Introduction to Ansible

What is Ansible ?

Ansible is an open source IT automation engine that automates

- provisioning
- configuration management
- application deployment
- orchestration

and many other IT processes. It is free to use, and the project benefits from the experience and intelligence of its thousands of contributors.

How Ansible works ?

Ansible is agentless in nature, which means you don't need install any software on the manage nodes.

For automating Linux and Windows, Ansible connects to managed nodes and pushes out small programs—called Ansible modules—to them. These programs are written to be resource models of the desired state of the system. Ansible then executes these modules (over SSH by default), and removes them when finished. These modules are designed to be idempotent when possible, so that they only make changes to a system when necessary.

For automating network devices and other IT appliances where modules cannot be executed, Ansible runs on the control node. Since Ansible is agentless, it can still communicate with devices without requiring an application or service to be installed on the managed node.

Comparison with Shell Scripting

- Shell Scripting works only for Linux.
- Becomes complex and less readable(for non-experts) as the script size goes high.
- Idempotence and predictability

When the system is in the state your playbook describes Ansible does not change anything, even if the playbook runs multiple times.

for example, run the below shell script twice and you will notice the script will fail. Which means shell scripting is not idempotent in nature.

```
#/bin/bash

set -e

mkdir test-demo
echo "hi"
```

```
```

- Scalability and flexibility

Easily and quickly scale the systems you automate through a modular design that supports a large range of operating systems, cloud platforms, and network devices.

How to setup Passwordless Authentication

EC2 Instances

Using Public Key

```
ssh-copy-id -f "-o IdentityFile <PATH TO PEM FILE>" ubuntu@<INSTANCE-PUBLIC-IP>
```

- ssh-copy-id: This is the command used to copy your public key to a remote machine.
- -f: This flag forces the copying of keys, which can be useful if you have keys already set up and want to overwrite them.
- "-o IdentityFile ": This option specifies the identity file (private key) to use for the connection. The -o flag passes this option to the underlying ssh command.
- ubuntu@: This is the username (ubuntu) and the IP address of the remote server you want to access.

Using Password

- Go to the file `/etc/ssh/sshd_config.d/60-cloudimg-settings.conf`
- Update `PasswordAuthentication yes`
- Restart SSH -> `sudo systemctl restart ssh`

Inventory

Ansible inventory file is a fundamental component of Ansible that defines the hosts (remote systems) that you want to manage and the groups those hosts belong to. The inventory file can be static (a simple text file) or dynamic (generated by a script). It provides Ansible with the information about the remote nodes to communicate with during its operations.

Static Inventory

A static inventory file is typically a plain text file (usually named hosts or inventory) and is structured in INI or YAML format. Here are examples of both formats:

INI Format

```
# inventory file: hosts

[webservers]
```

```
web1.example.com
web2.example.com

[dbservers]
db1.example.com
db2.example.com

[all:vars]
ansible_user=admin
ansible_ssh_private_key_file=/path/to/key
```

YAML

```
# inventory file: hosts.yaml

all:
  vars:
    ansible_user: admin
    ansible_ssh_private_key_file: /path/to/key
  children:
    webservers:
      hosts:
        web1.example.com:
        web2.example.com:
    dbservers:
      hosts:
        db1.example.com:
        db2.example.com:
```

Dynamic Inventory

A dynamic inventory is generated by a script or plugin and can be used for environments where hosts are constantly changing (e.g., cloud environments). The script or plugin fetches the list of hosts from a source like AWS, GCP, or any other dynamic source.

Here is an example of a dynamic inventory script for AWS EC2:

```
#!/usr/bin/env python

import json
import boto3

def get_aws_ec2_inventory():
    ec2 = boto3.client('ec2')
    instances = ec2.describe_instances()
    inventory = {
        'all': {
            'hosts': [],
            'vars': {
                'ansible_user': 'ec2-user',
                'ansible_ssh_private_key_file': '/path/to/key'
            }
        },
        '_meta': {
            'hostvars': {}
        }
    }

    for reservation in instances['Reservations']:
        for instance in reservation['Instances']:
            if instance['State']['Name'] == 'running':
                public_ip = instance['PublicIpAddress']
                inventory['all']['hosts'].append(public_ip)
                inventory['_meta']['hostvars'][public_ip] = {
                    'ansible_host': public_ip
                }

    print(json.dumps(inventory, indent=2))

if __name__ == '__main__':
    get_aws_ec2_inventory()
```

Usage

```
ansible-playbook -i inventory <Adhoc command or Playbook.yml>
```

Understanding YAML

YAML (YAML Ain't Markup Language) is a human-readable data serialization format that is commonly used for configuration files and data exchange between languages with different data structures.

YAML Syntax

Strings, Numbers and Booleans:

```
string: Hello, World!
number: 42
boolean: true
```

List

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

Dictionary

```
person:
  name: John Doe
  age: 30
  city: New York
```

List of dictionaries

YAML allows nesting of lists and dictionaries to represent more complex data.

```
family:
  parents:
    - name: Jane
      age: 50
    - name: John
      age: 52
  children:
    - name: Jimmy
      age: 22
    - name: Jenny
      age: 20
```

Ansible Concepts: Playbook, Play, Modules, Tasks, and Collections

Playbook

A **Playbook** is a YAML file that defines a series of actions to be executed on managed nodes. It contains one or more "plays" that map groups of hosts to roles.

Example

```
---
- name: Update web servers
  hosts: webservers
  remote_user: root

  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

- name: Update db servers
  hosts: databases
  remote_user: root

  tasks:
  - name: Ensure postgresql is at the latest version
    ansible.builtin.yum:
      name: postgresql
      state: latest

  - name: Ensure that postgresql is started
    ansible.builtin.service:
      name: postgresql
      state: started
```

Play

A Play is a single, complete execution unit within a playbook. It specifies which hosts to target and what tasks to execute on those hosts. Plays are used to group related tasks and execute them in a specific order.

```
- name: Install and configure Nginx
  hosts: webservers
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
```

Modules

Modules are the building blocks of Ansible tasks. They are small programs that perform a specific action on a managed node, such as installing a package, copying a file, or managing services. Example

The apt module used in a task to install a package:

```
- name: Install Nginx
  apt:
    name: nginx
    state: present
```

Tasks

Tasks are individual actions within a play that use modules to perform operations on managed nodes. Each task is executed in order and can include conditionals, loops, and handlers.

```
- name: Install Nginx
  apt:
    name: nginx
    state: present

- name: Start Nginx service
  service:
    name: nginx
    state: started
```

Collections

Collections are a distribution format for Ansible content. They bundle together multiple roles, modules, plugins, and other Ansible artifacts. Collections make it easier to share and reuse Ansible content. Example

A collection structure might look like this:

```
my_collection/
├── roles/
│   └── my_role/
│       └── tasks/
│           └── main.yml
├── plugins/
│   └── modules/
│       └── my_module.py
└── README.md
```

Using a Collection

```
- name: Use a custom module from a collection
  community.general.my_module:
    option: value
```

```
---
- hosts: all
  become: true
  tasks:
    - name: Install apache httpd
      ansible.builtin.apt:
        name: apache2
```

```
          state: present
          update_cache: yes
      - name: Copy file with owner and permissions
        ansible.builtin.copy:
          src: index.html
          dest: /var/www/html
          owner: root
          group: root
          mode: '0644'
```

Ansible Roles

An Ansible role is a reusable, self-contained unit of automation that is used to organize and manage tasks, variables, files, templates, and handlers in a structured way.

Roles help to encapsulate and modularize the logic and configuration needed to manage a particular system or application component.

This modular approach promotes reusability, maintainability, and consistency across different playbooks and environments.

Key Components of an Ansible Role

Tasks

The main list of actions that the role performs.

Handlers

Tasks that are triggered by changes in other tasks, typically used for actions like restarting services.

Files

Static files that need to be transferred to managed hosts.

Templates

Jinja2 templates that can be rendered and transferred to managed hosts.

Vars

Variables that are used within the role.

Defaults

Default variables for the role, which can be overridden.

Meta

Metadata about the role, including dependencies on other roles.

Library

Custom modules or plugins used within the role.

Module_defaults

Default module parameters for the role.

Lookup_plugins

Custom lookup plugins for the role.

Directory Structure of an Ansible Role

An Ansible role follows a specific directory structure:

```
<role_name>/
  ├── defaults/
  │   └── main.yml
  ├── files/
  ├── handlers/
  │   └── main.yml
  ├── meta/
  │   └── main.yml
  ├── tasks/
  │   └── main.yml
  ├── templates/
  ├── vars/
      └── main.yml
```

Why Use Ansible Roles?

Modularity

Roles allow you to break down complex playbooks into smaller, reusable components. Each role handles a specific part of the configuration or setup.

Reusability

Once created, roles can be reused across different playbooks and projects. This saves time and effort in writing redundant code.

Maintainability

By organizing related tasks into roles, it becomes easier to manage and maintain the code. Changes can be made in one place and applied consistently wherever the role is used.

Readability

Roles make playbooks cleaner and easier to read by abstracting away the details into logically named roles.

Collaboration

Roles facilitate collaboration among team members by allowing them to work on different parts of the infrastructure independently.

Consistency

Using roles ensures that the same setup and configuration procedures are applied uniformly across multiple environments, reducing the risk of configuration drift.

Push your Ansible roles to Ansible Galaxy

1. Make sure your role is structured correctly. The basic structure should look like this:

```
my_role/
├── defaults/
│   └── main.yml
├── files/
├── handlers/
│   └── main.yml
├── meta/
│   └── main.yml
├── tasks/
│   └── main.yml
├── templates/
├── tests/
│   ├── inventory
│   └── test.yml
└── vars/
    └── main.yml
```

2. Make sure ansible-galaxy CLI exists

```
ansible-galaxy --version
```

3. Push Your Role to GitHub

```
cd <role-name>
git init
git remote add origin <https://github.com/your_github_username/my_role.git>
git add .
git commit -m "Initial commit"
git push -u origin main
```

4. Import the Role to Ansible Galaxy

```
ansible-galaxy role import <your_github_username> <role-name>
```

Setup EC2 Collection and Authentication

Install boto3

```
pip install boto3
```

Install AWS Collection

```
ansible-galaxy collection install amazon.aws
```

Setup Vault

1. Create a password for vault

```
openssl rand -base64 2048 > vault.pass
```

2. Add your AWS credentials using the below vault command

```
ansible-vault create group_vars/all/pass.yml --vault-password-file vault.pass
```

```
---
- hosts: localhost
  connection: local
  tasks:
  - name: start an instance with a public IP address
    amazon.aws.ec2_instance:
      name: "ansible-instance"
      # key_name: "prod-ssh-key"
      # vpc_subnet_id: subnet-013744e41e8088axx
      instance_type: t2.micro
      security_group: default
      region: us-east-1
      aws_access_key: "{{ec2_access_key}}"  # From vault as defined
      aws_secret_key: "{{ec2_secret_key}}"  # From vault as defined
      network:
        assign_public_ip: true
      image_id: ami-04b70fa74e45c3917
      tags:
        Environment: Testing
```

Ansible Realtime project

Task 1

Create three(3) EC2 instances on AWS using Ansible loops

- 2 Instances with Ubuntu Distribution
- 1 Instance with Centos Distribution

Hint: Use `connection: local` on Ansible Control node.

Task 2

Set up passwordless authentication between Ansible control node and newly created instances.

Task 3

Automate the shutdown of Ubuntu Instances only using Ansible Conditionals

Hint: Use `when` condition on ansible `gather_facts`

EC2_create -

```
---
- hosts: localhost
  connection: local

  tasks:
  - name: Create EC2 instances
    amazon.aws.ec2_instance:
      name: "{{ item.name }}"
      key_name: "abhi-aws-keypair"
      instance_type: t2.micro
      security_group: default
      region: ap-south-1
      aws_access_key: "{{ec2_access_key}}"  # From vault as defined
      aws_secret_key: "{{ec2_secret_key}}"  # From vault as defined
      network:
        assign_public_ip: true
      image_id: "{{ item.image }}"
      tags:
        environment: "{{ item.name }}"
    loop:
      - { image: "ami-0e1d06225679bc1c5", name: "manage-node-1" } # Update AMI ID 
according
      - { image: "ami-0f58b397bc5c1f2e8", name: "manage-node-2" } # to your
```

```
account
      - { image: "ami-0f58b397bc5c1f2e8", name: "manage-node-3" }
```

EC2_shutdown -

```
---
- hosts: all
  become: true

  tasks:
    - name: Shutdown ubuntu instances only
      ansible.builtin.command: /sbin/shutdown -t now
      when:
       ansible_facts['os_family'] == "Debian"
```

Error handling

```
---
- hosts: all
  become: true

  tasks:
    - name: Install security updates
      ansible.builtin.apt:
        name: "{{ item }}"
        state: latest
      loop:
        - openssl
        - openssh
      ignore_errors: yes
    - name: Check if docker is installed
      ansible.builtin.command: docker --version
      register: output
      ignore_errors: yes
    - ansible.builtin.debug:
        var: output
    - name: Install docker
      ansible.builtin.apt:
        name: docker.io
        state: present
      when: output.failed
```

MISC:

Ansible Playbook and Role Management

Running Ansible Playbooks

Example Commands

- Run a playbook with extra variables:

  ```
  ansible-playbook first-playbook.yaml -i inventory --extra-vars
  "node1_ip=10.221.242.53 username=root root_password=Gyp.s8m" -v
  ```

- Ping all nodes in the inventory with extra variables:

  ```
  ansible -i inventory -m ping all --extra-vars "node1_ip=10.221.242.53
  username=root root_password=Gyp.s8m" -v
  ```

- Run a shell command on all nodes:

  ```
  ansible -i inventory -m shell -a "ls /etc" all --extra-vars
  "node1_ip=10.221.242.53 username=root root_password=Gyp.s8m" -v
  ```

Working with Roles

Importing Roles from Ansible Galaxy

Ansible roles can be imported from Ansible Galaxy, which acts like Docker Hub for Ansible roles.

- To initialize a new role:

  ```
  ansible-galaxy role init test
  ```

- To install a role from Ansible Galaxy:

  ```
  ansible-galaxy role install <role_name>
  ```

  This command will make the role available in `~/.ansible/roles`.

Importing Custom Roles to Ansible Galaxy

1. Create a role and push the entire directory to a GitHub repository.
2. Import the role into Galaxy using the following command:

   ```
   ansible-galaxy import <github_username> <repo_name> <api_token>
   ```

You can generate the `api_token` from the Ansible Galaxy website under **Collections -> API Token -> Load Token**.

Working with Collections

Install Collections from Ansible Galaxy

To install a collection from Ansible Galaxy:

```
ansible-galaxy collection install <collection_name>
```

Ansible Vault

Ansible Vault is a tool for securely storing sensitive data such as passwords, private keys, or AWS credentials. It works like a "bank locker" where you can store sensitive information and access it directly within your playbook.

Steps to Create and Use Ansible Vault

let's say we have to hide aws credentials.

1. create a random password for vault. -> openssl rand -base64 2048 > vault.pass

```
openssl rand -base64 2048 > vault.pass
```

2. create a vault and pass the above password ->

```
ansible-vault create group_vars/all/aws-creds.yaml --vault-password-file
vault.pass
```

a vault (vi editor window) will open where we need to put your credentials in "key: value" pairs. Now save the file and use the same keys in playbook to access the values. If we run -> cat aws-creds.yaml then we can see encryoted data. If we want to see decrypted data then run ->

```
ansible-vault decrypt aws-creds.yaml --vault-password-file vault.pass
```

Note: We need a password to decrypt.