allow you to customize or enhance various aspects of Ansible’s behavior, such as how it connects to hosts, how it processes data, and how it logs information.

**Why Use Plugins?**

* **Customization**: Tailor Ansible to fit your specific needs.
* **Extend Functionality**: Add new features or modify existing ones.
* **Integration**: Connect Ansible with other tools and systems.

**Types of Plugins**: Ansible comes with several types of plugins, each serving a different purpose:

**1. Action Plugins:** Action plugins modify the behavior of Ansible tasks. They run on the control node and can change how tasks are executed.

**Example:** Imagine you want to create a custom action plugin that logs additional information when a task runs.

**2. Connection Plugins:** Connection plugins define how Ansible connects to hosts. For example, SSH is the default connection method, but you can create custom plugins for other methods.

**Example:** A custom connection plugin might connect to a host using a specific API.

**3. Lookup Plugins:** Lookup plugins fetch data from external sources. For example, you can use a lookup plugin to get data from a database or a web service.

**Example:** Using the file lookup plugin to read the contents of a file.

**4. Filter Plugins”** Filter plugins modify the output of variables. They are used within Jinja2 templates to transform data.

**Example:** Using the upper filter to convert a string to uppercase.

**5. Callback Plugins:** Callback plugins allow you to hook into various stages of Ansible’s execution to log information, send notifications, or perform other actions.

**Example:** A callback plugin might send a notification to a chat application whenever a playbook starts or finishes.

**6. Inventory Plugins:** Inventory plugins dynamically generate inventory from external sources like cloud providers, databases, or other services.

**Example:** Using the AWS EC2 inventory plugin to fetch hosts from AWS

**7. Strategy Plugins:** Strategy plugins control the order and manner in which tasks are executed on hosts. The default strategy is linear, but you can create custom strategies for different execution patterns.

**Example:** A custom strategy plugin might execute tasks in parallel across hosts.