What is Ansible ?

Ansible is a software tool that provides simple but powerful automation for cross-platform computer support. It is primarily intended for IT professionals, who use it for application deployment, updates on workstations and servers, cloud provisioning, configuration management, intra-service orchestration, and nearly anything a systems administrator does on a weekly or daily basis. Ansible doesn't depend on agent software and has no additional security infrastructure, so it's easy to deploy.

Because Ansible is all about automation, it requires instructions to accomplish each job. With everything written down in simple script form, it's easy to do version control. The practical result of this is a major contribution to the "infrastructure as code" movement in IT: the idea that the maintenance of server and client infrastructure can and should be treated the same as software development, with repositories of self-documenting, proven, and executable solutions capable of running an organization regardless of staff changes.

While Ansible may be at the forefront of automation, systems administration, and DevOps, it's also useful to everyday users. Ansible allows you to configure not just one computer, but potentially a whole network of computers at once, and using it requires no programming skills. Instructions written for Ansible are human-readable. Whether you're entirely new to computers or an expert, Ansible files are easy to understand.

How Ansible works :

In Ansible, there are two categories of computers: the control node and managed nodes. The control node is a computer that runs Ansible.

Ansible works by connecting to nodes (clients, servers, or whatever you're configuring) on a network, and then sending a small program called an Ansible module to that node. Ansible executes these modules over SSH and removes them when finished. The only requirement for this interaction is that your Ansible control node has login access to the managed nodes. SSH keys are the most common way to provide access, but other forms of authentication are also supported.

Ansible Installation:

Control node requirements

For your control node (the machine that runs Ansible), you can use nearly any UNIX-like machine with Python 3.9 or newer installed. This includes Red Hat, Debian, Ubuntu, macOS, BSDs.

Managed node requirements

The managed node (the machine that Ansible is managing) does not require Ansible to be installed, but requires Python 2.7, or Python 3.5 - 3.11 to run Ansible library code. The managed node also needs a user account that can SSH to the node with an interactive POSIX shell.

Node requirement summary

The table below lists the current and historical versions of Python required on control and managed nodes.

ansible-core Version	Control node Python	Managed node Python
2.11	Python 2.7, Python 3.5 - 3.9 [†]	Python 2.6 - 2.7, Python 3.5 - 3.9
2.12	Python 3.8 - 3.10	Python 2.6 - 2.7, Python 3.5 - 3.10
2.13	Python 3.8 - 3.10	Python 2.7, Python 3.5 - 3.10
2.14	Python 3.9 - 3.11	Python 2.7, Python 3.5 - 3.11

Installation Steps on CentOs 9:

#which python **(Locate Python Installation Location)**

#python --version **(Verify python is installed with compatable Version)**

Setting up EPEL Repository on CentOS 9

#dnf config-manager --set-enabled crb

#dnf install epel-release epel-next-release

#cd /etc/yum.repos.d **(Place where all the repo files are stored)**

#ls **(verify EPEL repo is configured)**

#yum install ansible -y **(Installing Ansible)**

#which ansible **(Locate the ansible installation Location)**

#ansible --version **(Displays Ansible Version)**

Ansible Inventory :

Ansible is a modern configuration management tool that facilitates the task of setting up and maintaining remote servers, with a minimalist design intended to get users up and running quickly. Ansible uses an inventory file to keep track of which hosts are part of your infrastructure, and how to reach them for running commands and playbooks.

There are multiple ways in which you can set up your Ansible inventory file, depending on your environment and project needs. In this guide, we'll demonstrate how to create inventory files and organize servers into groups and subgroups, how to set up host variables, and how to use patterns to control the execution of Ansible commands and playbooks per host and per group.

Upon installation, Ansible creates an inventory file that is typically located at /etc/ansible/hosts. This is the default location used by Ansible when a custom inventory file is not provided with the -i option, during a playbook or command execution.

Even though you can use this file without problems, using per-project inventory files is a good practice to avoid mixing servers when executing commands and playbooks. Having per-project inventory files will also facilitate sharing your provisioning setup with collaborators, given you include the inventory file within the project's code repository.

Organizing Servers Into Groups and Subgroups :-

Within the inventory file, you can organize your servers into different groups and subgroups. Beyond helping to keep your hosts in order, this practice will enable you

to use group variables, a feature that can greatly facilitate managing multiple staging environments with Ansible.

A host can be part of multiple groups. The following inventory file in INI format demonstrates a setup with four groups: webservers, dbservers, development, and production. You'll notice that the servers are grouped by two different qualities: their purpose **(web and database)**, and how they're being used **(development and production).**

#vi /etc/ansible/hosts
[webservers]
192.168.1.115
192.168.1.116

[dbservers]
192.168.1.117
192.168.1.118

[development]
192.168.1.119
192.168.1.120

[production]
192.168.1.121
192.168.1.122

[servers:children]
webservers
dbservers
:wq!

Listing all the Servers from a specific Host Group
#ansible webservers --list-hosts

Ansible Config File (/etc/ansible/ansible.cfg) :-
With a fresh installation of Ansible, like every other software, it ships with a default configuration file. This is the brain and the heart of Ansible, the file that governs the behavior of all interactions performed by the control node. In Ansible's case that default configuration file is (ansible.cfg) located in /etc/ansible/ansible.cfg.

We can mention our ansible.cfg file in different locations. While running your ansible it will follow some priority and based on that it will select correct ansible.cfg file.

Location with Priority :-
1)ANSIBLE_CONFIG Environmental variable
2) ./ansible.cfg
3) /etc/ansible/ansible.cfg

NOTE :

We can run Tasks from ansible engine in 2 ways
1) AD-hoc Commands
2) Playbooks

AD-hoc : An Ansible ad hoc command uses the /usr/bin/ansible command-line tool to automate a single task on one or more managed nodes. ad hoc commands are quick and easy, but they are not reusable. Ansible AD-hoc commands are useful to execute one task on your remote clients at a time

Playbook :
An Ansible Playbook is a blueprint of automation tasks—which are complex IT actions executed with limited or no human involvement. Ansible Playbooks are executed on a set, group, or classification of hosts, which together make up an Ansible inventory.

Ansible Playbooks are essentially frameworks, which are prewritten code developers can use ad-hoc or as starting template. Ansible Playbooks are regularly used to automate IT infrastructure (such as operating systems and Kubernetes platforms), networks, security systems, and developer personas (such as Git).

Ansible Playbooks help IT staff program applications, services, server nodes, or other devices without the manual overhead of creating everything from scratch. And Ansible Playbooks—as well as the conditions, variables, and tasks within them—can be saved, shared, or reused indefinitely.

#ansible db –m copy –a “src=./hosts dest=/tmp/inv1”

Rerun to check it is Idempotent

#ansible db –m copy –a “src=./hosts dest=/tmp/inv1”

#ansible db –m copy –a “src=./hosts dest=/tmp/inv1”

#ansible db –m copy –a “content=’Hello, GM’ dest=/tmp/hello.txt”

#ansible db –m copy –a “content=’Updated content’ dest=/tmp/hello.txt backup=yes”

#ansible db –m copy –a “content=’This is from db servers\n’ dest=/home/ansadmin/db.conf”

#ansible db –m fetch –a “src=/home/ansadmin/demo.txt dest=./demo/”

#ansible db –m fetch –a “src=/home/ansadmin/demo.txt dest=./newdemo/{{inventory_hostname}}_demo.txt flat=yes”

#ansible db –m file –a “path=/tmp/hello.txt state=touch”

#ansible db –m file –a “path=/tmp/newfile.txt stat=touch mode=’0777’”

#ansible db –m file –a “path=/tmp/newfile.txt state=absent”

#ansible db –m file –a “path=/tmp/hello state=directory”

Playbooks

EX :

#vi play1.yml

```
---
 - name : This Play is to create a user
   hosts : 192.168.0.105
   tasks :
    - name : Task1 of Play1 ( Creating user Rajesh )
      user :
        name : rajesh
```

```
        uid : 9009
        state : present
:wq!
```

#ansible-playbook - - syntax-check play1.yml (To verify Syntax errors)
#ansible-playbook -C play1.yml (Performing Dry run)
#ansible-playbook play1.yml (Executing Playbook)

EX :

#vi play2.yml

```
---
 - name : Setting up NTP Server and Mail Server( Play1 )
   hosts : db
   tasks :
    - name : Task1 of Play1 ( Installing NTP Package )
      yum :
        name : chrony
        state : latest
    - name : Task2 of Play1 ( Starting NTP Service )
      service :
        name : chronyd
        state : started
        enabled : true
    - name : Task3 of Play1 ( Installing Mail package )
      yum :
        name : postfix
        state : latest
    - name : Task4 of Play1 ( Starting Mail Service )
      service :
        name : postfix
        state : started
        enabled : true
:wq!
```

EX :

#vi play3.yml

```
---
 - name : Setting up Webserver (Play1)
```

```
  hosts : db
  tasks :
   - name : Task1 of Play1 ( Installing Apache Package )
     yum :
       name : httpd
       state : latest
   - name : Task2 of Play1 ( Defining index.html file )
     copy :
       src : /web/index.html
       dest : /var/www/html/index.html
   - name : Task3 of Play1 ( Starting Apache Service )
     service :
       name : httpd
       state : started
       enabled : true
:wq!
```

EX :

```
#vi play4.yml
---
 - name : Setting up DNS Server
   hosts : db
   remote_user : ansadmin
   become : true
   tasks :
    - name : Installing DNS Package
      yum :
        name : bind
        state : latest
    - name : Starting DNS Service
      service :
        name : named
        state : started
        enabled : true
:wq!
```

EX :

```
#vi play5.yml
---
```

```
 - name : Setting up DNS Server
   hosts : db
   tasks :
    - name : Installing DNS Package
      yum :
        name : bind
        state : latest
    - name : Starting DNS Service
      service :
        name : named
        state : started
        enabled : true
:wq!
```

EX :

```
#vi play6.yml
---
 - name : SOme Playbook
   hosts :
    - 192.168.0.105
    - 192.168.0.236
    - 192.168.0.116
   remote_user : ansadmin
   become : true
   ignore_errors : true
   gather_facts : false
   tasks :
    - name : Installing Some selected packages
      yum :
        name :
         - httpd1
         - wget
         - curl
        state : latest
    - name : Defining index.html file
      copy :
        content : "welcome to my webserver!\n"
        dest : /var/www/html/index.html
    - name : starting Apache Service
      service :
```

```
      name : httpd
      state : started
      enabled : true
    - name : Ensure firewall is started & Enabled
      service :
      name : firewalld
      state : started
      enabled : true
    - name : adding port 80 to firewall rule
      firewalld :
      port : 80/tcp
      permanent : true
      state : enabled
      immediate : yes
:wq!
```

Ansible Variables :

variables are used to store and manipulate data within playbooks.They allow you to make your configurations more dynamic and flexible. Ansible variables can be defined at different levels, such as global scope, play scope, or host scope.

Global Variables: These variables are defined in a configuration file or can be passed through the command line. They can be accessed from any playbook or role.

Play Variables: These variables are specific to a particular playbook and are defined within that playbook. They can be used by all tasks within that playbook.

Host Variables: These variables are defined for individual hosts or groups of hosts. They are useful for setting different values for specific hosts or groups.

EX :

```
#vi play7.yml
---
 - name : Performing few tasts with variables
   hosts : db
```

```
  remote_user : ansadmin
  become : true
  vars :
   user : varun
  tasks :
   - name : Creating user {{user}}
     user :
       name : "{{user}}"
:wq!
```

EX :

```
#vi play8.yml
---
 - name : Setting up Web server for front end prod Appln
   hosts : db
   gather_facts : false
   ignore_errors : true
   remote_user : ansadmin
   become : true
   vars :
    web_pkg : httpd
    firewall_pkg : firewalld
    web_service : httpd
    firewall_service : firewalld
    rule : http
   tasks :
    - name : Installing Apache package
      yum :
        name :
          - "{{web_pkg}}"
          - "{{firewall_pkg}}"
        state : latest
    - name : The {{firewall_service}} need to be started
      service :
        name : "{{firewall_service}}"
        enabled : true
        state : started
    - name : The {{web_service}} need to be started
      service :
```

```
      name : "{{web_service}}"
      state : started
      enabled : true
  - name : Defining index.html
    copy :
      content : "Welcome to my Web Appln"
      dest : /var/www/html/index.html
  - name : Adding http to the firewall rule
    firewalld :
      service : "{{rule}}"
      permanent : true
      immediate : true
      state : enabled
:wq!
```

Ansible Vault :

It is a feature of ansible that allows you to keep sensitive data such as passwords or keys in encrypted format rather than as plain text in playbooks or roles. The vault files can then be distributed or placed in version source control.

Ansible Vault is a feature in Ansible that allows you to encrypt sensitive data, such as passwords, API keys, or any other confidential information, within your playbooks or variable files. It provides a secure way to store and manage sensitive data as part of your Ansible automation workflows.

Creating an encrypted file

#ansible-vault create play11.yml

#ansible-vault create - - vault-password-file=/passwords/file1 play12.yml **(Non interactively setting the password)**

#ansible-vault view play11.yml **(Viewing an encrypted file)**

#ansible-vault edit play11.yml **(Editing an existing encrypted file)**

#ansible-vault encrypt play1.yml play2.yml play3.yml **(Encrypting an existing files)**

#ansible-vault decrypt play1.yml **(Decrypting an existing file)**

#ansible-vault decrypt play1.yml - - output=decrypted-play1.yml **(Creating a decrypted file with out decrypting the source file)**

#ansible-vault rekey play1.yml **(Changing the password of an existing encrypted file)**

#ansible-playbook - -vault-id @prompt play2.yml **(Interactively executing the encrypted playbook)**

#ansible-playbook - -vault-password-file=/passwords/file1 play3.yml **(Non-Interactively executing the encrypted playbook)**

Ansible Loops :

Ansible loops are a way to repeat a task or a set of tasks multiple times with different values. They allow you to iterate over a list, dictionary, or other data structures in Ansible playbooks, making your configurations more dynamic and flexible.

There are several types of loops available in Ansible:

loop: This is the most commonly used loop in Ansible. It allows you to iterate over a list of items and perform tasks for each item in the list.

Ansible loops are a way to repeat a task or a set of tasks multiple times with different values. They allow you to iterate over a list, dictionary, or other data structures in Ansible playbooks, making your configurations more dynamic and flexible.

Examples :

#Vi loops1.yml

```
---

 - name : Installing Some packages
   hosts : db
   gather_facts : false
   remote_user : ansadmin
   become : true
   tasks :
```

```
    - name : Installing Some Packages
      yum :
        name : "{{item}}"
        state : latest
      loop :
        - net-tools
        - sysstat
        - dovecot
        - postfix
:wq!

#vi loops2.yml
---
 - name : Ensuring Postfix & Dovecot started
   hosts : db
   gather_facts : false
   remote_user : ansadmin
   become : true
   vars :
    mail_services :
      - postfix
      - dovecot
   tasks :
    - name : Starting Mail Services
      service :
        name : "{{item}}"
        state : started
      loop : "{{mail_services}}"
:wq!
```

with_items: This is an older syntax for the loop loop, but it still works in Ansible. It allows you to iterate over a list of items.

```
#vi loops3.yml
---
 - name : Creating users with specific group
   hosts : db
   gather_facts : false
   remote_user : ansadmin
   become : true
```

```
  tasks :
   - name : creating users in selected groups
     user :
      name : "{{item.name}}"
      state : present
      groups : "{{item.groups}}"
     loop :
      - name : user31
        groups : g31
      - name : user32
        groups : g32
:wq!
```

NOTE : Ensure on the destination Ansible clients, Groups must exist.

Performing Ansible Tasks Conditionally

In Ansible, conditions, also known as conditional statements or tasks, allow you to control the flow of your playbooks based on certain conditions or variables. They help you make decisions and execute specific tasks only when the conditions are met.

Ansible provides several conditional statements that you can use in your playbooks:

when: The when statement is the most commonly used conditional statement in Ansible. It allows you to specify a condition that determines whether a task should be executed or skipped. The condition can be based on variable values, facts gathered from hosts, or the result of a previous task.

failed_when: The failed_when statement allows you to specify a condition that determines when a task is considered failed. By default, a task fails if the command it executes returns a non-zero exit status. With failed_when, you can customize the failure condition based on specific criteria.

changed_when: The changed_when statement is used to override the default determination of whether a task has made changes on the target system. By default, Ansible considers a task as changed if it has made any modifications. With changed_when, you can define custom conditions for when a task should be marked as changed.

```
#vi condition.yml
---
 - name : Installing DB
   hosts : db
   remote_user : ansadmin
   become : true
   tasks :
    - name : Installing mysql
      yum :
        name : mysql
        state : latest
      when :
        ansible_nodename == "ac3"
:wq!
```

Ansible Error Handling :

In Ansible, the ignore_errors attribute is used to control how Ansible handles errors encountered during the execution of tasks.

By default, when a task encounters an error (e.g., a command returns a non-zero exit status), Ansible stops executing subsequent tasks and marks the playbook as failed. However, you can use the ignore_errors attribute to override this behavior and allow the playbook to continue running even if errors occur.

Task-level ignore_errors: You can specify ignore_errors: true at the task level to indicate that Ansible should ignore any errors that occur during the execution of that specific task. This allows the playbook to continue running, even if the task fails.

```
- name: Task with ignore_errors
  yum : mycommand
    name : Installing Apache
    state : latest
  ignore_errors: true
```

Play-level ignore_errors: You can also define ignore_errors: true at the play level to indicate that Ansible should ignore errors for all tasks within that play. This can be useful if you want to continue executing other plays in the playbook even if errors occur in a particular play.

```
- name: Play with ignore_errors
  hosts: db
  gather_facts : true
  remote_user : ansadmin
  become : true
  ignore_errors: true
  tasks:
   - name: Task1 with ignore_errors
     yum :
       name : Installing Apache
       state : latest
   - name: Task2 with ignore_errors
     service :
       name : httpd
       state : started
       enabled : true
```

Ansible Tags :

Ansible tags allow you to selectively run or skip specific tasks, roles, or plays within your Ansible playbook. Tags provide a way to organize and control the execution of different parts of your playbook based on your needs.

Assigning Tags: You can assign tags to individual tasks, roles, or plays by using the tags attribute. Tags can be a single value or a list of values.

```
#vi tags.yml
---
  - name : Creating multiple users
    hosts : db
    gather_facts : false
    remote_user : ansadmin
    become : true
    tasks :
     - name : Crerating user41 ( Task 1 )
       user :
         name : user41
         shell : /bin/sh
       tags :
         - task1
```

```
  - name : Crerating user42 ( Task 2 )
    user :
      name : user42
      shell : /bin/sh
    tags :
      - task2
  - name : Crerating user43 ( Task 3 )
    user :
      name : user43
      shell : /bin/sh
    tags :
      - task3
  - name : Crerating user44 ( Task 4 )
    user :
      name : user44
      shell : /bin/sh
    tags :
      - task4
  - name : Crerating user45 ( Task 5 )
    user :
      name : user45
      shell : /bin/sh
    tags :
      - task5
:wq!

#ansible-playbook --list-tasks
#ansible-playbook tags.yml --tags all
#ansible-playbook tags.yml --tags task5
```

Ansible Handlers :

In Ansible, handlers are special tasks that are only executed when explicitly triggered. They are typically used to respond to specific events or changes in the system, such as service restarts or configuration updates.

Here's how handlers work in Ansible

Handlers are defined in the playbook under the handlers section. Each handler is a task that performs a specific action, such as restarting a service or reloading a configuration file.

Handlers are triggered using the notify keyword within other tasks. When a task notifies a handler, Ansible records it and executes the handler at the end of the play, after all tasks have been run. Multiple tasks can notify the same handler.

Handlers are executed sequentially at the end of the play, after all tasks have been completed. Ansible identifies which handlers need to be executed based on the notifications received during the play's execution.

Handlers are executed in the order they are defined in the playbook, regardless of the order in which they are notified.

Handlers are designed to be idempotent, meaning they can be executed multiple times without causing any harm or unnecessary changes. Ansible ensures that handlers are executed only once, even if multiple notifications are received.

Play book wiht out Handler

```
#vi without-handler.yml
---
 - name : Setting up a Webserver
   hosts : db
   tasks :
    - name : Installing Apache Package
      yum :
        name : httpd
        state : latest
    - name : Defining index.html
      copy :
        src : /web/index.html
        dest : /var/www/html/index.html
    - name : Defining httpd.conf
      copy :
        src : /web/httpd.conf
        dest : /etc/httpd/conf/httpd.conf
    - name : Starting Apache Service
      service :
        name : httpd
        state : restarted
        enabled : true
:wq!
```

Play book with Handler

```
#vi with-handler.yml
---
 - name : Setting up Web Server
   hosts : db
   tasks :
    - name : Installing Apache Pkg
      yum :
        name : httpd
        state : latest
      notify : restart apache
    - name : Defining index.html file
      copy :
        src : /web/index.html
        dest : /var/www/html/index.html
      notify : restart apache
    - name : Defining httpd.conf
      copy :
        src : /web/httpd.conf
        dest : /etc/httpd/conf/httpd.conf
      notify : restart apache
   handlers :
    - name : restart apache
      service :
        name : httpd
        state : restarted
        enabled : true
:wq!
```

Ansible Roles

In Ansible, roles are a way to organize and package reusable code that represents a specific functionality or configuration. Roles provide a structured approach to managing and sharing Ansible code across different playbooks and projects.

A role in Ansible typically consists of the following components:

Directory Structure: A role has a specific directory structure that helps organize its components. The main components of a role directory structure include:

roles
role_name
tasks: Contains the main tasks that define the role's functionality.
handlers: Contains the handlers that can be used to respond to events or triggers.
defaults: Contains default variable values for the role.
vars: Contains additional variable files that can be loaded by the role.
templates: Contains template files that can be used to generate configuration files.
files: Contains files that need to be copied to remote hosts.
meta: Contains metadata and dependencies information for the role.
Tasks: The tasks/ directory within a role contains the tasks that define the role's functionality. These tasks are written in YAML format and can include modules, loops, conditionals, and other Ansible constructs to perform various actions on target hosts.

Variables: Roles can define default variable values in the defaults/ directory. These variables provide sensible defaults for the role and can be overridden by the user. Roles may also have additional variable files in the vars/ directory to provide more flexibility.

Templates: The templates directory within a role contains template files that can be used to generate configuration files. These templates can include variables and expressions to customize the generated content.

Handlers: The handlers directory contains handlers that are defined within the role. Handlers are tasks that respond to specific events or triggers, such as service restarts or configuration changes.

Roles enable code reuse and modularization in Ansible by encapsulating related tasks, variables, templates, and handlers into a single package. They provide a structured approach to organizing and sharing Ansible code, making it easier to manage and maintain complex configurations and playbooks. Roles can be shared through Ansible Galaxy, a hub for sharing and discovering Ansible content.

Creating Apache Role

#cd /etc/ansible
#ls
#cd roles

```
#ansible-galaxy init apache --offline
#ls
#cd apache
#ls
#cd tasks
#vi main.yml
---
 - include_tasks : install.yml
 - include_tasks : configure.yml
 - include_tasks : service.yml
:wq!

#vi install.yml
---
 - name : Installing Apache Packages
   yum :
     name : httpd
     state : latest
:wq!
#vi configure.yml
---
 - name : Defining Custom Apache config file
   copy : src=httpd.conf dest=/etc/httpd/conf/httpd.conf
   notify : restart apache
 - name : Defining index.html file
   copy : src=index.html dest=/var/www/html/index.html
:wq!
#vi service.yml
---
 - name : Starting Apache Service
   service :
     name : httpd
     state : started
:wq!
#pwd
#cd ..
#pwd
#ls
#cd files
#ls
#yum install httpd -y ( Installing this for httpd.conf )
```

#cp /etc/httpd/conf/httpd.conf .
#vi index.html
Welcome to Pragathi Technologies Bangalore Ansible training
:wq!

#pwd
#cd ..
#pwd
#ls
#cd handlers
#vi main.yml

 - name : restart apache
   service :
     name : httpd
     state : restarted
:wq!

#pwd
#cd ..
#pwd
/etc/ansible
#vi site.yml

- hosts : db
  roles :
    - apache
:wq!

#ansible-playbook--syntax-check site.yml
#ansible-playbook site.yml