

Agenda:

- ```
=====
1. Introduction to Configuration management
2. CM tool - Ansible
3. Architecture of Ansible
4. Setup Ansible on the lab
5. Ansible Inventory and adhoc commands
```

```
=====
Configuration
```

```
=====
making a change on a server or set of servers
```

- > installing package
- > copying files
- > executing commands
- > executing scripts
- > deploying application
- > creation of Users
- > giving permissions

I want to do configuration change on 200 servers - Manually

- > Time Consuming
- > repetitive
- > error prone
- > trouble shooting is time Consuming
- > all updates may not be available
- > keep track of all changes will be difficult

Configuration management: Ansible

- ```
=====
> Using an automation tool to make changes on several servers in very less time
> Using an automation tool to bring in consistency across various environments
> A automation tool will ensure our desired changes are always available on the destination
  servers
> In Configuration management tool we will write code to make changes on various servers
> This code can be maintained in VC tools
> CM tools will also maintain logs of what changes were done on which servers
```

Infrastructure as a code tool: Terraform

=====

We can write the code to provision(create/modify/delete) infrastructure on the cloud

CM tool - Ansible

=====

- It is an open source tool - Free to use
- It is a simple tools and easy to learn
- Ansible works on a push approach
- Ansible 2 parts:
 - Ansible Core : Ansible is available as Command line tool
 - Ansible Tower/AWX : GUI of ansible or Ansible dashboard
- Ansible will always be installed on linux machine
- Cannot be installed on Windows OS
- Ansible can connect to linux or windows servers and apply changes on them
- It is python based tools- we required python >=2.7
- Ansible automation code is written in YAML
- Ansible is setup as a master and worker node architecture
- Installation of ansible is very easy
- No ansible process will be installed on the worker node - Agentless tool
- Can automate cloud machines or on premise machines

=====

Ansible Components

1. Inventory: WHERE TO DO THE CHANGES

=====

- It is a simple file in which we will write IP address or hostnames of the servers where ansible has to changes
- This file is present by default at location /etc/ansible
- Default name of the file is hosts
- We can create our own inventory file in any directory

There are 2 types of inventory :

=====

Static Inventory:

=====

where the infrastructure is static

This file is manually created by the user

Ip address are manually written by the user

Whenever we have less or limited infra then go for static inventory

Dynamic Inventory :

=====

This inventory file is created by Ansible
The inventory IP address are dynamic and keep changing
This type of inventory is created when your infra is on the cloud
Ansible --> connect to cloud provider --> ansible will go to given region of aws --> check the available VM
-> fetch the hostname/IP and ---> compute the inventory file

2. Modules : What Changes to do

Modules are small-small python programs (ansible pre-written code) provided to us by ansible
These are reusable code which when executed will perform configuration change on the servers.

There 4000+ modules

For example :

Download the file from <https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war> at location /opt on all servers

```
get_url(url=https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war dest=/opt)
```

3. Playbook

- ansible code is written inside a playbook
- Playbooks are written in YAML
- Playbook consist of:
 - host IP details(come form Inventory file)
 - Tasks details(modules to be executed)

4. ansible.cfg

This is default file of ansible

This file is present at /etc/ansible directory

The name of the ansible.cfg

This is the main configuration file of ansible

There are 3 main sections in this file

1. defaults: in this section ansible will note
 - inventory location
 - module lib location
 - default ssh user
 - > fork
 - > ssh connection timeout

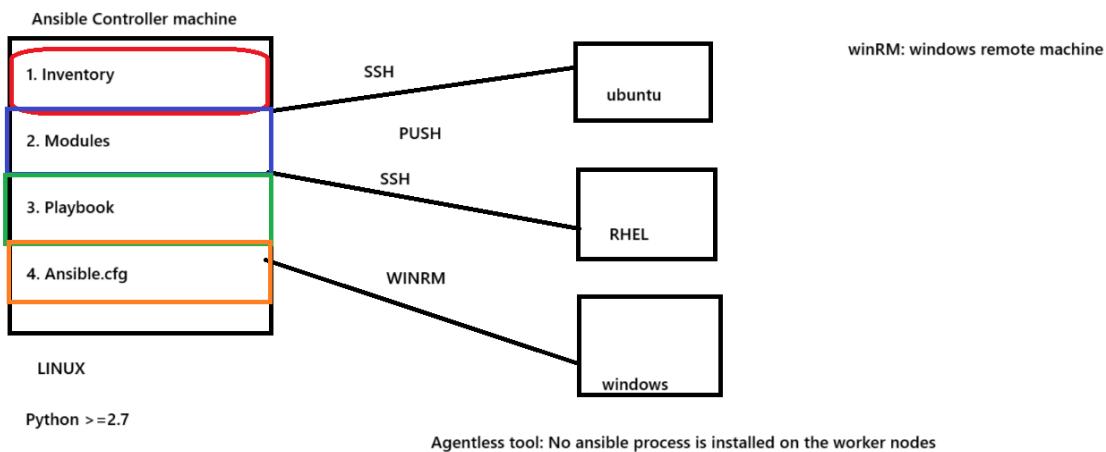
- ansible log path

2. SSH_Connection

SSh related details will be written inside this file

3. Privilege Escalation

in this file we provide information related to sudo users and its privileges



Launch Configuration management LAB

There are 2 labs:

AWS lab

- > Create EC2 virtual Machine - ubuntu
- > Copy the SSH keys of the DevOps Lab(ACM)
- > This machine becomes the host server.

DevOps Lab (New)

ubuntu 22 machine

Python 3

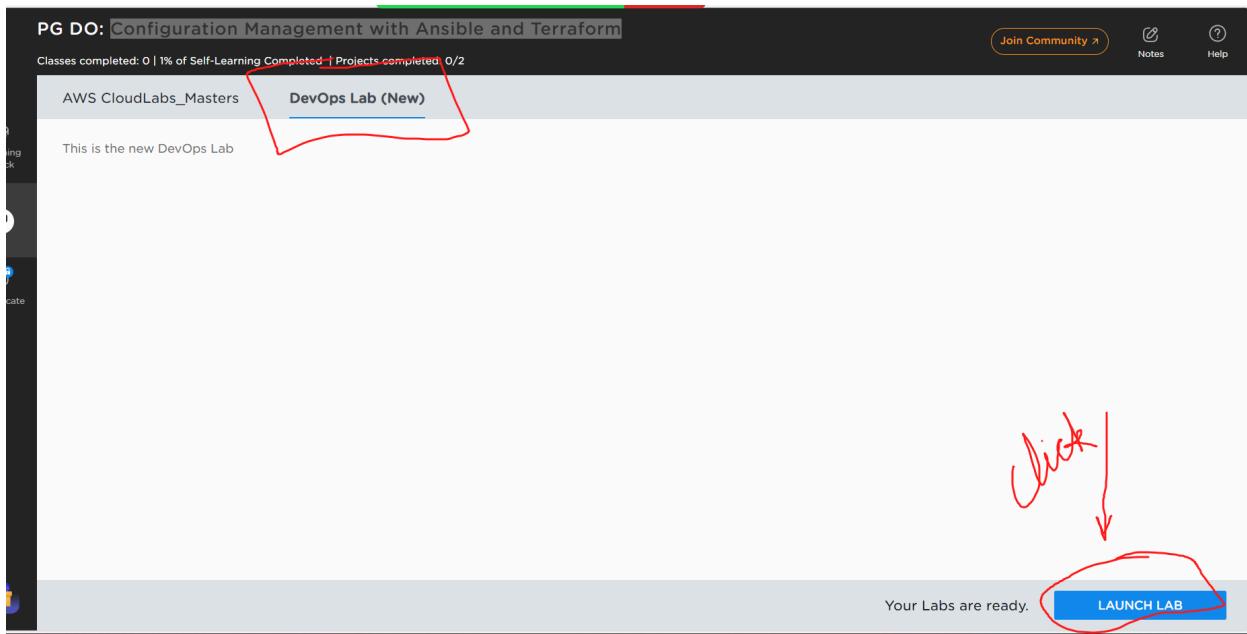
Ansible is already installed

It is Ansible controller machine

We can create current ACM as an inventory itself -> local host
Ansible can execute playbook on itself

=====

Connect to SL Lab - Configuration Management with Ansible and Terraform



Click on start lab:

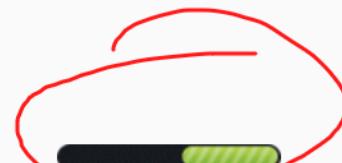
PG DO: Configuration Management with Ansible and Terraform

Classes completed: 0 | 1% of Self-Learning Completed | Projects completed: 0/2

AWS CloudLabs_Masters

DevOps Lab (New)

DevopsELK



Practice Labs | PG DO: Configuration Management with Ansible and Terraform | i-0af4b945fb6df9ec | +

<https://lms.simplilearn.com/courses/7158/PG-DO--Configuration-Management-with-Ansible-and-Terraform/practice-labs>

PG DO: Configuration Management with Ansible and Terraform

Classes completed: 0 | 1% of Self-Learning Completed | Projects completed: 0/2

AWS CloudLabs_Masters DevOps Lab (New) This Lab will get res...

Learning Track Certificate

DevopsELK

Web Desktop. ▾

Applications

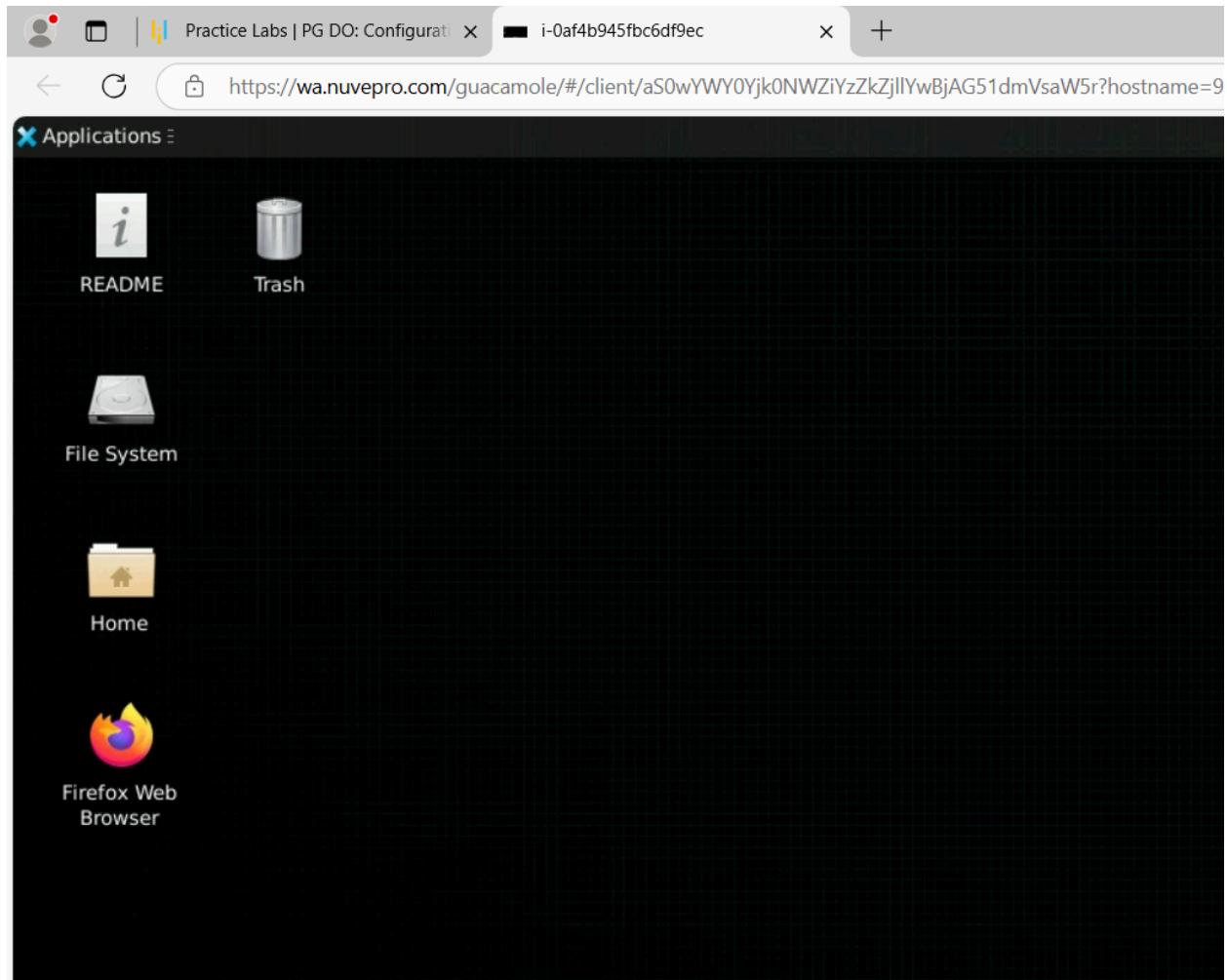
README Trash

Sat 12 Oct, 16:11 labuser

Notifications

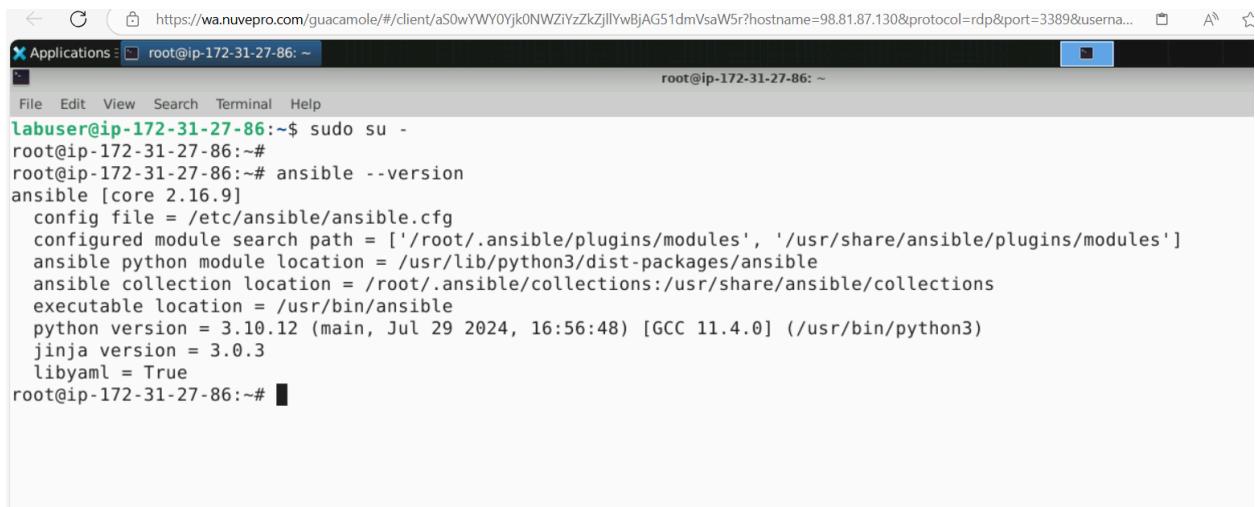
Instructions

A red hand-drawn box highlights the 'Join' button in the top right corner of the main header area. Another red hand-drawn circle highlights the progress bar at the bottom of the screen, specifically the green portion of the bar.



Start terminal on the Lab and check if ansible installed or not

```
# sudo su -  
  
# ansible --version
```



The screenshot shows a terminal window titled "root@ip-172-31-27-86: ~". The window displays the output of the "ansible --version" command. The output includes details about the ansible configuration file, module search paths, Python module location, collection location, executable location, Python version, Jinja version, and libyaml status.

```
labuser@ip-172-31-27-86:~$ sudo su -
root@ip-172-31-27-86:~
root@ip-172-31-27-86:~# ansible --version
ansible [core 2.16.9]
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/root/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3/dist-packages/ansible
  ansible collection location = /root/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/bin/ansible
  python version = 3.10.12 (main, Jul 29 2024, 16:56:48) [GCC 11.4.0] (/usr/bin/python3)
  jinja version = 3.0.3
  libyaml = True
root@ip-172-31-27-86:~#
```

Open SSH connection between Controller and worker nodes:

ON the Controller Machine -> DevOps Lab

sudo su -

Create a new user on the terminal:

adduser ansiuser

Enter New password : **ansiuser**

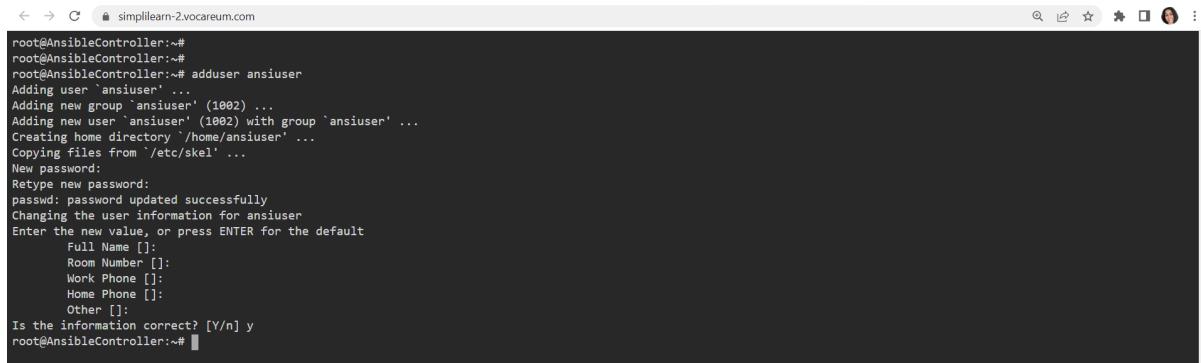
Retype new Password: **ansiuser**

Don't enter any value for fullname, room number, workphone, homephone other

Just keep pressing enter key.

And give Y for

Is the information correct ? [Y/n] : y



```
← → C simplilearn-2.vocareum.com
root@AnsibleController:~#
root@AnsibleController:~# adduser ansiuser
Adding user 'ansiuser' ...
Adding new group 'ansiuser' (1002) ...
Adding new user 'ansiuser' (1002) with group 'ansiuser' ...
Creating home directory '/home/ansiuser' ...
Copying files from '/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for ansiuser
Enter the new value, or press ENTER for the default
  Full Name []:
  Room Number []:
  Work Phone []:
  Home Phone []:
  Other []
Is the information correct? [Y/n] y
root@AnsibleController:~#
```

User will now be created.

Step 2:

We will give sudo permission to this user so that it can execute linux commands without need of password

Add ansiuser in sudoers files and give all permission

vim /etc/sudoers

Press i

Scroll down until your find : # User privilege specification

Now enter below line under #labuser ALL=(ALL) NOPASSWD:ALL

ansiuser ALL=NOPASSWD: ALL

```
# While you shouldn't normally run git as root
Defaults:%sudo env_keep += "GIT_AUTHOR_* GIT_

# Per-user preferences; root won't have sensible
Defaults:%sudo env_keep += "EMAIL DEBEMAIL DE

# "sudo scp" or "sudo rsync" should be able to
Defaults:%sudo env_keep += "SSH_AGENT_PID SSH

# Ditto for GPG agent
Defaults:%sudo env_keep += "GPG_AGENT_INFO"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL
labuser  ALL=(ALL) NOPASSWD:ALL
ansiuser ALL=NOPASSWD: ALL
# Members of the admin group may gain root privileges
%admin  ALL=(ALL) ALL
```

Save the file. (:wq!)

Step 4:

Generate SSH key on Master machine

Step 1:

Change user from root to ansiuser

```
# su - ansiuser
```

Step 2:

Generate ssh key on Master node for ansiuser

Execute below command:

```
# ssh-keygen
```

press enter

press enter

press enter

ssh key will be generated

```
← → ⌂ simplilearn-2.vocareum.com
root@AnsibleController:~# su - ansiuser
ansiuser@AnsibleController:~$ ssh-keygen
ansiuser@AnsibleController:~$ Generating public/private rsa key pair.
Enter file in which to save the key (/home/ansiuser/.ssh/id_rsa):
Created directory '/home/ansiuser/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ansiuser/.ssh/id_rsa
Your public key has been saved in /home/ansiuser/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:z7AzPWXMtaj+DfF88JUW8da2sCX5Vi+lSQiakv4EwDQ ansiuser@AnsibleController
The key's randomart image is:
+---[RSA 3072]----+
|   |
| o . |
| o . . |
| o . o.. |
| o.Soo. . |
| ooo@ ... |
| ..B # +o.+ |
| .. O X.B* |
| .. o.O==. |
+---[SHA256]----+
ansiuser@AnsibleController:~$
```

=====

CREATE WORKER NODES on AWS LAB

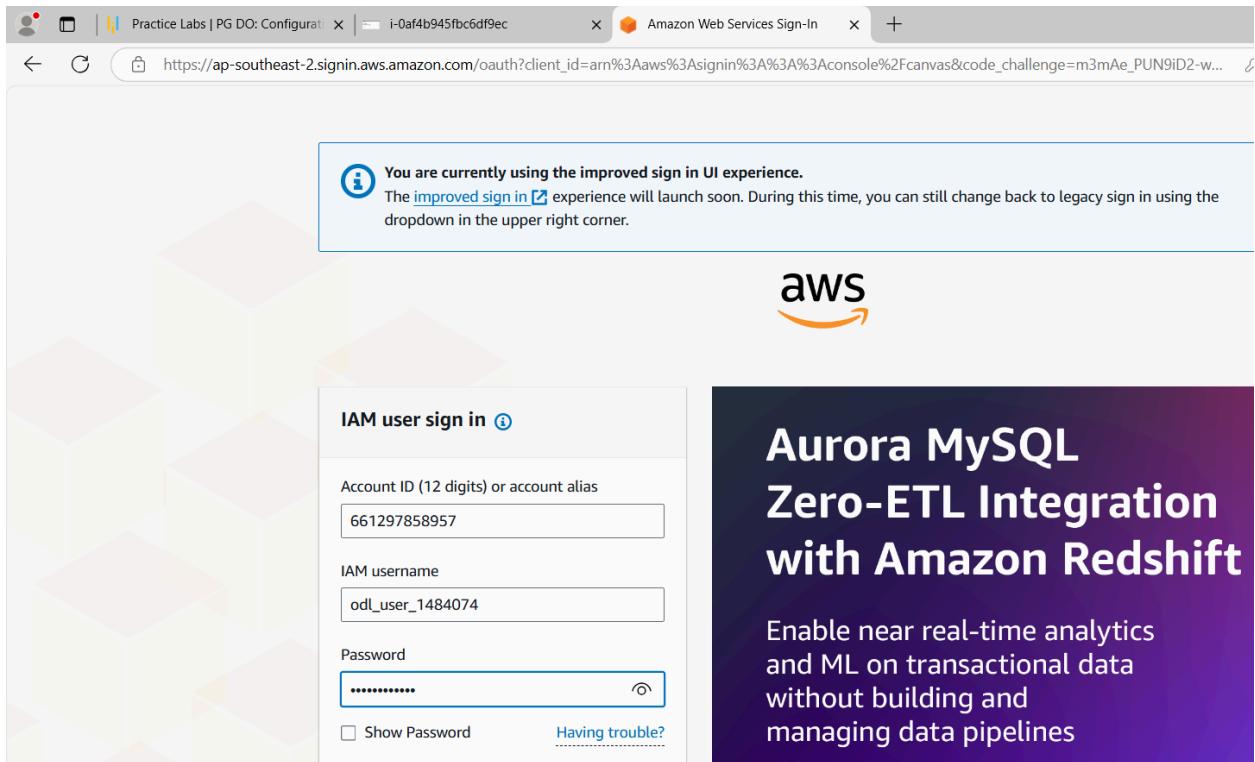
The screenshot shows a web browser window with the URL <https://ims.simplilearn.com/courses/158/PG-DO:-Configuration-management-with-Ansible-and-Terraform/practice-labs>. The page title is "PG DO: Configuration Management with Ansible and Terraform". Below the title, a progress bar indicates "Classes completed: 0 | 1% of Self-Learning Completed | Projects completed: 0/2". On the left, there is a sidebar with icons for "Learning Track" and "Certificate". The main content area shows two tabs: "AWS CloudLabs_Masters" (highlighted with a red circle and arrow) and "DevOps Lab (New)". Under the "AWS CloudLabs_Masters" tab, there is a section titled "DevopsELK". At the bottom, there is a "Web Desktop." button and a "Applications" menu. The "DevOps Lab (New)" tab is currently active.

The screenshot shows a web-based interface for AWS CloudLabs. At the top, it displays "PG DO: Configuration Management with Ansible and Terraform" and "Classes completed: 0 | 1% of Self-Learning Completed | Projects completed: 0/2". Below this, the title "AWS CloudLabs_Masters" and "DevOps Lab (New)" are visible. A sidebar on the left includes icons for Learning track, AWS Lambda, and Certificate. The main content area has tabs for "Environment" and "Resources", with "AWS Credentials" currently selected. A button for "Access key details" is also present. The text "Here are your credentials to login to [Amazon Web Services](#) and access the On Demand Lab." is displayed. A table below lists the credentials:

Auth Fields	Value	Action
Sign-in link	https://661297858957.signin.aws.amazon.com/console/	<input type="button" value="Copy"/>
Username	odl_user_1484074	<input type="button" value="Copy"/>
Password	mvsw74CKP*Yi	<input type="button" value="Copy"/>

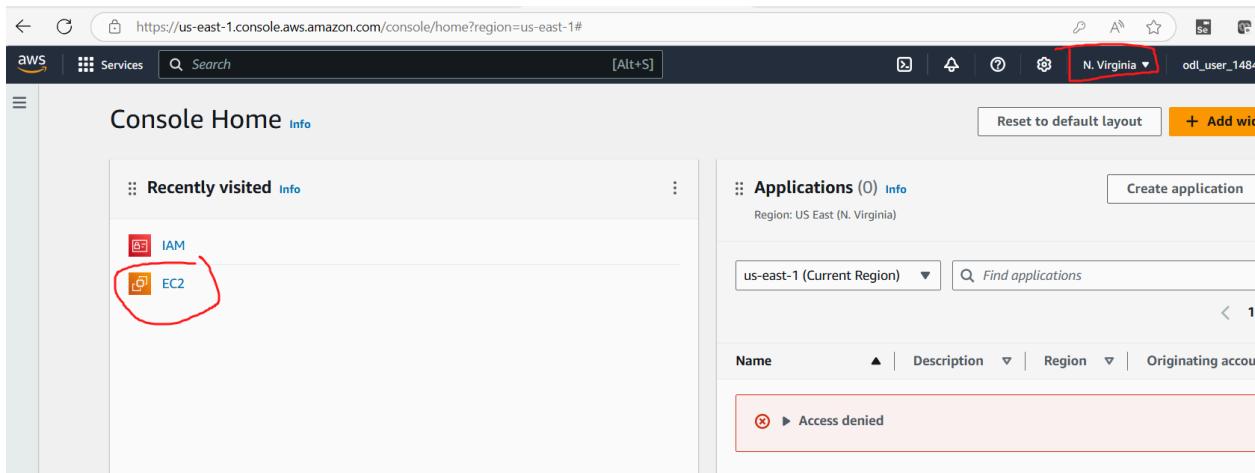
Copy the sign-in link and paste on a new windows tab

Login with given username and password



Click on sign -in . We will be on AWS dashboard

Create an Ec2 server:



The screenshot shows the AWS EC2 Home page. On the left, a sidebar menu includes EC2 Dashboard, EC2 Global View, Events, Instances (with sub-options like Instances, Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Capacity Reservations), Images (AMIs, AMI Catalog), and Elastic Block Store (Volumes). The main area displays 'Resources' with counts for Instances (running) 0, Auto Scaling Groups 0, Capacity Reservations 0 (with an API Error), Dedicated Hosts 0 (with an API Error), Elastic IPs 0, Instances 0, Key pairs 0, Load balancers 0, Placement groups 0, Security groups 1, Snapshots 0, and Volumes 0. Below this is a 'Launch instance' section with a large orange 'Launch instance' button and a 'Migrate a server' link. To the right is a 'Service health' section showing an error message: 'An error occurred' - An error occurred retrieving service health information. A red circle highlights the 'Launch instance' button.

Make sure AWS region is us-east-1

The screenshot shows the 'Launch instances' page for the 'Ubuntu' AMI. The 'Quick Start' section features icons for Amazon Linux, macOS, Ubuntu, Windows, Red Hat, and SUSE Linux. A red circle highlights the 'Ubuntu' icon. Below it, a box highlights the 'Amazon Machine Image (AMI)' section for 'Ubuntu Server 22.04 LTS (HVM), SSD Volume Type'. The description states: 'Ubuntu Server 22.04 LTS (HVM), EBS General Purpose (SSD) Volume Type. Support available from Canonical (<http://www.ubuntu.com/cloud/services>).'. The 'Free tier eligible' status is shown. The 'Summary' sidebar on the right lists: Number of instances (1), Software Image (AMI) Canonical, Ubuntu, 22.04 LTS, ... (with a 'read' link), Virtual server type (instance type) t2.micro, Firewall (security group) New security group, Storage (volumes) 1 volume(s) - 8 GiB, and a note about Free tier: In your first year.

▼ Instance type [Info](#) | [Get advice](#)

Instance type

t2.micro	Free tier eligible
Family: t2 1 vCPU 1 GiB Memory Current generation: true	
On-Demand Windows base pricing: 0.0162 USD per Hour	
On-Demand SUSE base pricing: 0.0116 USD per Hour	
On-Demand RHEL base pricing: 0.026 USD per Hour	
On-Demand Linux base pricing: 0.0116 USD per Hour	

All generations

[Compare instance types](#)

Additional costs apply for AMIs with pre-installed software

Number of instances: 1

Software: Canonical, ami-005fc0

Virtual service: t2.micro

[Alt+S]

Create key pair

Key pair name
Key pairs allow you to connect to your instance securely.
 The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type
 RSA RSA encrypted private and public key pair ED25519 ED25519 encrypted private and public key pair

Private key file format
 .pem For use with OpenSSH .ppk For use with PuTTY

⚠ When prompted, store the private key in a secure and accessible location on your computer. You will need it later to connect to your instance. [Learn more](#)

Cancel **Create key pair**

▼ Network settings [Info](#)

Network [Info](#)
vpc-06633e57dd05f2a94

Subnet [Info](#)
No preference (Default subnet in any availability zone)

Scroll down: you will see a button for ADD security group rule.

Description - required [Info](#)
launch-wizard-1 created 2024-10-12T16:37:38.174Z

Inbound Security Group Rules

▼ Security group rule 1 (TCP, 22, 0.0.0.0/0) [Remove](#)

Type Info	Protocol Info	Port range Info
ssh	TCP	22

Source type [Info](#)
Anywhere

Source [Info](#)
Add CIDR, prefix list or security
0.0.0.0/0 [X](#)

Description - optional [Info](#)
e.g. SSH for admin desktop

⚠ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only. [X](#)

Add security group rule

Click on launch instance.

The screenshot shows the AWS EC2 Instances page. The left sidebar includes links for EC2 Dashboard, EC2 Global View, Events, Instances (selected), Images (AMIs, AMI Catalog), Elastic Block Store (Volumes, Snapshots, Lifecycle Manager), and Network & Security. The main content area has a title 'Instances (1/1) Info' with a search bar and filters for 'All states'. A table lists one instance: 'AnsibleHost' (Instance ID: i-011a404dcf1b82f22, State: Running, Type: t2.micro, Status: Initializing). Buttons for Actions and Launch instances are at the top right.

Connect to the server

The screenshot shows the AWS EC2 Instances page. A red circle highlights the 'Connect' button in the top right corner of the main content area. Another red circle highlights the checkbox next to the instance name 'AnsibleHost' in the list.

The screenshot shows the 'Connect to instance' details page for the instance 'i-011a404dcf1b82f22'. A red box highlights the 'SSH client' tab. A red arrow points from the 'SSH client' tab to the terminal command example below.

Connect to your instance i-011a404dcf1b82f22 (AnsibleHost) using any of these options

EC2 Instance Connect | Session Manager | **SSH client** | EC2 serial console

Instance ID
 i-011a404dcf1b82f22 (AnsibleHost)

1. Open an SSH client.
2. Locate your private key file. The key used to launch this instance is 12Oct24.pem
3. Run this command, if necessary, to ensure your key is not publicly viewable.
 chmod 400 "12Oct24.pem"
4. Connect to your instance using its Public DNS:
 ec2-54-158-236-220.compute-1.amazonaws.com

Example:
 ssh -i "12Oct24.pem" ubuntu@ec2-54-158-236-220.compute-1.amazonaws.com

Note: In most cases, the guessed username is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI username.

Open command prompt on windows or terminal on Mac

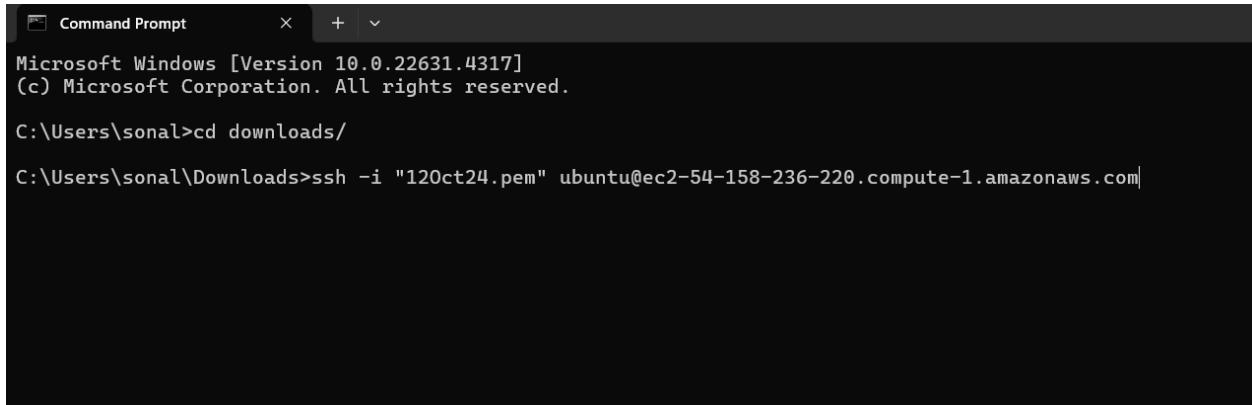
Execute below commands:

```
# cd downloads/
```

```
# Paste the AWS ssh command
```

Example:

```
ssh -i "12Oct24.pem" ubuntu@ec2-54-158-236-220.compute-1.amazonaws.com
```



A screenshot of a Microsoft Windows Command Prompt window. The title bar says "Command Prompt". The window shows the following text:

```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\sonal>cd downloads/
C:\Users\sonal\Downloads>ssh -i "12Oct24.pem" ubuntu@ec2-54-158-236-220.compute-1.amazonaws.com
```

Steps to be executed on Worker Node:

```
# sudo su -
```

Create a new user on the terminal:

```
# adduser ansiuser
```

Enter New password : **ansiuser**

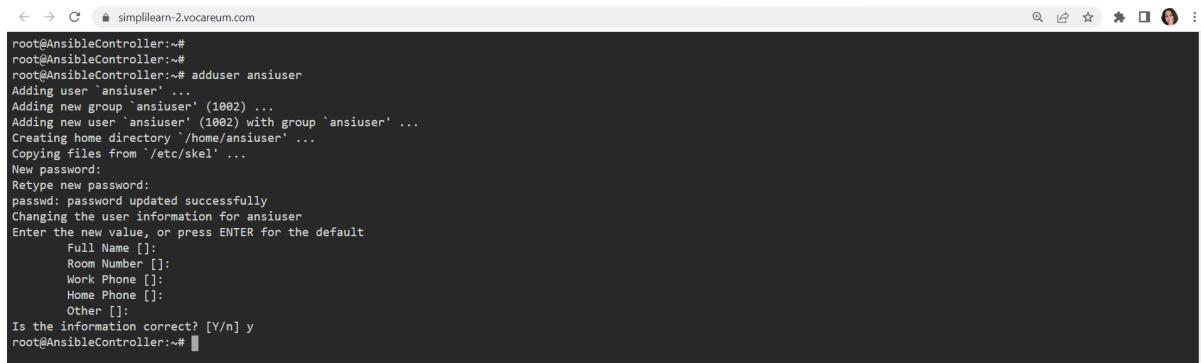
Retype new Password: **ansiuser**

Don't enter any value for fullname, room number, workphone, homephone other

Just keep pressing enter key.

And give Y for

Is the information correct ? [Y/n] : y



The screenshot shows a terminal window with the following text:

```
root@AnsibleController:~#
root@AnsibleController:~# adduser ansiuser
Adding user 'ansiuser' ...
Adding new group 'ansiuser' (1002) ...
Adding new user 'ansiuser' (1002) with group 'ansiuser' ...
Creating home directory '/home/ansiuser' ...
Copying files from '/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for ansiuser
Enter the new value, or press ENTER for the default
    Full Name []:
    Room Number []:
    Work Phone []:
    Home Phone []:
    Other []
Is the information correct? [Y/n] y
root@AnsibleController:~#
```

User will now be created.

Step 2:

We will give sudo permission to this user so that it can execute linux commands without need of password

Add ansiuser in sudoers files and give all permission

vim /etc/sudoers

Press i

Scroll down until you find : # User privilege specification

Now enter below line under %root ALL=(ALL:ALL) ALL

ansiuser ALL=NOPASSWD: ALL

```
# Ditto for GPG agent
Defaults:%sudo env_keep += "GPG_AGENT_INFO"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL
ansiuser ALL=NOPASSWD: ALL
# Members of the admin group may gain root privileges
%admin  ALL=(ALL) ALL

# Allow members of group sudo to execute any command
```

Save the file (:wq!)

Now switch to ansiuser on worker node and create .ssh folder

su - ansiuser

mkdir .ssh

ls -al

```

root@ip-172-31-44-99:~#
root@ip-172-31-44-99:~# su - ansiuser
ansiuser@ip-172-31-44-99:~$ ls -al
total 20
drwxr-x--- 2 ansiuser ansiuser 4096 Oct 12 16:56 .
drwxr-xr-x 4 root      root     4096 Oct 12 16:56 ..
-rw-r--r-- 1 ansiuser ansiuser  220 Oct 12 16:56 .bash_logout
-rw-r--r-- 1 ansiuser ansiuser 3771 Oct 12 16:56 .bashrc
-rw-r--r-- 1 ansiuser ansiuser  807 Oct 12 16:56 .profile
ansiuser@ip-172-31-44-99:~$ mkdir .ssh
ansiuser@ip-172-31-44-99:~$ ls -al
total 24
drwxr-x--- 3 ansiuser ansiuser 4096 Oct 12 16:59 .
drwxr-xr-x 4 root      root     4096 Oct 12 16:56 ..
-rw-r--r-- 1 ansiuser ansiuser  220 Oct 12 16:56 .bash_logout
-rw-r--r-- 1 ansiuser ansiuser 3771 Oct 12 16:56 .bashrc
-rw-r--r-- 1 ansiuser ansiuser  807 Oct 12 16:56 .profile
drwxrwxr-x 2 ansiuser ansiuser 4096 Oct 12 16:59 .ssh
ansiuser@ip-172-31-44-99:~$ |

```

=====

Go to master node(devops lab) and copy the ssh public key of ansiuser

cat /home/ansiuser/.ssh/id_rsa.pub

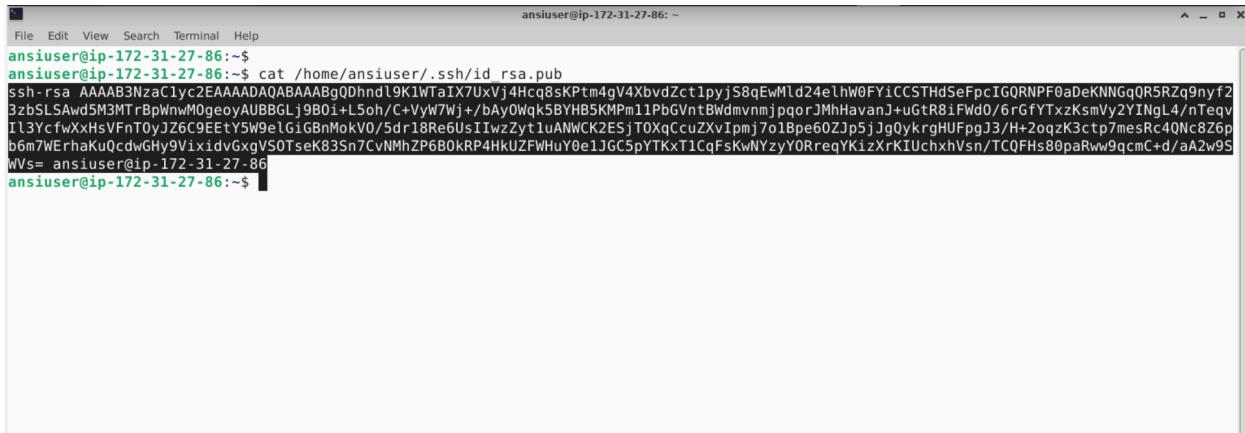
```

Applications: ansiuser@ip-172-31-27-...
ansiuser@ip-172-31-27-86: ~
File Edit View Search Terminal Help
ansiuser@ip-172-31-27-86:~$ cat /home/ansiuser/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAQABAAAQBgQDhndl9K1wTaIX7UxVj4Hcp8skPtm4gV4XbvdZct1pyjS8qEwMld24elhW0FYiCCSTHdSeFpcIGQRNPf0aDeKNNGq0R5RZq9nyf2
3zbSLSAwd5M3MTrBpWnwM0geoyAU8GLj980i+L5oh/C+VyW7Wj+/bAy0Wqk5BYHBSKMPm1lPgVnt8WdmvnmjpqorJMHavanJ+uGtR8iFwd0/6rGfyTxzKsmVy2YIngL4/nTeqv
I13YcfwXsVFnT0yJ26C9EEtY5W9elG1GBnMokVO/5dr18Re6UsI1wzzytluANWCK2EsjTOXqCcuZXvIpmpj7o1Bpe60Zjp5jJgQykrgHUFpgJ3/H+2oqzK3ctp/mesRc4QNc8Z6p
b6m7WEraKuQcdwGhy9VixidvGxgvSOtseK83Sn7CvNmhzP680kRP4HKuZFwHuY0e1JGC5pYTKit1CqFsKwNYzy0RreqYKizXrKIUchxhVsn/TCQFHs80paRww9qcmC+d/aA2w9S
WVs= ansiuser@ip-172-31-27-86
ansiuser@ip-172-31-27-86:~$

```

Right click and copy the copy

Carefully copy the key

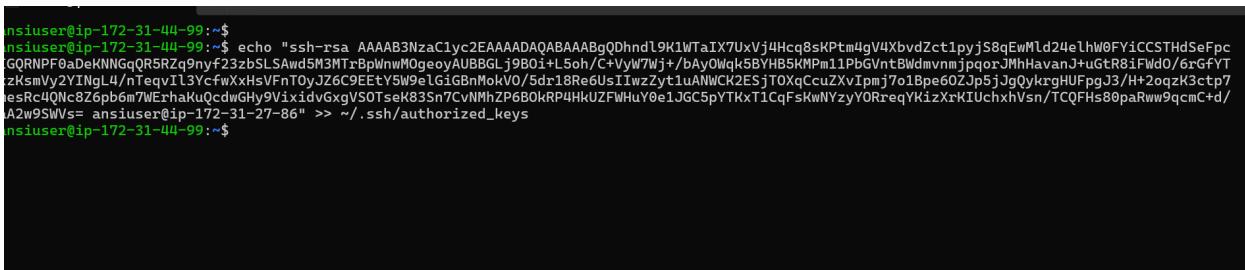


A screenshot of a terminal window titled "ansiuser@ip-172-31-27-86: ~". The window contains the following text:

```
File Edit View Search Terminal Help
ansiuser@ip-172-31-27-86:~$ cat /home/ansiuser/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQgQDhndl9K1WTaIX7UxVj4Hcq8sKPt4gV4XbcdZct1pyjS8qEwMld24elhW0FYiCCSThdSeFpcIGQRNPFOaDeKNNGqQR5RZq9nyf2
3zbLSAw5M3MTrBpWnwM0geoyAUBBGLj980i+L5oh/C+vYw7Wj+/bAy0Wqk5BYHB5KMPm11PbGVntBwdmvnmjpqrJMHavanJ+Ugtr81Fwd0/6rGfYT
I13YcfwXxHsVFnT0yJZ6C9EEtY5W9e1gGBnMokVO/5dr18Re6UsIIiwzZyt1uANWCK2EsjTOxqCcuZxvIpmpj7o1Bpe60ZJp5jJgQykrghUFpgJ3/H+2oqzK3ctp7mesRc4Qnc8Z6p
b6m7WEraKuQcdwGhy9VixidvGxgVSOTsek83Sn7CvNMhZP6B0kRP4HKUZFWhuY0e1JGC5pYTKxT1CqFsKwNYzyYORreqYKizXrKIUchxhVsn/TCQFHs80paRww9qcmC+d/aA2w9S
WVs= ansiuser@ip-172-31-27-86:~$
```

Go to Worker node (AWS VM)

```
# echo "<give your public key>" >> ~/.ssh/authorized_keys
```



A screenshot of a terminal window titled "ansiuser@ip-172-31-44-99:~\$". The window contains the following text:

```
ansiuser@ip-172-31-44-99:~$ echo "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQgQDhndl9K1WTaIX7UxVj4Hcq8sKPt4gV4XbcdZct1pyjS8qEwMld24elhW0FYiCCSThdSeFpcIGQRNPFOaDeKNNGqQR5RZq9nyf2
3zbLSAw5M3MTrBpWnwM0geoyAUBBGLj980i+L5oh/C+vYw7Wj+/bAy0Wqk5BYHB5KMPm11PbGVntBwdmvnmjpqrJMHavanJ+Ugtr81Fwd0/6rGfYT
I13YcfwXxHsVFnT0yJZ6C9EEtY5W9e1gGBnMokVO/5dr18Re6UsIIiwzZyt1uANWCK2EsjTOxqCcuZxvIpmpj7o1Bpe60ZJp5jJgQykrghUFpgJ3/H+2oqzK3ctp7mesRc4Qnc8Z6p
b6m7WEraKuQcdwGhy9VixidvGxgVSOTsek83Sn7CvNMhZP6B0kRP4HKUZFWhuY0e1JGC5pYTKxT1CqFsKwNYzyYORreqYKizXrKIUchxhVsn/TCQFHs80paRww9qcmC+d/aA2w9S
WVs= ansiuser@ip-172-31-44-99:~$
```

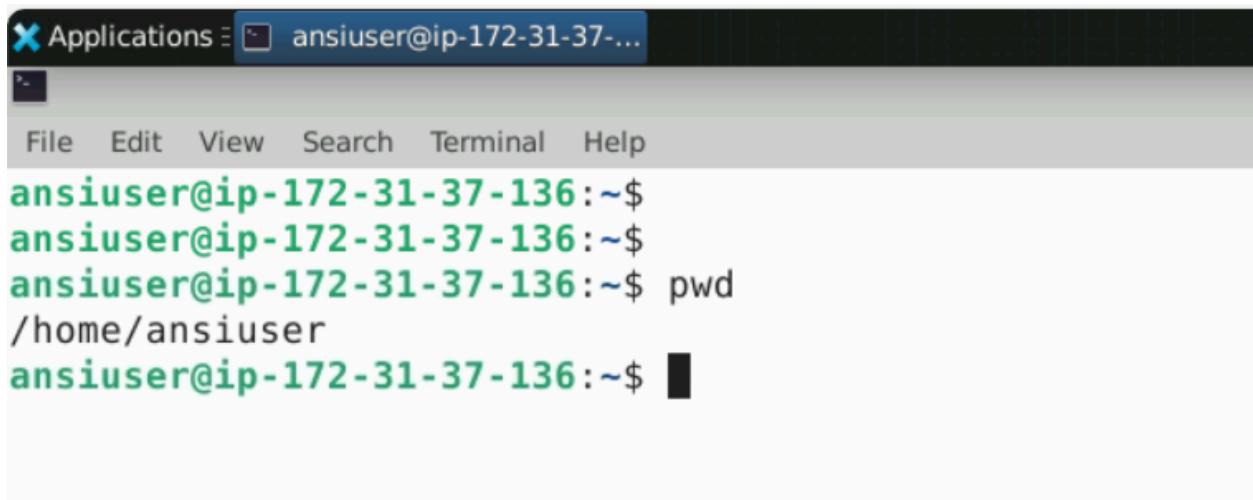
Great job, the setup is complete

```
# su - ansiuser
```

```
# cd
```

```
# pwd
```

You should be in the ansiuser home directory



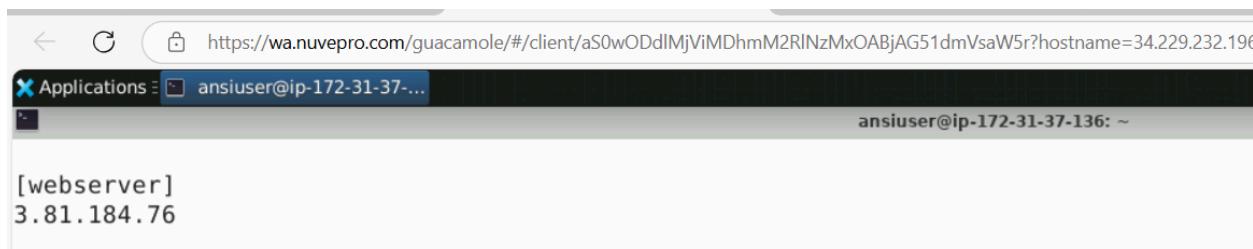
A screenshot of a terminal window titled "Applications" with the session identifier "ansiuser@ip-172-31-37-...". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows the user's session:

```
ansiuser@ip-172-31-37-136:~$  
ansiuser@ip-172-31-37-136:~$  
ansiuser@ip-172-31-37-136:~$ pwd  
/home/ansiuser  
ansiuser@ip-172-31-37-136:~$
```

```
# vim myinventory
```

[webserver]

Public ipaddress of aws VM



A screenshot of a terminal window titled "Applications" with the session identifier "ansiuser@ip-172-31-37-...". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows the user's session and the output of the command "curl https://checkip.amazonaws.com":

```
< https://wa.nuvepro.com/guacamole/#/client/aS0wODdlMjViMDhmM2RINzMxOABjAG51dmVsaW5r?hostname=34.229.232.196  
ansiuser@ip-172-31-37-136: ~  
[webserver]  
3.81.184.76
```

```
ansiuser@ip-172-31-37-136:~$  
ansiuser@ip-172-31-37-136:~$  
ansiuser@ip-172-31-37-136:~$ pwd  
/home/ansiuser  
ansiuser@ip-172-31-37-136:~$ vim myinventory  
ansiuser@ip-172-31-37-136:~$ ls  
myinventory  
ansiuser@ip-172-31-37-136:~$
```

Execute the below command to use the created inventory file for running module

ansible -i /home/ansiuser/myinventory webserver -m ping

We will create our own ansible.cfg file now to set myinventory as the default inventory file for ansiuser.

vim ansible.cfg

[defaults]

inventory = /home/ansiuser/myinventory
interpreter_python = auto_silent

Save the file.

```
# ansible webserver -m ping
```

```
=====
```

AnsibleCode is written in 2 ways:

```
=====
```

> Ad Hoc commands

Ad hoc commands are single line commands

Using ad hoc command we execute a single module on the worker nodes

Whenever we have to quickly check something on the worker node we execute an ad hoc command

When wever have to validate an output on worker nodes, we will use adhoc commands.

Syntax:

```
# ansible <hostgroupName> -m <moduleName> -a  
“par1=value par2=value”
```

Demo:

```
# ansible webserver -m command -a "uptime"
```

```
# ansible webserver -m command -a "df -h"
```

```
# ANSIBLE_KEEP_REMOTE_FILES=1 ansible all -m shell -a  
"uptime"
```

Create a directory:

```
# ansible webserver -m file -a "path=/tmp/mydir  
state=directory"
```

Validate:

```
# ansible webserver -m command -a "ls /tmp"
```

Create a file in the directory

```
# ansible webserver -m file -a "path=/tmp/mydir/file1  
state=touch"
```

Validate:

```
# ansible webserver -m command -a "ls /tmp/mydir"
```

Add content to the file

```
# ansible webserver -m copy -a 'content="Hello form Ansible  
controller" dest=/tmp/mydir/file1'
```

Validate it:

```
# ansible webserver -m command -a "cat /tmp/mydir/file1"
```

Agenda - Day 2 - 25 May 2025

- > Understand playbook structure**
 - > Write playbooks to execute multiple tasks**
 - > Variables in Ansible**
 - > Fact variables**
 - > Conditions and loops in Ansible**
-

A playbook is -> a set of plays

A play consist of :

- host server details (which servers we have to do the changes)
- tasks (which modules have to be executed)

A playbook is written in YAML, its extension will be .yml or .yaml

Recap of YAML

In a YAML file we write the code in format

key: value

A key can store a single value, list of value and map

A YAML file will start with ---

In playbook, the key is given by ansible and value is provided by user

Example 1: A key storing single value

```
name: play1
hosts: webserver
package: git
file: file1
```

Example 2: A key storing a list of values

```
package:
  - git
  - mysql
  - php
  - wordpress
```

- tree
- wget

Example 3: A key storing again a key and value

tasks:

- name: Task1
 command: echo "command1"
- name: task2
 package: name=tree

workflow: name

push:

job1:

 runs-on: ubuntu

 steps:

- name:
 uses:
- name:
 runs:

Various sections of playbook:

=====

- name: Give name to the play
 hosts: give the inventory host group
 tasks:
 - name: give a name to the tasks
 module_name: par1=value par2=value

=====

Demo 1: Write a playbook that will use a debug module to print statements

vim playbookDebug.yml

- name: Print a message on Console

```
hosts: webserver
tasks:
- name: Print a welcome message
  debug: msg="Hello form Ansible Controller"

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

# ansible-playbook playbookDebug.yml
```

Register Variables:

Variables : are temporary locations where data is stored.

Whenever a module is executed and you want to store its output we will use ansible Register variable

Register variable will store output of the module that was written just above the register keyword

```
# vim playbookRegister.yml
```

```
- name: Register Variables in ansible
hosts: webserver
tasks:
- name: Print a message
  debug: msg="Run a command"
- name: Execute command on the host server
```

```
command: hostname -s
register: command_output
- name: print the register variable value
  debug: var=command_output.stdout
```

Save the file and execute

```
# ansible-playbook playbookRegister.yml
```

```
- name: Debug module in ansible
  hosts: webserver
  tasks:
    - name: Print greetings on the console
      debug: msg="Hello form Ansible controller!"
    - name: Run a Linux command
      command: hostname -s
      register: cmd_output
    - name: Print the register variable value
      debug: var=cmd_output.stdout
```

~
~
~

```
=====
```

Execute a playbook to install package and create a file

```
# vim playbook2.yml
```

```
- name: Install packages on the worker node
```

```
hosts: webserver
become: true
become_user: root
tasks:
- name: update the apt repo
  command: apt-get update
- name: install php on hostserver
  package: name=php state=present
- name: Create a file on hostserver
  file: path=/tmp/myfile state=touch
```

Save the file(:wq!)

```
# ansible-playbook playbook2.yml
```

=====

Variables in Playbooks:

=====

Variables -> temporary memory locations that store data

To make our playbooks reusable use variables
using variables, we will be able to pass new data to the parameters in
the tasks sections

A variable can store a single value or a list of values
variables can store a string, number, boolean value
Variables are 2 types in ansible - Custom and fact variables

Custom variables:

=====

- Variables created by users
- variable name given by user
- variable value given by user
- These variables can be declared in a playbook or in a separate file or in a inventory file also
- The variable values can be referred in the tasks section as {{ var_name }}

```
# vim playbookvariables.yml
```

```
- name: Custom Variables in Ansible
  hosts: webserver
  become: true
  become_user: root
  vars:
    pkg_name: git
    pkg_state: present
    file_path: /tmp/mydemo
    file_state: directory
  tasks:
    - name: update the apt repo
      command: apt-get update
    - name: install {{ pkg_name }} on hostserver
      package: name={{ pkg_name }} state={{ pkg_state }}
    - name: Create a directory on hostserver
      file: path={{ file_path }} state={{ file_state }}
```

Save the playbook and run it.

=====

Store the variables in a separate file

```
# vim variables.yml
```

```
pkg_name: git
pkg_state: present
file_path: /tmp/mydemo
file_state: directory
```

Save the file (:wq!)

```
# vim playbook4.yml
```

```
- name: Custom Variables in Ansible
  hosts: webserver
  become: true
  become_user: root
  # vars: local variables-
  vars_files:
    - variables.yml
  tasks:
    - name: update the apt repo
      command: apt-get update
    - name: install {{ pkg_name }} on hostserver
      package: name={{ pkg_name }} state={{ pkg_state }}
    - name: Create a directory on hostserver
```

```
file: path={{ file_path }} state={{ file_state }}
```

Save the file and run the playbook

```
=====
```

Run the playbook by passing values to the variables at runtime.
Use the option : --extra-vars

```
# ansible-playbook playbook4.yml --extra-vars "pkg_name=maven  
pkg_state=absent file_path=/tmp/newdemo file_state=touch"
```

```
=====
```

FACT Variables:

```
=====
```

**Whenever ansible controller connects to its worker nodes
Always Ansible will gather complete information or details
about each worker node**

**It will gather information like host details,
architecture, network, ipaddress, BIOS
information, OS, memory etc etc**

This gathered information is called as FACTS

**Ansible stores each fact in a variable called as fact variables.
These variables are created by Ansible and values are also
computed by Ansible**

**These variables will always start with
“ansible_variableName”**

These variable values are computed at runtime

Some of these variable values are unique to each host server

These variables are also called dynamic variables.

The value of the variable changes from host to host

In ansible we have a module called as setup module.

Demo 1: See the fact variables:

=====

```
# ansible webserver -m setup
# ansible webserver -m setup -a "filter=ansible_hostname"
# ansible webserver -m setup -a "filter=ansible_os_family"
# ansible webserver -m setup -a "filter=ansible_distribution"
# ansible webserver -m setup -a "filter=ansible_dist"
```

```
# vim playbook5.yml
```

```
- name: Fact variables in Ansible
  hosts: webserver
  become: true
  tasks:
    - name: install httpd package
      package: name=httpd state=present
      when: ansible_os_family == "RHEL"
    - name: install apache2 package
```

```
package: name=apache2 state=present
when: ansible_os_family == "Debian"
```

Save the file and run it

Example 2: When keyword

```
=====
```

```
# vim playbookWhen.yml
```

```
- name: When condition in playbook
hosts: webserver
become: true
tasks:
- name: install package on Debain machines
  package: name=apache2 state=present
  when: ansible_distribution == "Ubuntu" and
ansible_distribution_major_version == "24"
- name: Execute a command
  command: hostname -s
  when: (ansible_distribution == "Ubuntu") or
        (ansible_distribution == "Amazon") or
        (ansible_distribution == "RHEL")
```

Save the file and execute the file.

```
=====
```

Tags in Ansible Playbook:

Tags allows us to run specific tasks in the playbook
If we want set of tasks to be executed together then we can use tags

```
# vim playbookTags.yml
```

```
- name: Tags in ansible
  hosts: webserver
  become: true
  tasks:
    - name: execute a command
      command: echo "Hello All"
      tags: cmd
    - name: execute a command to update apt
      command: apt-get update
      tags: cmd
    - name: Install a package
      package: name=tree state=present
      tags: install
    - name: Uninstall tree package
      package: name=tree state=absent
```

```
ansible-playbook playbookTags.yml --tags untagged
```

```
# ansible-playbook playbookTags.yml --tags install
```

```
# ansible-playbook playbookTags.yml --tags cmd
```

```
# ansible-playbook playbookTags.yml --tags cmd,install  
  
# ansible-playbook playbookTags.yml --tags untagged  
  
# ansible-playbook playbookTags.yml --tags tagged
```

Day 3: 31-May-2025

Agenda:

- > Jinja2 templates
 - > Handlers
 - > Ansible Roles
 - > Dynamic Inventory
-

Problem statement:

We have to always copy config files on each server

These config files will have:

- > some common content/code/parameters
 - > some content which is unique to each host server
- So it is not a good practice to manually create unique files for each host server

Solution:

Use Templates in Ansible:

A file with text and data

This data is computed by ansible for every worker node
Ansible will update host server specific data in the template file
Ansible will then copy that unique file to the hosts servers

Ansible will perform this task using Jinja2 templates

=====

Jinja2 is a templating engine present in Ansible

It comes preinstalled in ansible controller

There is no jinja2 on worker nodes

Templating happens only on the controller machine.

A jinja2 template is a file with extension .j2

The jinja2 file consist of :

- plain text
- FACT variables
- Custom Static Variable

When we execute the template Ansible will:

- > keep the text as it is
- > it will replace the static variables with their values as given by user
- > it will replace fact variables with values unique to each host

This way unique config file will be copied on each host server.

=====

Demo:

=====

```
# vim file.config.j2
```

This is my application configuration file

This file is created by {{ author }}

This file is present in directory /etc

```
hostname = {{ ansible_nodename }}
```

```
Ip address = {{ ansible_default_ipv4["address"] }}
```

```
OS = {{ ansible_os_family }}
```

For any concerns send an email to {{ email }}

Save the file

```
# vim playbookJinja2.yml
```

```
- name: Jinja2 templates
  hosts: webserver
  become: true
  vars:
    author: admin
    email: admin@gmail.com
  tasks:
    - name: copy jinja2 file on the host
      template: src=file.config.j2 dest=/etc/file.config
```

Run the playbook

Validate:

```
# ansible webserver -m command -a "cat /etc/file.config"
```

Demo2:

```
# vim myinventory
```

```
[webserver]
18.233.160.220 http_port=80
```

```
# vim ports.conf.j2
```

```
# If you just change the port or add more ports here, you will
likely also
# have to change the VirtualHost statement in
# /etc/apache2/sites-enabled/000-default.conf
```

```
Listen {{ http_port }}
```

```
<IfModule ssl_module>
    Listen 443
</IfModule>
```

```
<IfModule mod_gnutls.c>
    Listen 443
</IfModule>
```

```
# vim: syntax=apache ts=4 sw=4 sts=4 sr noet
```

Save the above file.

```
# vim 000-default.conf.j2
<VirtualHost *:{{http_port}}>
    # The ServerName directive sets the request scheme,
    # the server uses to identify itself. This is used when
    # creating
    # redirection URLs. In the context of virtual hosts, the
    # ServerName
    # specifies what hostname must appear in the request's
    # Host: header to
    # match this virtual host. For the default virtual host (this
    # file) this
    # value is not decisive as it is used as a last resort host
    # regardless.
    # However, you must set it for any further virtual host
    # explicitly.
    #ServerName www.example.com

    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html

    # Available loglevels: trace8, ..., trace1, debug, info, notice,
    # warn,
    # error, crit, alert, emerg.
    # It is also possible to configure the loglevel for particular
    # modules, e.g.
    #LogLevel info ssl:warn
```

```
ErrorLog ${APACHE_LOG_DIR}/error.log  
CustomLog ${APACHE_LOG_DIR}/access.log combined
```

```
# For most configuration files from conf-available/, which are  
# enabled or disabled at a global level, it is possible to  
# include a line for only one particular virtual host. For  
example the  
# following line enables the CGI configuration for this host  
only  
# after it has been globally disabled with "a2disconf".  
#Include conf-available/serve-cgi-bin.conf  
</VirtualHost>
```

Save the file

```
# vim playbookapache.yml
```

```
- name: Update config files of apache2 server  
hosts: webserver  
become: true  
tasks:  
- name: Update the repo  
  command: apt-get update  
- name: Install apache2  
  package: name=apache2 state=present  
- name: Copy the update apache2 ports.conf file  
  template: src=ports.conf.j2 dest=/etc/apache2/ports.conf  
- name: Copy the updated 000-defualt.conf file
```

```
template: src=000-default.conf.j2
dest=/etc/apache2/sites-enabled/000-default.conf
- name: Restart apache2 service
  service: name=apache2 state=restarted
```

Save the file and execute

Validate:

```
# ansible webserver -m command -a "cat /etc/apache2/ports.conf"
# ansible webserver -m command -a " cat /etc/apache2/sites-enabled/000-default.conf"
```

HANDLERS:

Handlers:

**When we have a set of tasks to be executed only when required
then use handlers**

**Handlers are executed only when notified by the parent tasks -
notify keyword**

**Handler tasks will be notified only and only when the status of
the parent task is changed**

**IF the parent task status is Ok - ansible will not notify the handler
to run**

**Handlers tasks are execute after all the parent task have been
executed**

As the task is executed - tasks status is changed --notify the handler to run- handler executed at this point

We make use of concept of flush_handlers

Using --force-handlers at runtime - notified handlers will be executed even if playbook has failed Tasks

Observations:

=====

- 1. Tasks under the handler section of the playbook are not executed by themselves**
- 2. Even though we have added notify keyword under the parent tasks to notify handlers, however the handlers were not executed**
- 3. Handler tasks though notified multiple times during the play but they will execute only once at the end.**
- 4. Using meta and flush handlers task are executed immediately as they are notified**
- 5. If any task fails in the playbook -> all handlers will not execute even though they have been notified by previous successful task.**

Using --force-handlers at runtime - notified handlers will be executed even if playbook has failed Tasks

```
# vim playbookHandlers.yml
```

```
- name: handlers in Ansible
  hosts: webserver
  become: true
```

tasks:

- name: To run a command
 command: echo "Trigger a command"
 notify: Run a command
- name: Start a service
 command: echo "start service"
 notify: Restart apache2 service
- name: Create a file
 command: echo "File created"
 notify: File Create

handlers:

- name: Run a command
 command: hostname -s
- name: Restart apache2 service
 service: name=apache2 state=restarted
- name: File Create
 file: path=/tmp/newfile state=touch

Save and execute, handlers will run at the last

Demo : Flush the handlers as they are notified

```
# vim playbookhandlers2.yml
```

```
- name: handlers in Ansible  
hosts: webserver  
become: true  
tasks:  
- name: To run a command
```

```
command: echo "Trigger a command"
notify: Run a command
- meta: flush_handlers # all handlers notified till here will get
executed
- name: Start a service
  command: echo "start service"
  notify: Restart apache2 service
- name: Create a file
  command: echo "File created"
  notify: File Create
handlers:
- name: Run a command
  command: hostname -s
- name: Restart apache2 service
  service: name=apache2 state=restarted
- name: File Create
  file: path=/tmp/newfile state=touch
```

Save and run it.

```
# vim playbookHandlers3.yml

- name: handlers in Ansible
hosts: webserver
become: true
tasks:
- name: To run a command
  command: echo "Trigger a command"
  notify: Run a command
- name: Start a service
  command: ech "start service"
```

```
  notify: Restart apache2 service
  ignore_errors: true
- name: Create a file
  command: echo "File created"
  notify: File Create
handlers:
- name: Run a command
  command: hostname -s
- name: Restart apache2 service
  service: name=apache2 state=restarted
- name: File Create
  file: path=/tmp/newfile state=touch
```

```
# ansible-playbook playbookHandlers3.yml --force-handlers
```

If any task fails in the playbook -> all handlers will not execute even though they have been notified by previous successful task.

Using --force-handlers at runtime - notifed handlers will be executed even if playbook has failed Tasks

Ansible Roles:

```
# mkdir roles
```

```
# cd roles
```

```
# ansible-galaxy init apache

# cd apache

# ls

# vim tasks/main.yml

- name: Update the repo
  command: apt-get update
- name: Install apache2
  package: name={{ pkg_name }} state=present
- name: Copy the update apache2 ports.conf file
  template: src=ports.conf.j2 dest={{ dest_ports_path }}
- name: Copy the updated 000-defualt.conf file
  template: src=000-default.conf.j2 dest={{ dest_default_path }}
  notify: Restart apache2 service
```

Save the file

```
ansiuser@ip-172-31-22-122:~$ cat roles/apache/tasks/main.yml
---
# tasks file for apache

- name: Update the repo
  command: apt-get update
- name: Install apache2
  package: name={{ pkg_name }} state=present
- name: Copy the update apache2 ports.conf file
  template: src=ports.conf.j2 dest={{ dest_ports_path }}
- name: Copy the updated 000-defualt.conf file
  template: src=000-default.conf.j2 dest={{ dest_default_path }}
  notify: Restart apache2 service
```

```
ansiuser@ip-172-31-22-122:~$
```

```
# vim vars/main.yml

pkg_name: apache2
dest_ports_path: /etc/apache2/ports.conf
dest_default_path: /etc/apache2/sites-enabled/000-default.conf
http_port: 80
```

Save the file

```
# vim templates/ports.conf.j2
```

```
# If you just change the port or add more ports here, you will
likely also
```

```
# have to change the VirtualHost statement in
# /etc/apache2/sites-enabled/000-default.conf
```

```
Listen {{ http_port }}
```

```
<IfModule ssl_module>
    Listen 443
</IfModule>
```

```
<IfModule mod_gnutls.c>
    Listen 443
</IfModule>
```

```
# vim: syntax=apache ts=4 sw=4 sts=4 sr noet
```

Save the file

```
# vim templates/000-default.conf.j2
```

```
<VirtualHost *:{{http_port}}>
    # The ServerName directive sets the request scheme,
    # the server uses to identify itself. This is used when
    # hostname and port that
    # redirection URLs. In the context of virtual hosts, the
    # specifies what hostname must appear in the request's
    # Host: header to
    # match this virtual host. For the default virtual host (this
    # file) this
    # value is not decisive as it is used as a last resort host
    # regardless.
    # However, you must set it for any further virtual host
    # explicitly.
    #ServerName www.example.com

    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html

    # Available loglevels: trace8, ..., trace1, debug, info, notice,
    # warn,
    # error, crit, alert, emerg.
    # It is also possible to configure the loglevel for particular
    # modules, e.g.
    #LogLevel info ssl:warn

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
```

```
# For most configuration files from conf-available/, which are
# enabled or disabled at a global level, it is possible to
# include a line for only one particular virtual host. For
example the
```

```
# following line enables the CGI configuration for this host
only
```

```
# after it has been globally disabled with "a2disconf".
```

```
#Include conf-available/serve-cgi-bin.conf
```

```
</VirtualHost>
```

Save this file

```
# vim handlers/main.yml
```

```
- name: Restart apache2 service
  service: name={{ pkg_name }} state=restarted
```

Save the file

```
# cd
```

Create a playbook to run the role

```
# vim playbookRoles.yml
```

```
- name: Roles in ansible
  hosts: webserver
  become: true
  roles:
    - apache
```

Save the file and run the playbook.

```
=====
```

Roles Demo 2:

```
=====
```

```
# cd roles
```

```
# ansible-galaxy init servers
```

```
# cd servers
```

```
# vim tasks/main.yml
```

```
- name: Install required software
  package: name={{ item }} state=present
  loop:
    - apache2
    - mysql-server
    - php-mysql
    - php
    - libapache2-mod-php
    - python3-mysqldb
```

Save the file

Create another role

```
# cd  
  
# cd roles  
  
# ansible-galaxy init php  
  
# cd php  
  
# vim tasks/main.yml  
  
- name: Install php extensions  
  package: name={{ item }} state=present  
  loop:  
    - php-gd  
    - php-ssh2
```

Save the file

Now create another role

```
# cd  
  
# cd roles  
  
# ansible-galaxy init mysql  
  
# cd mysql  
  
# vim tasks/main.yml
```

```
- name: Create mysql database
  mysql_db: name={{ wp_mysql_db }} state=present
- name: Create mysql user
  mysql_user:
    name={{ wp_mysql_user }}
    password={{ wp_mysql_password }}
    priv=*.*:ALL
```

Save the file

```
# vim defaults/main.yml
```

```
wp_mysql_db: wordpress
wp_mysql_user: wordpress
wp_mysql_password: wordpress
```

Save the file

Come out of the roles directory

```
# cd
```

```
# vim playbookRoles.yml
```

```
- name: Roles in ansible
  hosts: webserver
  become: true
  roles:
    - apache
    - servers
    - php
    - mysql
```

Save and run the playbook

Day 4: Agenda:

- Ansible Roles
- Ansible Vault
- Ansible dynamic Inventory
- Terraform

Ansible Vault:

1. Ansible playbook cannot store data in an encrypted format
2. So Ansible comes with a utility called as Ansible vault which can be used to store secrets or passwords in an encrypted format
3. Ansible vaults are password protected

- 4. Ansible vaults is tool that will take your plain text and convert it to encrypted data**
- 5. An encrypted data can decrypted using vault**
- 6. The ansible vault will use an algorithm called as AES256 for encryption of data**
- 7. Ansible playbook can refer to the encrypted data in the vault only and only if we provide the vault password while executing the playbook**
- 8. Ansible vault utility is present on the controller machine only.**

Demo:

- 1. Use ansible vault to create a new encrypted file and store it in the vault.**

```
# su - ansiuser
```

```
# ansible-vault create vault1.yml
```

New Vault password: 123

Confirm New Vault password: 123

Now the file will open, press i to insert data

Confidential data!

Save the file (:wq!)

Now see the content of the file

```
# cat vault1.yml
```

Demo 2: Use ansible-vault command to view the encrypted data of the file:

```
# ansible-vault view vault1.yml
```

Vault password: 123

You should see actual data

Demo 3: use ansible vault to encrypt data of an existing file

```
# echo "Text to be encrypted" >> encrypt_me.txt
```

```
# ansible-vault encrypt encrypt_me.txt
```

New Vault password: 123

Confirm New Vault password: 123

Encryption successful

```
# cat encrypt_me.txt
```

Demo 4: Use ansible-vault to decrypt an encrypted file:

```
# ansible-vault decrypt encrypt_me.txt
```

Vault password: 123

Decryption successful

```
# cat encrypt_me.txt
```

Execution of Ansible playbook using vault secrets

=====

```
# ansible-vault create secrets.yml
```

New Vault password: 123

Confirm New Vault password: 123

Press i

Give data as:

```
username: ansible123
```

```
password: ansible@123
```

Save the file

```
# vim playbookSecrets.yml
```

```
- name: use ansible vault to fetch variable values
  hosts: webserver
  become: true
  vars_files:
    - secrets.yml
  tasks:
    - name: Create a user on hostserver
      user: name={{username}} password={{password}}
```

Save the file and run the playbook

```
# ansible-playbook playbookSecrets.yml --ask-vault-pass
```

Ansible Dynamic Inventory

By default in ansible we have a hosts file where we write the list of Ip address of the hosts servers where we have to do the changes.

This file is static where the ip address are fixed, user will manually add new ips or remove Ipaddress that are not required.

But consider a use case where your infrastructure on the cloud and the number of servers are scaling up or scaling down dynamically.

Since the inventory on AWS is dynamic, we cannot hard code the inventory/hosts file on Ansible controller. That will not be correct approach

What is the solution then?

We will take help of Ansible where

- Ansible will connect to AWS securely**
- We will use an ansible plugin that will check number of VMs in the desired region on AWS and fetch the ip address or hostnames on the ansible controller machine.**

- Once ansible collects the ipaddress of the VMs on the cloud it will automatically compute an Inventory file for the user
- We will use a command to generate this inventory file
- Whenever the command is run Ansible → connect to AWS → go to desired region → collects IP of available VMs → displays the inventory file on the controller
- Now whenever the user will run the playbook, ansible will use the dynamic inventory and execute the changes on the dynamic VMs

Steps for 1 st part:

1. First Prepare Ansible Controller to install packages that required to connect to AWS
2. Install the ansible Cloud plugin and update its details in ansible.cfg file
3. Create a new inventory file ->The name of the inventory file should always end with aws_ec2.yml

Steps of Part 2:

1. Create credentials on AWS, so ansible can connect to AWS

2. Create some Ec2 Vms on AWS of which ansible will fetch the inventory details.

Steps of Part3:

- 1. Ansible Controller connects to those EC2 servers using SSH**
- 2. Ansible controller executes playbook on the server**

Note:

The SDK is composed of two key Python packages: Botocore (the library providing the low-level functionality shared between the Python SDK and the AWS CLI) and Boto3 (the package implementing the Python SDK itself).

<https://boto3.amazonaws.com/v1/documentation/api/latest/guide/quickstart.html>

Execute below steps:

**Whichever user you are logged in with the same user
you will install the packages and run the ansible
inventory command**

su - ansiuser

Install ansible aws_ec2 plugins

The plugin is part of the amazon.aws collection

We will install the desired collection

ansible-galaxy collection install amazon.aws

**In order to install boto3 and botocore we need
python3-pip package**

sudo apt install python3-pip

pip3 install boto3

pip show boto3

sudo apt-get update

sudo apt-get install awscli -y

Update the ansible.cfg file for it to use the aws_ec2 plugin to fetch the inventory

vim ansible.cfg

[defaults]

inventory = /home/ansiuser/myinventory

enable_plugins = aws_ec2

Save the file

```
[defaults]
inventory = /home/ansiuser/myinventory
```

```
enable_plugins = aws_ec2
```

```
~  
~  
~  
~
```

Create the aws_ec2 inventory file

```
# vim aws_ec2.yml
```

```
plugin: amazon.aws.aws_ec2
regions:
  - us-east-1
```

```
Save the file
```

```
plugin: amazon.aws.aws_ec2
regions:
  - us-east-1
~
~
~
~
~
~
```

**Go to AWS LAB AND pick up the accesskey
credentials:**

The screenshot shows the AWS CloudLabs DevOps Lab interface. At the top, it displays 'PG DO: Configuration Management with Ansible and Terraform' and 'Classes completed: 0 | 1% of Self-Learning Completed | Projects completed: 0/2'. A 'Join Course' button is visible. Below this, the lab title is 'AWS CloudLabs_Masters' and the sub-project is 'DevOps Lab (New)'. A message says 'This Lab will get reset on 2024-01-15'. On the left, there's a sidebar with 'Learning Track' and 'Certificate' sections. The main content area has tabs for 'Environment' and 'Resources', with 'AWS Credentials' selected. It shows two entries: 'Access Key' with value 'AKIASMBTBWNE7QFJAPAO' and 'Secret Key' with value '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'. Both have a copy icon next to them.

On the Ansible Lab:

Take the access key and secret key and save it as environment variables on the Lab:

Run these commands

```
# export AWS_ACCESS_KEY_ID=AKIAIXYX74ZQAJ2C4XK35
#export
AWS_SECRET_ACCESS_KEY=eVq4tjJ31qaDzN9HVbdKgGT5gtiHGjpHE9oFHpgA
```

Now execute the command to list the inventory:

```
# ansible-inventory -i /home/ansiuser/aws_ec2.yml --list
```

```

ansiuser@ip-172-31-5-209:~$ 
ansiuser@ip-172-31-5-209:~$ export AWS_ACCESS_KEY_ID=AKIAUJU24ZR3SHDUKZ74
ansiuser@ip-172-31-5-209:~$ export AWS_SECRET_ACCESS_KEY=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
ansiuser@ip-172-31-5-209:~$ ansible-inventory -i /home/ansiuser/aws_ec2.yml --list
[WARNING]: Collection amazon.aws does not support Ansible version 2.12.10
{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": [
            "aws_ec2",
            "ungrouped"
        ]
    }
}
ansiuser@ip-172-31-5-209:~$ 

```

Create Ec2 instances on AWS

On the master node run the inventory command:

```
# ansible-inventory -i /home/ansiuser/aws_ec2.yml --list
```

```

"subnet_id": "subnet-054dac8bc062c8061",
"tags": {
    "Name": "AnsibleWorker"
},
"usage_operation": "RunInstances",
"usage_operation_update_time": "2024-07-07T15:28:01+00:00",
"virtualization_type": "hvm",
"vpc_id": "vpc-0696fb7329db78a32"
}
},
"all": {
    "children": [
        "aws_ec2",
        "ungrouped"
    ]
},
"aws_ec2": {
    "hosts": [
        "ec2-174-129-147-148.compute-1.amazonaws.com",
        "ec2-3-91-81-20.compute-1.amazonaws.com",
        "ec2-34-228-22-225.compute-1.amazonaws.com",
        "ec2-54-196-237-76.compute-1.amazonaws.com",
        "ec2-54-242-112-1.compute-1.amazonaws.com",
        "ec2-54-90-100-196.compute-1.amazonaws.com"
    ]
}

```

Do these steps as Assignment:

1. Connecting Ansible controller to your cloud machine over SSH

2. Run the Ansible command on the dynamic inventory

3. If we want to create 10 servers, we would not want to execute steps of creating ansiuser, copying ssh keys and updating config file manually on all server

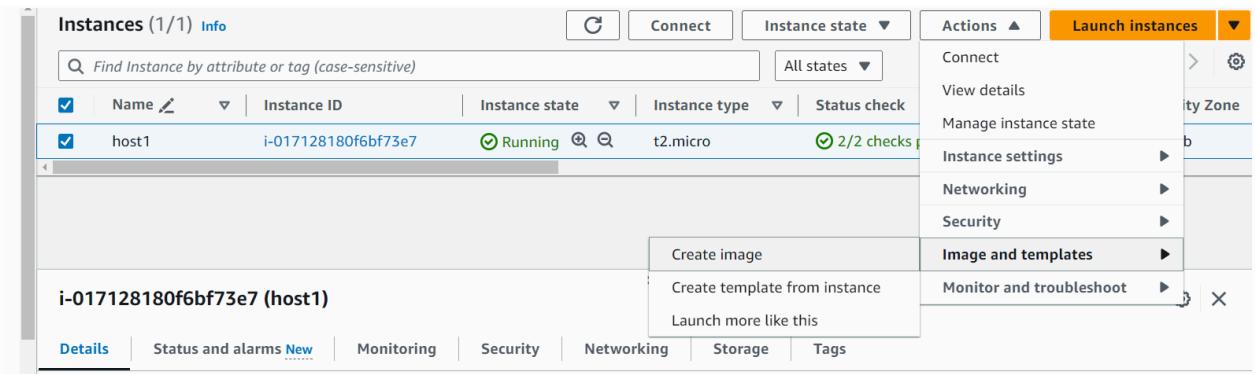
4. Solution for this is :

- Take the created ANsible worker to aws that connected to Ansible controller via ssh
- Convert the node as an AMI
- So all the worker nodes configuration/softwares/OS/user details will be available as AMI
- Now we will create new Vms with our custom AMI -> So all the new instances will have
- Ansible controller SSH keys
- Will have ansiuser and config file updates

Assignment/homework steps:

1. Now go to AWS and select your EC2 worker server

Convert the Ec2 server in to an Image



Give the Image name as AnsibleHostImage → click on create Image

You can see you image by clicking on AMI

EC2 Dashboard X

EC2 Global View

Events

Console-to-Code [Preview](#)

▼ Instances

Instances

Instance Types

Launch Templates

Spot Requests

Savings Plans

Reserved Instances

Dedicated Hosts

Capacity

Reservations [New](#)

▼ Images

AMIs

AMI Catalog

▼ Elastic Block Stores

It will take 5 mins to create the AMI

Once the AMi is available.

7. We will now create multiple Instance with the new Image itself

Click on launch Instance → neter the name and select the AMi that we have created

An AMI is a template that contains the software configuration (operating system, applications) required to launch your instance. Search or Browse for AMIs if you don't see what you're looking for below

The screenshot shows the AWS Lambda console interface. At the top, there is a search bar with the placeholder text "Search our full catalog including 1000s of application and OS images". Below the search bar, there are three tabs: "Recents", "My AMIs" (which is highlighted with a pink oval), and "Quick Start". Underneath these tabs, there are two buttons: "Owned by me" (which is selected, indicated by a blue outline and a checked radio button) and "Shared with me". The main content area displays an "Amazon Machine Image (AMI)" card. This card contains the following information: "AnsibleHostImage", "ami-0cc8d062da4d6b935", "2024-05-12T14:00:48.000Z", "Virtualization: hvm", "ENAv enabled: true", and "Root device type: ebs". A pink oval surrounds the "AnsibleHostImage" section. At the bottom of the card, there is a "Description" field containing the text "-".

Launch the instance

8 go to the master node:

```
# ansible-inventory -i aws_ec2.yml --list
```

```
# ansible aws_ec2 -i aws_ec2.yml -m ping
```

```
=====
```

Run the playbook

```
# vim playbook-dynamic.yml
```

```
- name: Execute variables from vault
```

```
hosts: aws_ec2
```

```
become: true
```

```
tasks:
```

```
- name: create a user on worker node
```

```
file: path=/tmp/file state=touch
```

Save the file and run it

```
# ansible-playbook -i aws_ec2.yml playbook-dynamic.yml
```

Terraform

Install Terraform on lab:

```
# wget -O - https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg  
  
# echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(grep -oP '(?=<UBUNTU_CODENAME=).*' /etc/os-release || lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list
```

```
# sudo apt update && sudo apt install terraform
```

Terraform Code is always written in the form of blocks:

There are different types of blocks:

- **Provider block**
- **Resource block**
- **Variable block**
- **Output block**
- **Association block**

- **Nested block**
- **Dynamic block**
- **Local block**
- **Provisioner block**
- **Module block**

```
Type of block "resourceName" "Unique BlockName" {  
    Desired infra Code  
}
```

1st block is Provider block

Example:

```
provider "aws" {  
    region  = "us-west-2"  
    access_key = "my-access-key"  
    secret_key = "my-secret-key"  
}
```

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

Demo 1: Store AWS credentials in a shared credentials file and then use it in the TF config file.

In this way your accesskey and secret key will not be exposed to the outside world.

```
# apt-get update
```

```
# apt-get install awscli -y
```

```
# aws configure
```

Give the valid access key

Give the valid secret key

Press enter, no need to give any region and format option

To verify if the credentials have been set for aws

```
# cat ~/.aws/credentials
```

Now create the provider block for terraform:

```
=====
```

```
# mkdir myterraformfiles
```

```
# cd myterraformfiles
```

```
# vim aws_infra.tf
```

```
provider "aws" {
    region = "us-east-1"
    shared_credentials_files = ["~/.aws/credentials"]
}
```

Save the file and we will install aws provider plugin

```
# terraform init
```

Demo 3: Create the first resource to create EC2 server via terraform

```
# vim aws_infra.tf
```

Add this block of code

```
resource "aws_instance" "myec2" {  
  
    ami      = "ami-02457590d33d576c3"  
    instance_type = "t2.micro"  
  
    tags = {  
        Name = "Instance1"  
    }  
}
```

Save the file.

```
# terraform validate
```

```
# terraform plan
```

```
# terraform apply --auto-approve
```

```
# terraform destroy --auto-approve
```

Demo 3:

1. Write a data block which will filter details and fetch AMI id from AWS

The fetched Ami id will be present in terraform.tfstate

Data block will not create any resource on AWS.

It is just a block that will filter AWS ami data based on your inputs and fetch it

2. Pass the fetched AMI ID form data block to aws_instance resource block.

Add below code in the file

```
# vim aws_infra.tf
```

```
data "aws_ami" "myami" {

  most_recent = true

  owners      = ["amazon"]

  filter {
    name  = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

```
resource "aws_instance" "myec2" {

  ami      = data.aws_ami.myami.id
  instance_type = "t2.micro"
```

```
tags = {  
    Name = "Instance1"  
}  
  
# terraform apply --auto-approve
```

Destroy a specific resource

```
# terraform destroy -target aws_instance.myec2
```


=====

Variables in Terraform:

```
=====
```


Variables are used to store any data

In terraform variables are created using the variable block

A variable can store a single value or a list of values

A variable block can be created with the TF config file or can be created in a separate file

Syntax of variable block:

=====

```
variable "name_variable" {  
  default = "variable-value"  
}  
=====
```

Demo:

```
# sudo su -  
  
# apt-get update  
  
# apt-get install awscli -y  
  
# aws configure
```

Give the valid access key

Give the valid secret key

Press enter, no need to give any region and format option

To verify if the credentials have been set for aws

```
# cat ~/.aws/credentials  
  
  
# mkdir myterraformfiles
```

```
# cd myterraformfiles
```

```
# vim variables.tf
```

```
variable "region" {
```

```
  default = "us-east-1"
```

```
}
```

```
variable "credentials" {
```

```
  default = "~/.aws/credentials"
```

```
}
```

```
variable "instance_type" {
```

```
  default = "t2.micro"
```

```
}
```

```
variable "env" {
```

```
  default = "Dev"
```

```
}
```

Save the file

```
# vim aws_infra.tf
```

```
provider "aws" {  
    region = var.region  
    shared_credentials_files = [var.credentials]  
}
```

```
data "aws_ami" "myami" {
```

```
    most_recent    = true
```

```
    owners        = ["amazon"]
```

```
    filter {  
        name  = "name"  
        values = ["amzn2-ami-hvm*"]  
    }
```

```
}
```

```
resource "aws_instance" "myec2" {  
  
    ami      = data.aws_ami.myami.id  
    instance_type = var.instance_type  
  
    tags = {  
        Name = var.env  
    }  
}
```

Save the file

```
# terraform init
```

```
# terraform apply
```

COUNT - META LOOP ARGUMENT

Open the same aws_infra.tf file and update the resource block with highlighted code

```
# vim aws_infra.tf
```

```
provider "aws" {
    region = var.region
    shared_credentials_files = [var.credentials]
}

data "aws_ami" "myami" {

most_recent      = true

owners           = ["amazon"]

filter {
    name  = "name"
    values = ["amzn2-ami-hvm*"]
}
}

resource "aws_instance" "myec2" {

ami      = data.aws_ami.myami.id
instance_type = var.instance_type
```

count = 5

```
tags = {  
    Name = "${var.env}-${count.index}"  
}  
}
```

Save the file and run it

```
# terraform apply --auto-approve
```

=====

Terraform Modules:

=====

Modules in terraform are nothing but collection of terraform configuration files that are written in dedicated directories

Modules encapsulates group of resources, variables, output block than this modules can be called again and again to create infrastructure in different directories

In a team one can create modules of various resources and other team members can reuse the module for infra creation.

Modules are reusable terraform code.

There are various types of modules:

- Root module
- Child modules [these are sources in the main.tf file]
- Local modules - modules that you write in your local machine and reuse them in your local machine itself
- Published modules - modules that are available for free on terraform registry written terraform team
- You can also push your local module code to VCS where others can use it. -> published Modules.

A typical module directory consist :

> TF config file
> variables.tf
> output.tf
> README.md

Demo:

```
# cd  
  
# mkdir modules dev  
  
# cd modules
```

```
# mkdir myec2    ⇒ this is a module name
```

```
# cd myec2
```

```
# vim myec2.tf
```

```
data "aws_ami" "myami" {
```

```
  most_recent = true
```

```
  owners = ["amazon"]
```

```
  filter{
```

```
    name = "name"
```

```
    values = ["amzn2-ami-hvm*"]
```

```
}
```

```
}
```

```
resource "aws_instance" "test-ec2" {
```

```
  ami = data.aws_ami.myami.id
```

```
instance_type = var.instance_type
```

```
tags = {  
  Name = "Instance-test"  
}
```

```
}
```

Save the file,

```
# vim variables.tf
```

```
variable "instance_type" {  
  default = "t2.micro"  
}
```

Save the file

Copy the path of directory

```
/root/modules/myec2
```

```
=====
```

Now lets call or use the modules in the TF config file:

```
=====
```

```
# cd

# cd dev

# vim main.tf

provider "aws" {
    region = "us-east-1"
    shared_credentials_files = ["~/.aws/credentials"]
}

module "myec2module" {

source = "/root/modules/myec2"

}

module "myec2module-2" {

source = "/root/modules/myec2"
instance_type = "t2.nano"

}
```

Save the file

```
# terraform init
```

This will download the modules in current directory

```
# terraform plan
```

Create a module that will create resources in different regions of AWS

```
# cd modules
```

```
# mkdir region_module
```

```
# cd region_module
```

```
# vim region.tf
```

```
variable "region" {
```

```
    type = string
```

```
}
```

```
provider "aws" {

region = var.region

shared_credentials_files = ["~/.aws/credentials"]

}

data "aws_ami" "myami"{

most_recent = true

owners = ["amazon"]

filter {

  name  = "name"
  values = ["amzn2-ami-hvm*"]
}

}

resource "aws_instance" "myec2-1" {
  ami      = data.aws_ami.myami.id
  instance_type = "t2.micro"
```

```
tags = {  
    Name = "terraform1"  
}
```

```
}
```

Save the file

The path of module is: /root/modules/region_module

```
# mkdir provider_demo
```

```
# cd provider_demo
```

```
# vim main.tf
```

```
variable "regions" {
```

```
    type = list(string)
```

```
    default = ["us-east-1", "us-east-2", "us-west-1"]
```

```
}
```

```
module "aws_region_demo" {

  source = "/root/modules/region_module"

  region = var.regions[0]

}

module "aws_region_demo1" {

  source = "/root/modules/region_module"

  region = var.regions[1]

}

# terraform init

# terraform plan
```

Dynamic Block

```
=====
```

```
# mkdir mydemo2
```

```
# cd mydemo2
```

```
# vim sg_aws.tf
```

```
provider "aws" {
```

```
    region = "us-east-1"
```

```
    shared_credentials_files = ["~/.aws/credentials"]
```

```
}
```

```
variable "sg_ports" {
```

```
    type = list(number)
```

```
default = [22,443,80,8080]
```

```
}
```

```
resource "aws_security_group" "mysg" {
```

```
  name      = "sl-sg"
```

```
  description = "Allow TLS inbound traffic and all outbound traffic"
```

```
  dynamic "ingress" {
```

```
    for_each = var.sg_ports
```

```
    iterator = ports
```

```
    content{
```

```
      from_port      = ports.value
```

```
      to_port       = ports.value
```

```
    protocol      = "tcp"

    cidr_blocks  = ["0.0.0.0/0"]

}

}

}
```

```
# terraform init
```

```
# terraform plan
```

Variable of type Object

```
# vim demo1.tf
```

```
variable "ec2_data" {
```

```
  type = object ( {
```

```
    name = string
```

```
    instancetype = string
```

```
    instance = number
```

```
  }
```

```
default = {
```

```
  name = "Dev"
```

```
  instancetype = "t2.micro"
```

```
  instance = 2
```

```
}
```

```
}
```

```
data "aws_ami" "myami" {
```

```
  most_recent = true
```

```
  owners = ["amazon"]
```

```
  filter {
```

```
    name  = "name"
```

```
    values = ["amzn2-ami-hvm*"]
```

```
}
```

```
}
```

```
resource "aws_instance" "myec2-1" {

    ami      = data.aws_ami.myami.id
    instance_type = var.ec2_datainstancetype
    count = var.ec2_data.instance
    tags = {
        Name = var.ec2_data.name
    }
}
```

```
# terraform init
```

```
# terraform plan
```

```
=====
```

```
# mkdir mydemo2

# cd mydemo2

# vim variables.tf

variables.tf

variable "ec2_object"{

type = map(object ({

    name = string

    Instancetype = string

}))}

default = {

    "dev-env"= {

        name = "dev01"

        Instancetype = "t2.micro"
    }
}
```

},

"test-env"= {

name = "test"

Instancetype = "t2.medium"

},

"prod-env"= {

name = "prod01"

Instancetype = "t2.large"

}

}

}

Save the file

```
# vim main.tf

provider "aws" {

region = "us-east-1"

shared_credentials_files = ["~/.aws/credentials"]

}

data "aws ami" "myami"{

most_recent = true

owners = ["amazon"]
```

```
filter {

    name  = "name"

    values = ["amzn2-ami-hvm*"]

}

}
```

```
resource "aws_instance" "myec2-1" {

    ami      = data.aws_ami.myami.id

    for_each = var.ec2_object

    instance_type = each.valueinstancetype

    tags = {
```

```
Name = each.value.name  
  
}  
  
}
```

Save the file

```
# terraform init  
  
# terraform plan
```

Setup the lab by installing terraform and setting up AWS credentials

Install Terraform on lab:

```
# wget -O - https://apt.releases.hashicorp.com/gpg | sudo gpg  
--dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg  
  
# echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(grep -oP  
'(?<=UBUNTU_CODENAME).*/etc/os-release || lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list
```

```
# sudo apt update && sudo apt install terraform -y
```

Store AWS credentials in a shared credentials file and then use it in the TF config file.

In this way your accesskey and secret key will not be exposed to the outside world.

apt-get update

apt-get install awscli -y

aws configure

Give the valid access key

Give the valid secret key

Press enter, no need to give any region and format option

To verify if the credentials have been set for aws

```
# cat ~/.aws/credentials
```

Agenda:

Provisioners

- Local-exec provisioner
- Remote-exec provisioner

Backup of Terraform state file

Enables logs in terraform

Provisioners:

If we want terraform to execute any command on the current local machine or on a remote server than we make use of provisioner block

Provisioner block is written inside a resource block

There are 2 types of provisioner blocks:

1. Local-exec
2. Remote- exec

Using local-exec provisioner, you can run any command on the local machine

=====

Demo:

=====

```
# mkdir myterraformfiles  
# cd myterraformfiles  
# vim aws_infra.tf
```

aws_infra.tf

```
provider "aws" {  
  
region = "us-east-1"  
shared_credentials_files = ["~/.aws/credentials"]
```

```
}
```

```
resource "null_resource" "local_cmd" {
```

```
provisioner "local-exec" {
```

```
  command = "echo 'Hello from terraform' > ./myfile.txt"
```

```
}
```

```
}
```

```
resource "tls_private_key" "mykey" {
```

```
  algorithm = "RSA"
```

```
}
```

```
resource "aws_key_pair" "aws-key" {
```

```
  key_name  = "web-key"
```

```
  public_key =
```

```
  tls_private_key.mykey.public_key_openssh
```

```
provisioner "local-exec" {
```

```
command = "echo  
'${tls_private_key.mykey.private_key_pem}' >  
./web-key.pem"
```

```
}
```

```
}
```

Save and run the commands

```
# terraform init
```

```
# terraform apply --auto-approve
```

Check if files have been created in local directory

Check if key is created in AWS

```
=====
```

Remote-exec provisioner:

```
=====
```

In the same directory as that of local exec , add a new file

```
# vim myinfra.tf
```

```
resource "aws_vpc" "sl-vpc" {
```

```
cidr_block = "10.0.0.0/16"
```

```
tags = {
```

```
    Name = "sl-vpc"
```

```
}
```

```
}
```

```
resource "aws_subnet" "sl-subnet" {
```

```
vpc_id      = aws_vpc.sl-vpc.id
```

```
cidr_block = "10.0.1.0/24"
```

```
depends_on = [aws_vpc.sl-vpc]
```

```
map_public_ip_on_launch = true
```

```
tags = {
    Name = "sl-subnet"
}
}

resource "aws_route_table" "sl-route-table" {
    vpc_id = aws_vpc.sl-vpc.id
```

```
tags = {
    Name = "sl-route-table"
}
}
```

```
resource "aws_route_table_association" "a" {
    subnet_id      = aws_subnet.sl-subnet.id
    route_table_id = aws_route_table.sl-route-table.id
}
```

```
resource "aws_internet_gateway" "gw" {
    vpc_id = aws_vpc.sl-vpc.id
    depends_on = [aws_vpc.sl-vpc]
    tags = {
        Name = "sl-gw"
```

```
    }
}

resource "aws_route" "sl-route" {
    route_table_id      =
aws_route_table.sl-route-table.id
    destination_cidr_block  = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
}

variable "sg_ports" {

type = list(number)
default = [22,443,80,8080]

}

resource "aws_security_group" "sl-sg" {
    name      = "sl-sg"
    description = "Allow TLS inbound traffic and all
outbound traffic"
    vpc_id = aws_vpc.sl-vpc.id
```

```
dynamic "ingress" {
  for_each = var.sg_ports
  iterator = ports
  content{
    from_port      = ports.value
    to_port        = ports.value
    protocol       = "tcp"
    cidr_blocks   = ["0.0.0.0/0"]

  }
}

egress {
  from_port      = 0
  to_port        = 0
  protocol       = "-1"
  cidr_blocks   = ["0.0.0.0/0"]

}

data "aws_ami" "myami" {
```

```
most_recent      = true

owners          = ["amazon"]

filter {
    name  = "name"
    values =
["amzn2-ami-kernel-5.10-hvm-2.0.20250610.0-x86_64-
gp2"]
}

resource "aws_instance" "myec2" {

    ami      = data.aws_ami.myami.id
    instance_type = "t2.micro"
    key_name = "web-key"
    subnet_id = aws_subnet.sl-subnet.id
    security_groups = [aws_security_group.sl-sg.id]
    tags = {
        Name = "terraform-instance"
    }
}
```

```
provisioner "remote-exec" {

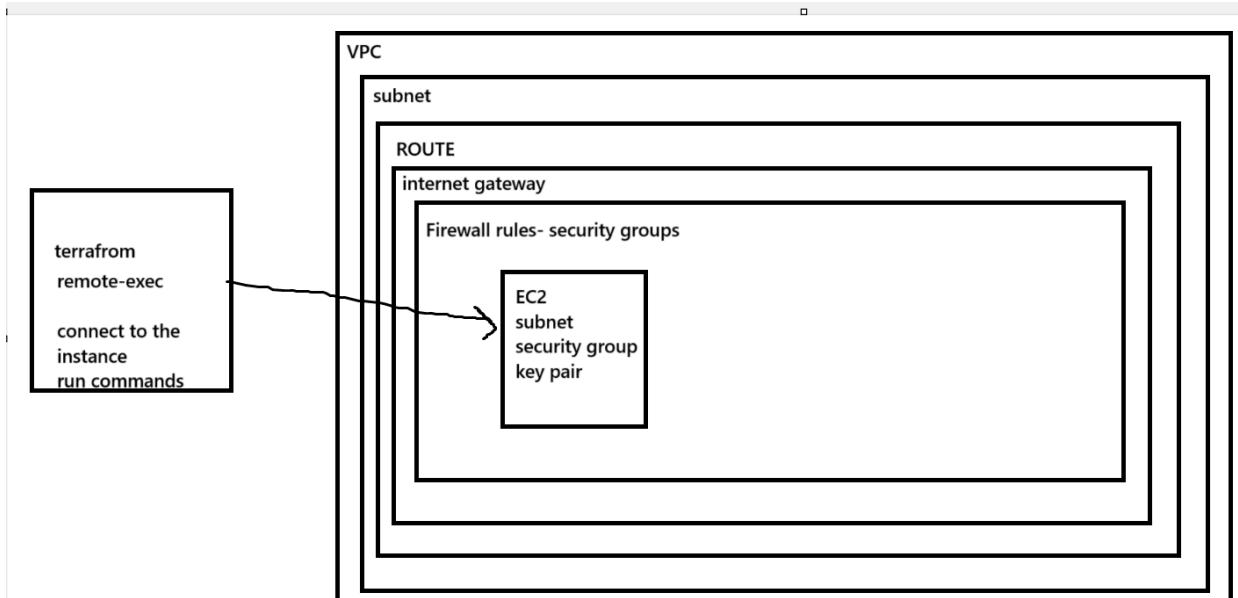
    connection {
        type = "ssh"
        user = "ec2-user"
        private_key =
        tls_private_key.mykey.private_key_pem
        host = self.public_ip
    }

    inline = [
        "sudo yum install git -y",
        "sudo yum install httpd -y",
        "sudo systemctl start httpd",
        "sudo systemctl enable httpd"
    ]

}

# terraform init
```

```
# terraform apply --auto-approve
```



Manage terraform State File:

Terraform we do create or update resources -> we will find a terraform state file created or updated automatically

What ever infrastructure that terraform creates it will be recorded in the terraform state file

In which ever folder we run the terraform command → same folder terraform state file will be present

The terraform state file is created in JSON format storing all parameters of resources created in AWS

The terraform state file is a Private API

=====

The state file format is a private API that is meant to be used with in Terraform only

We should never manually update the terraform state file, it should be automatically managed by Terraform. If we have to do any changes on the terraform state file it is recommended to use terraform state command or terraform import command

It is always better to take back of the terraform state file:

=====

> if we are using terraform for personal practice or use then we can maintain the state file in our local machine itself.

But if we are using terraform with multiple team members in a real time project then in that case:

- > we will have to maintain the state file in a shared storage or location so that each team member can access the same terraform state file
- > But if multiple people try to run terraform command using the same state file, then we will get into errors related to state file locking
- > Also when multiple people are using the same state file than we may have more 1 person updating the same infrastructure, so we will get conflicts or our file may have data loss or may be corrupted
- > this can be resolved by isolating the environments for each team member and each one of them having their copy of state file

The solution could be:

=====

The most common solution is -> to place the file in the version control tool along with the terraform code.

However this may also lead to major issues:

Manual error -> team members may pull the latest code and state file.. Make changes but may forget to push the changes to the VC tool

In VC tool when the team members pull the changes there is concept of lock in VC tool..

Now both the team members may be running the same state file

There are chances that we may expose sensitive data via state file in VC tool

Storing State files in VC tool is not the best solution.

So some other solutions could be:

Using terraform built-in support for remote backends

Remote backends determines how terraform loads and stores the state files

We have been using this default backend in terraform so far,

but that was local backends which has been storing your state file in local disk or directory

Terraform remote backends allows us to store the state file in a remote shared store.

For example in Amazon S3, Azure storage, GCP storage, terraform cloud

Using the remote backend storage we will be able to resolve:

Manual errors: With remote backend, terraform will by itself update the state file on remote storage whenever you run `terraform apply` command

So there is no chance of manual errors.

Locking: Most of the remote backend storage supports Locking. When anybody runs `terraform apply` command,

terrafrom will automatically acquire a lock on the state file,

so now if someone else runs a `terraform apply`, the user will have to wait as the state file is locked.

Here, we can also add a `-local-timeout=10 mins` parameter to instruct terraform to wait up to 10 mins for the local to be released and then run the next terraform apply

Secrets: Most of the backends store supports encryption in transit and encryption in the state file

Also all the storage on remote also are secure and one should have correct permissions and roles to access the storage

We will use AMazon S3 (Simple storage service) bucket to store out state files remotely.

The reason being:

We already using this provider in our terraform configuration, so no special set up is needed

We already know that S3 bucket is designed in such a way that it is 99.999% always available, which means we dont have to worry about data loss or outages Its supports encryption so we can easily manage sensitive data

It supports locking of state file using dynamo DB

It supports versioning of the state file, so you can always roll back to older versions if something goes down.

It is inexpensive and available in AWS for free

Demo:

```
# mkdir statedemo
# cd statedemo

# vim state-demo.tf

provider "aws" {
  region = "us-east-1"
  shared_credentials_files = ["~/.aws/credentials"]

}

resource "aws_s3_bucket" "terraform-state" {
  bucket = "terraform-s3-bucket-state-file"
  lifecycle{
    prevent_destroy = true
  }
}

resource "aws_s3_bucket_versioning" "versioning_example" {
  bucket = aws_s3_bucket.terraform-state.id
  versioning_configuration {
    status = "Enabled"
```

```
    }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "example" {
  bucket = aws_s3_bucket.terraform-state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

```
resource "aws_s3_bucket_public_access_block" "example" {
  bucket = aws_s3_bucket.terraform-state.id

  block_public_acls      = true
  block_public_policy     = true
  ignore_public_acls     = true
  restrict_public_buckets = true
```

```
}

resource "aws_dynamodb_table" "terraform-local" {
    name          = "terraform-state-table"
    billing_mode = "PAY_PER_REQUEST"
    hash_key     = "LockID"
    attribute{
        name = "LockID"
        type = "S"
    }
}
```

```
# terraform init
# terraform apply
```

Open the same file and add the backend information

```
terraform {
    backend "s3" {
        bucket = "terraform-s3-bucket-state-file"
        key    = "global/s3/terraform.tfstate"
        region = "us-east-1"
```

```
dynamodb_table = "terraform-state-table"
encrypt = true
}
}
```

```
# terraform init
```

Give yes

Go to aws → search for s3 bucket -> you will see your bucket.

```
=====
```

Terraform Cloud:

```
=====
```

- Hashicorp provides terraform as a service over the internet which we call as Terraform cloud
- This service is provided with terraform enterprise, however we do have a community version for trial
- On terraform cloud we can login with our gmail account and run our terraform config files
- TF cloud can securely connect to AWS using the provided access key and secret key and create/delete/update infra

- The TF code can be written in local IDE and then maintained in github or VC tool
- TF cloud can connect to VC tool fetch code and run it
- TF cloud will maintain the version of your TF state file and backup of the TF state file
- TF cloud we can create variables of type environment variables or secret variables to store data that will be passed to TF code at runtime
- TF cloud supports creation of workspaces where you can use the same TF config to create infra that is available for different environments like Test, dev, prod. Each workspace is unique and will have its own TF state file
- TF cloud they provide us execution history of terraform init, plan or apply command
- In TF cloud we create projects and we can run TF config file, it can be considered like a container
- TF cloud we can implement RBAC where we can created team, team members and give them roles and permissions.

Features of Terraform Cloud:

1. Organization:

Organization on TF cloud is a set of projects that can be called as a logical environment where we will maintain our configuration file, state file, variables and history

2. Workspaces in terraform:

Workspaces helps us to isolate TF state files for each environment

Can be considered as isolated folders for each environment

They can be considered like git branches

They are just logical directories that helps us to manage multiple deployments of the same configuration file.

3. Remote State & execution:

In TF cloud we maintain the state file centrally, this state file is version controlled and encrypted. SO there is no need to take backup in S3 bucket and dynamo DB

4. On the dashboard it clearly visible of resource that will created and its cost on AWS

5. Integration with VCS

As we update the github repo with new TF code, immediately the code will run and latest infra will be created on AWS

6. Policy enforcement:

TF cloud implements Sentinel policies which are conditions to be checked before infra is created in AWS

7. TF cloud allows to share our modules with the other workspaces in the organization

8. Displays the Run history and audits logs

9. TF cloud comes with a free version with limited features however we also have a complete enterprise version of TF cloud.

Demo:

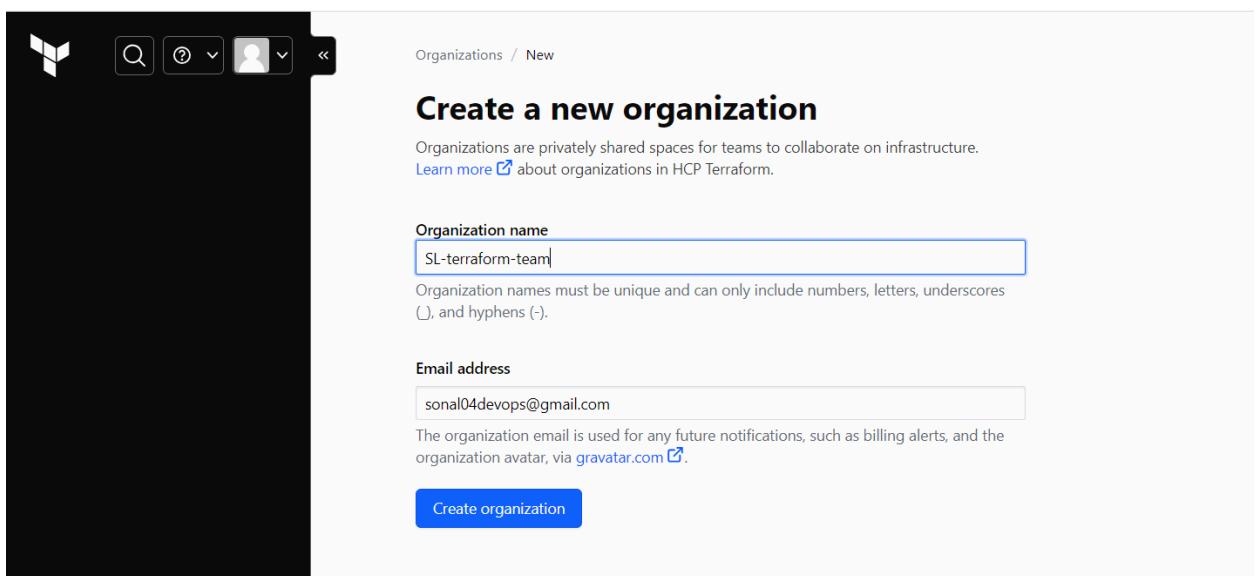
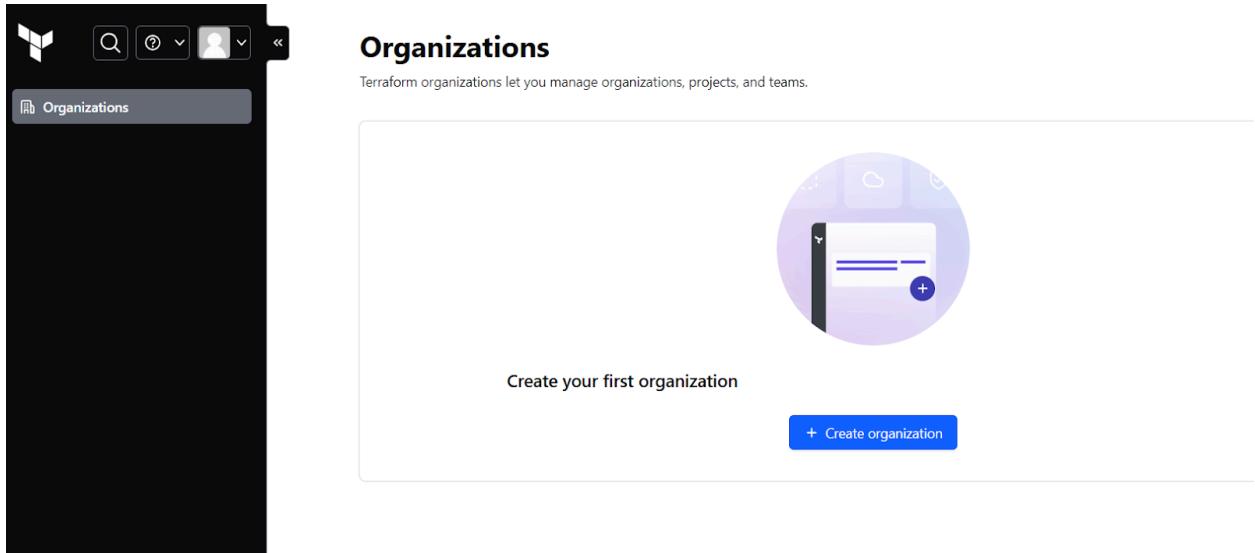
=====

Go to this URL and login with your valid Gmail id

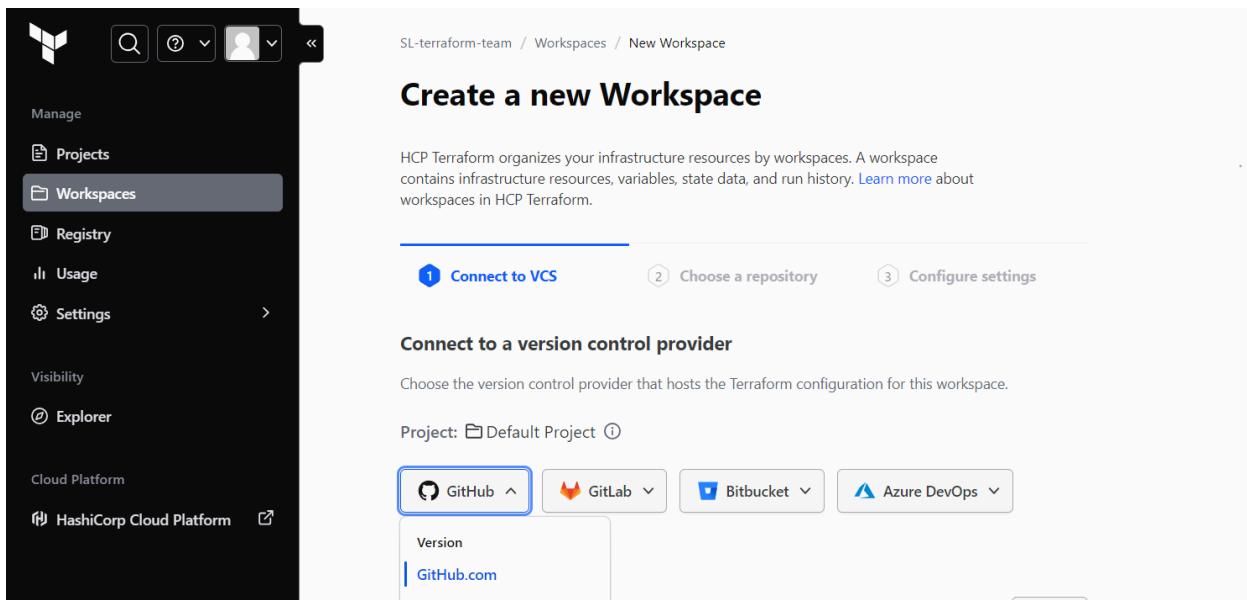
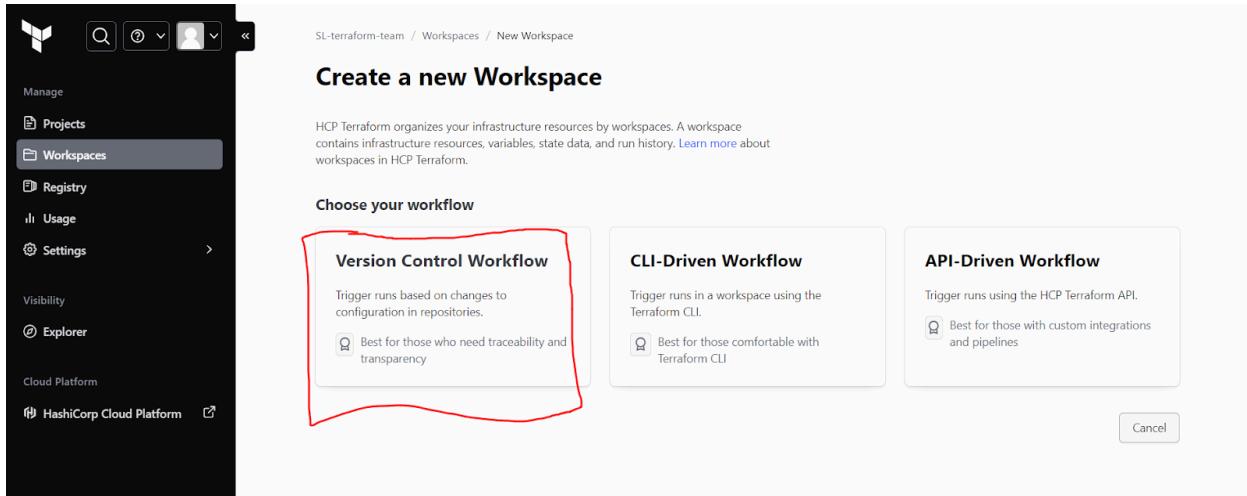
<https://app.terraform.io/public/signup/account>

You will be on organizations pge

Create organization



**You will be on workspace page
Click on version control workflow**



Select github.com → select your repo → click on the repo name → you will on configure settings

The screenshot shows the HCP Terraform interface. On the left is a dark sidebar with navigation links: Manage, Projects, Workspaces (which is selected and highlighted in grey), Registry, Usage, Settings, Visibility, Explorer, and Cloud Platform. At the bottom of the sidebar is the HashiCorp Cloud Platform logo. The main area has a light background. At the top, it says "SL-terraform-team / Workspaces / New Workspace". Below that is the title "Create a new Workspace". A descriptive text explains that HCP Terraform organizes resources by workspaces, mentioning infrastructure resources, variables, state data, and run history. It includes a link to "Learn more" about workspaces. The next section is titled "Configure Settings". Under "Workspace Name", there is a text input field containing "terraform-cloud-demo". A note below it states that workspace names must be unique and can use dashes, underscores, and alphanumeric characters. Under "Description (Optional)", there is a larger text input field labeled "Workspace description". At the bottom of the form is a section titled "Advanced options" with a collapse arrow. At the very bottom are "Previous" and "Create" buttons.

Click on continue to setup workspace

The screenshot shows the workspace overview for "terraform-cloud-demo". The sidebar on the left is identical to the previous screenshot, showing the "Workspaces" section is selected. The main area displays the workspace details: ID: ws-1h1kXrKVdwhUBUov, Status: Unlocked, Resources: 0, Terraform Version: v1.9.3, and Last Updated: 2 minutes ago. A success message "Configuration uploaded successfully" is shown with a green checkmark icon. Below it, a section titled "Next step: configure variables" instructs users to configure required variables before starting a run. A "Configure variables" button is available. A note states that if no variables are configured, a plan can be started. A "Start new plan" button is present. To the right, there are sections for "Metrics" (not yet available) and "Tags (0)". Metrics will appear once the next run is applied. Tags are currently empty, with a placeholder "Add a tag". Other workspace metadata includes: Sona0409/terraform-cloud-demo, Execution mode: Remote, Auto-apply API, CLI, & VCS runs: Off, Auto-apply run triggers: Off, and Project: Default Project.

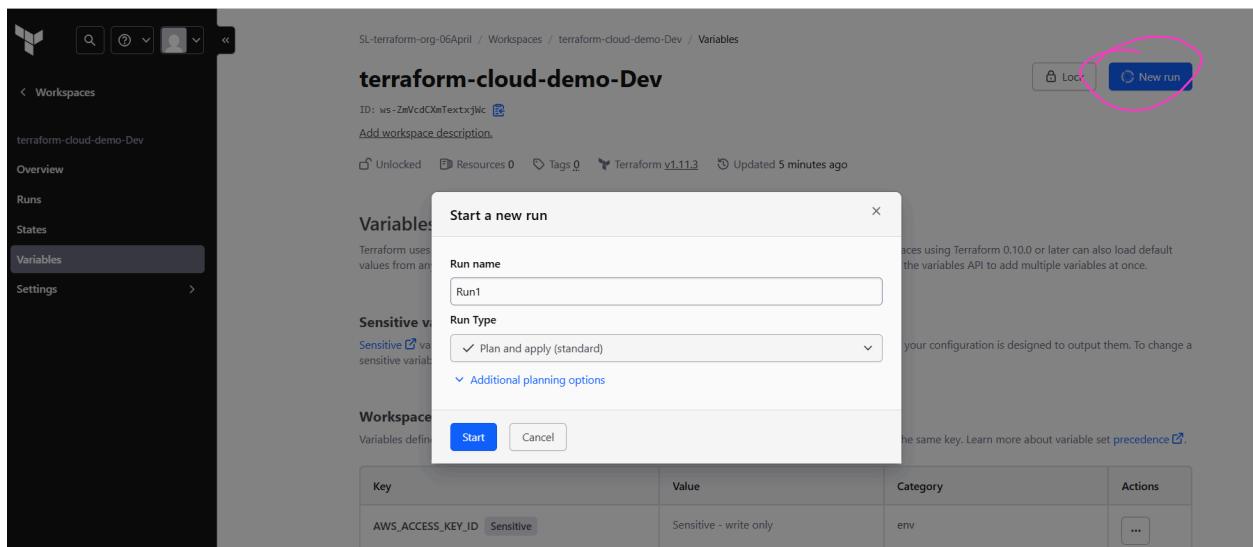
Click on configure variables → Click on add variable→ select environment

**Give key as : AWS_ACCESS_KEY_ID
Value as accesskey id create din aws
Select sensitive**

Click on add variable

**Give key as : AWS_SECRET_ACCESS_KEY
Value as secret key created in aws
Select sensitive
Click on add variable**

Now lets click on new run → Give run name →click on start



Sentinel Policy:

It is a powerful method of adding governance rules in the terraform Cloud

These allow organization or workspace to setup rules that have been passed when a terraform run is triggered

Sentinel Policy are rules that ensure correct infra is provisioned.

Sentinel Policy: policy as a code- framework developed by hashicorp

Readymade code for Sentinel Policy is available for aws cloud , we just have to integrate our Sentinel Policy code with the TF cloud organization.

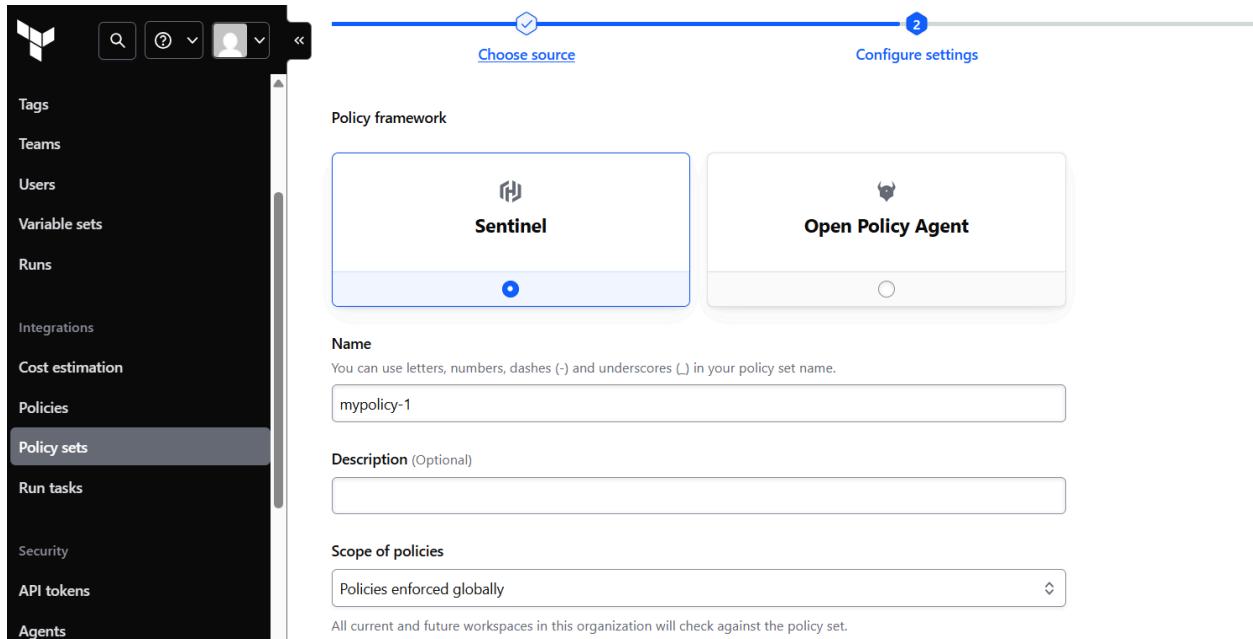
Sentinel Policy tightly integrates with your workspace plan during the runs

Sentinel policies have to be applied:

- 1. When we want to enforce naming conventions**
- 2. We want to restrict infra creation to specific regions**
- 3. We want to restrict resources type**
- 4. We want to restrict instance types**
- 5. We want to ensure correct tags are given to resources**
- 6. To restrict certain user to only run in a workspace.**

Demo: Sentinel policy creation for an organization in TF cloud

**Go to organization → go to settings → go to policy sets
Click on new policy set → click on Individually managed policy → Add below details**



Name
You can use letters, numbers, dashes (-) and underscores (_) in your policy set name.

Description (Optional)

Scope of policies

Policies enforced globally

All current and future workspaces in this organization will check against the policy set.

Execution mode

Agent
Policy sets can be pinned to a specific runtime version.

Legacy
Cost estimation is supported in these policy sets.

[Go back](#)

Press next → click on connect policy set

Now go to policy on left side → click on create policy -> select sentinel -> give a name

SL-terraform-ORG-June2025 / Settings / Policies / New

Create a new Policy

Add your policy to a policy set (below). Your policy will apply to all workspaces attached to the policy sets you choose.

Policy framework

Sentinel

Open Policy Agent

Name
You can use letters, numbers, dashes (-) and underscores (_) in your policy name.

Description (Optional)

Enforcement behaviour

- Advisory
Failed policies produce a warning.
- Soft mandatory
Failed policies can be overridden.
⚠ You have 1 soft mandatory policy left.
- Hard mandatory
Failed policies stop the run.
⚠ You have 1 hard mandatory policy left.

Policy code (Sentinel)

```
1 import "tfplan"
2 main = rule {
3     all tfplan.resources.aws_instance as _, instances {
4         all instances as _, r {
5             (length(r.applied.tags) else 0) > 0
6         }
7     }
8 }
```

②

Add the below code

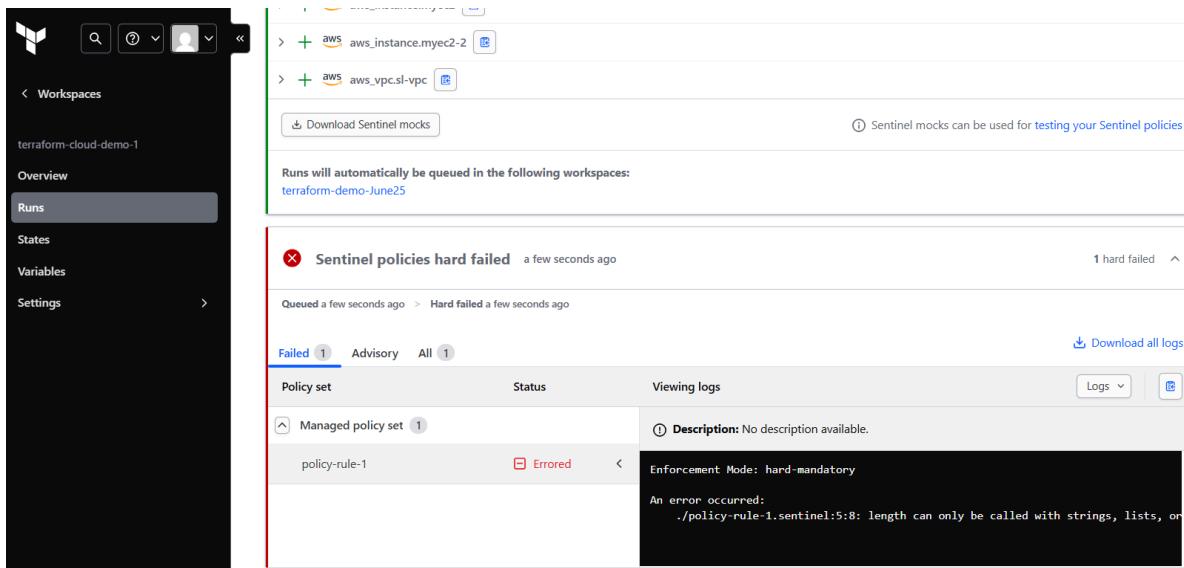
```
import "tfplan"
main = rule {
    all tfplan.resources.aws_instance as _, instances {
        all instances as _, r {
            (length(r.applied.tags) else 0) > 0
        }
    }
}
```

}

Select the policy set → click on add policy set → Click on Create policy

Now go to github repo → remove tags from the aws instance resource block

Go to Workspace in TF cloud → automatically the run will start → sentinel policy will fail and tag is not available



Add this new code for applying rules for allowed instance type:

```
import "tfplan"

# Get all AWS instances from all modules
get_aws_instances = func() {
    instances = []
    for tfplan.module_paths as path {
        instances += values(tfplan.module(path).resources.aws_instance) else []
    }
    return instances
}

# Allowed Types
allowed_types = [
    "t2.small",
    "t2.medium",
    "t2.large",
]

aws_instances = get_aws_instances()

# Rule to restrict instance types
instance_type_allowed = rule {
    all aws_instances as _, instances {
        all instances as index, r {
            r.applied.instance_type in allowed_types
        }
    }
}
```

```
    }
}

# Main rule that requires other rules to be true
main = rule {
  (instance_type_allowed) else true
}
```

Update the policy

Go to workspace – run it - policy will fail

OR:

=====

This policy restricts the AWS region based on the region set for

instances of the AWS provider in the root module of the workspace.

It does not check providers in nested modules.

```
import "tfconfig"
import "tfplan"
import "strings"
```

```
# Initialize array of regions found in AWS providers
region_values = []
```

```
# Allowed Regions
allowed_regions = [
    "us-east-1",
    "us-east-2",
    "us-west-1",
    "us-west-2",
]

# Iterate through all AWS providers in root module
if ((length(tfconfig.providers) else 0) > 0) {
    providers = tfconfig.providers
    if "aws" in keys(providers) {
        aws = tfconfig.providers.aws
        aliases = aws["alias"]
        for aliases as alias, data {
            print ( "alias is: ", alias )
            region = data["config"]["region"]
            if region matches "\$\${var\.(.*)}" {
                # AWS provider was configured with variable
                print ( "region is a variable" )
                region_variable =
                    strings.trim_suffix(strings.trim_prefix(region, "${var."), })
                print ( "region variable is: ", region_variable )
                print ( "Value of region is: ",
tfplan.variables[region_variable] )
                region_value = tfplan.variables[region_variable]
                region_values += [region_value]
            } else {

```

```
    print ( "region is a hard-coded value" )
    print ( "Value of region is: ", region )
    region_value = region
    region_values += [region_value]
}
}
}
}

# Print all regions found in AWS providers
print ( "region_values is: ", region_values )

aws_region_valid = rule {
  all region_values as rv {
    rv in allowed_regions
  }
}

main = rule {
  (aws_region_valid) else true
}
```

```
=====
=====
```

Terraform Graph:

```
=====
```

By default the result is a simplified graph which describes only the dependency ordering of the resources (resource and data blocks) in the configuration.

```
# sudo apt-get update
```

```
# sudo apt-get install graphviz -y
```

Create a TF config file

```
# mkdir mydemo
```

```
# cd mydemo
```

```
# vim main.tf
```

```
provider "aws" {
```

```
  region = "us-east-1"
```

```
}
```

```
resource "tls_private_key" "mykey" {
```

```
  algorithm = "RSA"
```

```
}
```

```
resource "aws_key_pair" "aws_key" {
    key_name  = "web-key"
    public_key = tls_private_key.mykey.public_key_openssh

    provisioner "local-exec" {
        command = "echo
'${tls_private_key.mykey.private_key_openssh}' > ./web-key.pem"

    }
}

resource "aws_vpc" "sl-vpc" {
    cidr_block = "10.0.0.0/16"
    tags = {
        Name = "sl-vpc"
    }
}

resource "aws_subnet" "subnet-1" {
```

```
vpc_id = aws_vpc.sl-vpc.id
cidr_block = "10.0.1.0/24"
depends_on = [aws_vpc.sl-vpc]
map_public_ip_on_launch = true
tags = {
    Name = "sl-subnet"
}
}
```

```
resource "aws_route_table" "sl-route-table"{
vpc_id = aws_vpc.sl-vpc.id
tags = {
    Name = "sl-route-table"
}
}
```

```
resource "aws_route_table_association" "a" {
    subnet_id      = aws_subnet.subnet-1.id
    route_table_id = aws_route_table.sl-route-table.id
}
```

```
}
```

```
resource "aws_internet_gateway" "gw" {  
    vpc_id = aws_vpc.sl-vpc.id  
    depends_on = [aws_vpc.sl-vpc]  
    tags = {  
        Name = "sl-gw"  
    }  
}
```

```
}
```

```
resource "aws_route" "sl-route" {  
  
    route_table_id = aws_route_table.sl-route-table.id  
    destination_cidr_block = "0.0.0.0/0"  
    gateway_id = aws_internet_gateway.gw.id  
  
}
```

Save the file

```
# terraform init
```

```
# terraform plan -out=myplan.out
```

Turn the plan into DOT graph

```
# terraform graph -plan=myplan.out -type=apply -draw-cycles > graph.dot
```

Render the graph into an Image using graphviz

```
# dot -Tpng graph.dot > /tmp/mygraph.png
```

**Go to lab desktop→ click on Home folder >
→ click on file system on the left side → Click on tmp folder -> click on your mygraph.png file.**

Command to see the myplan.out

```
# terraform show myplan.out
```

Demo2:

Come back to the terminal

```
# terraform apply --auto-approve
```

Run the below command to generate the graph

```
# terraform graph | dot -Tpng > /tmp/demo.png
```

Go to lab desktop→ click on Home folder >

→ click on file system on the left side → Click on tmp folder
-> click on your demo.png file.

You will see the graph

=====

Logging and Debugging

=====

Verbose Output: Use -debug or -trace flags with Terraform commands (terraform plan -debug, terraform apply -debug) to get more detailed logs for debugging.

Provider Logs: Some providers (like AWS) generate detailed logs that can help diagnose issues.

Check provider-specific logs or enable detailed logging where possible.

Demo:

=====

Go to the directory with your main.tf file.

Enable logging

```
# export TF_LOG="DEBUG"
```

Run the command

```
# TF_LOG=ERROR terraform plan
```

```
# TF_LOG=TRACE terraform plan
```

```
# TF_LOG=DEBUG terraform plan
```

Save the log in the file

```
# export TF_LOG_PATH="/tmp/terraform-debug.log"
```

Run the

```
# terraform apply
```

See the log now

```
cat /tmp/terraform-debug.log
```

```
=====
```

Project 1 : Configuration Management with Ansible and Terraform

Step 1: Configure AWS CLI with access key and secret key to establish connection remotely

```
# apt-get update && apt-get install awscli -y
```

```
# aws configure
```

Step 2: Install Terraform

```
# wget -O - https://apt.releases.hashicorp.com/gpg | sudo
gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg

# echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gp
g] https://apt.releases.hashicorp.com $(grep -oP
'(?=<UBUNTU_CODENAME=).*' /etc/os-release || lsb_release
-cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

# sudo apt update && sudo apt install terraform
```

Step 3: Configure Terraform with new ssh key which will be used as key pair for launching VMs

```
# mkdir myproject

# cd myproject

# vim mykey.tf

provider "aws" {

region = "us-east-1"
}

resource "tls_private_key" "mykey" {
algorithm = "RSA"
```

```
}
```

```
resource "aws_key_pair" "aws-key" {
    key_name  = "web-key"
    public_key = tls_private_key.mykey.public_key_openssh

provisioner "local-exec" {

command = "echo
`${tls_private_key.mykey.private_key_pem}` > ./web-key.pem"

}

}

# terraform init

# terraform apply
```

Step 4: Terraform script to provision and empty sandbox, add various setting to the sandbox like VPC, security group, route table, subnets, and key pair

```
# vim main.tf

resource "aws_vpc" "sl-vpc" {

cidr_block = "10.0.0.0/16"
```

```
tags = {
    Name = "sl-vpc"
}

}

resource "aws_subnet" "sl-subnet" {

    vpc_id      = aws_vpc.sl-vpc.id

    cidr_block = "10.0.1.0/24"

    depends_on = [aws_vpc.sl-vpc]

    map_public_ip_on_launch = true

    tags = {
        Name = "sl-subnet"
    }
}

resource "aws_route_table" "sl-route-table" {

    vpc_id = aws_vpc.sl-vpc.id

    tags = {
        Name = "sl-route-table"
    }
}

resource "aws_route_table_association" "a" {
```

```
subnet_id      = aws_subnet.sl-subnet.id
route_table_id = aws_route_table.sl-route-table.id
}
```

```
resource "aws_internet_gateway" "gw" {
  vpc_id = aws_vpc.sl-vpc.id
  depends_on = [aws_vpc.sl-vpc]
  tags = {
    Name = "sl-gw"
  }
}
```

```
resource "aws_route" "sl-route" {
  route_table_id      = aws_route_table.sl-route-table.id
  destination_cidr_block  = "0.0.0.0/0"
  gateway_id = aws_internet_gateway.gw.id
}
```

```
variable "sg_ports" {
  type = list(number)
  default = [22,443,80,8080]
}
```

```
resource "aws_security_group" "sl-sg" {
  name      = "sl-sg"
```

```
description = "Allow TLS inbound traffic and all outbound
traffic"
vpc_id = aws_vpc.s1-vpc.id
dynamic "ingress" {
  for_each = var.sg_ports
  iterator = ports
  content{
    from_port      = ports.value
    to_port        = ports.value
    protocol       = "tcp"
    cidr_blocks   = ["0.0.0.0/0"]
  }
}

egress {
  from_port      = 0
  to_port        = 0
  protocol       = "-1"
  cidr_blocks   = ["0.0.0.0/0"]
}

data "aws_ami" "myami" {
  most_recent    = true
}
```

```
owners      = ["amazon"]

filter {
  name  = "name"
  values =
  ["amzn2-ami-kernel-5.10-hvm-2.0.20250610.0-x86_64-gp2"]
}

}

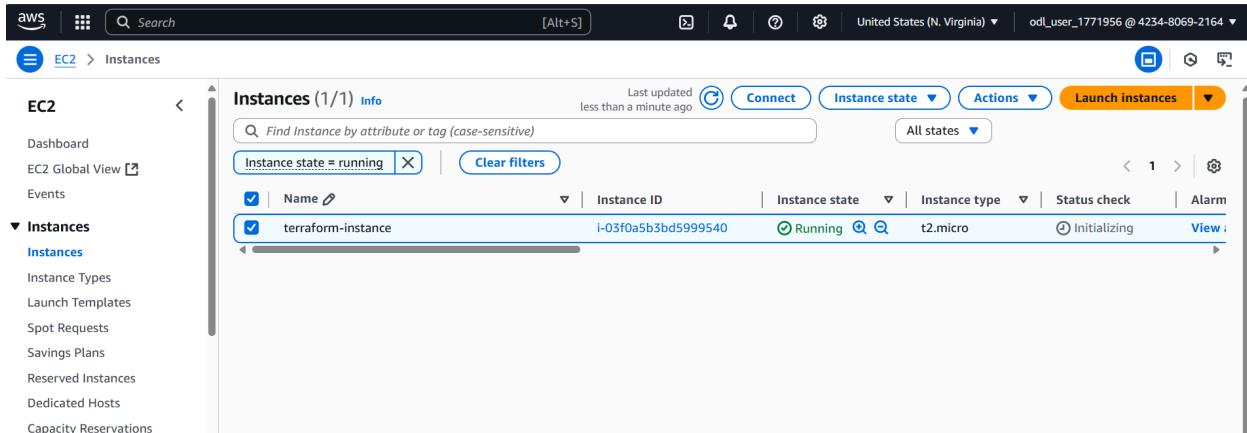
resource "aws_instance" "myec2" {

  ami      = data.aws_ami.myami.id
  instance_type = "t2.micro"
  key_name = "web-key"
  subnet_id = aws_subnet.sl-subnet.id
  security_groups = [aws_security_group.sl-sg.id]
  tags = {
    Name = "terraform-instance"
  }

}
```

Save the file

```
# terraform apply --auto-approve
```



Step 5: Ansible Setup for running the playbook on above created VM

Generate ssh keys for root user on Lab terminal(Ansible Controller)

ssh-keygen

cat /root/.ssh/id_rsa.pub

Copy the keys carefully

GO to AWS EC2 server - Ansible worker

cd .ssh

echo "GIVE YOUR SSH PUBLIC KEY" >> ~/.ssh/authorized_keys

Go back to the lab terminal.

Step 6: Create Ansible Inventory file with IP address of AWS EC2 server

In same directory where the terraform code is create the ansible inventory

vim myinventory

```
[webserver]  
54.162.0.24
```

Save the file

pwd

Copy the path of the directory

vim ansible.cfg

```
[defaults]  
inventory = /root/myproject/myinventory
```

Save the file

Validate the setup :

```
# ansible webserver -m ping
```

Step 7: Write the playbook

```
# vim playbook.yml
```

```
- name: run playbook using terraform
  hosts: webserver
  become: true
  tasks:
    - name: Install python3
      package: name=python3 state=present
    - name: Install maven
      package: name=maven state=present
    - name: Create a file
      file: path=/tmp/ansible.txt state=touch
```

Save the file

Step 8: Write Terraform code to run the playbook:

```
# vim runplaybook.tf

resource "null_resource" "run_playbook" {

  provisioner "local-exec" {
    command = "ansible-playbook playbook.yml"
  }
}
```

```
}
```

Save the file

Step 9: Execute the playbook using terraform command

```
# terraform init

# terraform apply -target null_resource.run_playbook

null_resource.run_playbook (local-exec): interpreter could change the meaning of that path. See
null_resource.run_playbook (local-exec): https://docs.ansible.com/ansible-
null_resource.run_playbook (local-exec): core/2.16/reference_appendices/interpreter_discovery.html for more information.
null_resource.run_playbook (local-exec): ok: [54.162.0.24]

null_resource.run_playbook (local-exec): TASK [Install python3] ****
null_resource.run_playbook (local-exec): ok: [54.162.0.24]

null_resource.run_playbook (local-exec): TASK [Install maven] ****
null_resource.run_playbook: Still creating... [20s elapsed]
null_resource.run_playbook (local-exec): ok: [54.162.0.24]

null_resource.run_playbook (local-exec): TASK [Create a file] ****
null_resource.run_playbook (local-exec): changed: [54.162.0.24]

null_resource.run_playbook (local-exec): PLAY RECAP ****
null_resource.run_playbook (local-exec): 54.162.0.24 : ok=4    changed=1    unreachable=0    failed=0
    skipped=0   rescued=0   ignored=0

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
root@ip-172-31-30-167:~/myproject# ls
ansible.cfg  myinventory  runplaybook.tf  terraform.tfstate          web-key.pem
main.tf       playbook.yml  sshkey.tf     terraform.tfstate.backup
root@ip-172-31-30-167:~/myproject# cat runplaybook.tf
```

```
[root@ip-10-0-1-226 .ssh]#  
[root@ip-10-0-1-226 .ssh]# python --version  
Python 2.7.18  
[root@ip-10-0-1-226 .ssh]# mvn --version  
Apache Maven 3.0.5 (Red Hat 3.0.5-17)  
Maven home: /usr/share/maven  
Java version: 17.0.15, vendor: Amazon.com Inc.  
Java home: /usr/lib/jvm/java-17-amazon-corretto.x86_64  
Default locale: en_US, platform encoding: UTF-8  
OS name: "linux", version: "5.10.237-230.949.amzn2.x86_64", arch: "amd64", family: "unix"  
[root@ip-10-0-1-226 .ssh]# cd /tmp  
[root@ip-10-0-1-226 tmp]# ls  
ansible.txt  hsperfdata_root  systemd-private-a3b52cb56e7343779d49e900ead131b-chronyd.service-TULZAK  
[root@ip-10-0-1-226 tmp]#
```