**WEEK-6**

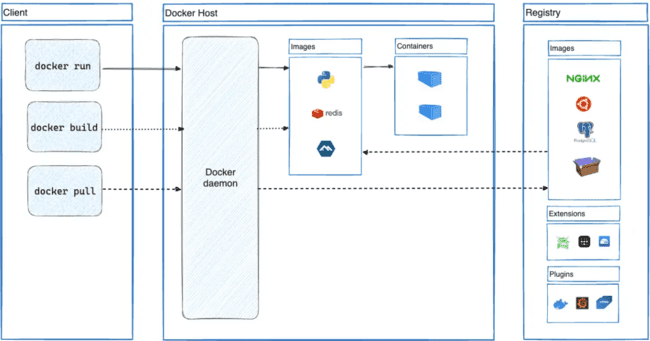
**Aim:** To understand Docker Architecture and Container Life Cycle, install Docker and execute docker

commands to manage images and interact with containers.

**Description:**

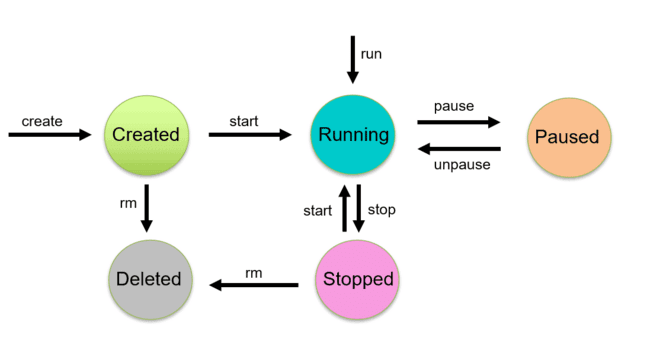
Docker is an open-source platform that simplifies building, shipping, and running applications by using containers. It streamlines the development-to-production lifecycle by allowing developers to package an application with all its dependencies, such as libraries, system tools, and code into a single, standardized unit. This process starts with a Dockerfile, a simple text file with instructions for building a Docker Image, which is the static, read-only template. When this image is run, it becomes a Docker Container, which is the live, isolated, and executable instance of the image. This container-based approach ensures the application runs uniformly and consistently across any environment, making software delivery faster and more reliable. Docker is widely used in DevOps, CI/CD pipelines, microservices, and cloud deployments.

**Docker Architecture**



The Docker architecture is based on a Client, Docker Host, and Registry model. The Client is where commands are issued, such as docker build (which tells the daemon to build an Image), docker pull (which tells the daemon to fetch an Image from the Registry), and docker run (which tells the daemon to use an Image to start a Container). The Docker Host runs the Docker daemon, the background service that manages images and containers. The daemon uses locally stored Images (like Python, Redis) to create and run Containers. The Registry (like Docker Hub) is a remote service that stores and distributes Images (like NGINX, Ubuntu) for the daemon to pull.

**Container Life Cycle**



TheContainer Life Cycle explains the different stages a container goes through from start to finish:

**Created → Running → Paused → Unpaused → Stopped → Restarted → Removed**.

A container is first created from an image, then it is started and enters the running state where it performs its tasks. If needed, it can be paused to temporarily freeze all processes, or stopped to end execution. A stopped container can be started again, and once it is no longer needed, it can be permanently removed. Docker provides simple lifecycle commands (run, pause, unpause, stop, start, rm) that make container management fast, flexible, and predictable compared to traditional environments.

**Procedure:**

**Step 1: Install Docker**

#Update system packages to prepare for Docker installation

sudo apt update

#Install the Docker engine

sudo apt install docker.io -y

#Verify Docker installation

sudo docker --version

**Step 2: Manage Images (Pull & Build)**

**#Pull image**: Download the nginx image from Docker Hub

sudo docker pull nginx:alpine

#**Build image**: Create a folder for building a custom image

mkdir my-nginx-test && cd my-nginx-test

#Create a Dockerfile using the base nginx image

echo "FROM nginx:alpine" > Dockerfile

#Build the image with a custom name

sudo docker build -t my-custom-image .

**#**Display all downloaded and built images

sudo docker images

**Step 3: Run and Interact with Containers**

**#Run container**: Create and run a container from the custom image in the background

sudo docker run --name my-webserver -d my-custom-image

**#**View the list of actively running containers

sudo docker ps

**#**Run a command inside the running container

sudo docker exec my-webserver ls

**Step 4: Container Life Cycle Commands**

**#Pause container**: Temporarily freeze the running container

sudo docker pause my-webserver

sudo docker ps

**#Unpause container**: Resume the paused container

sudo docker unpause my-webserver

sudo docker ps

**#Stop container**: Stop the running container

sudo docker stop my-webserver

sudo docker ps -a

**#Start container**: Restart a stopped container

sudo docker start my-webserver

sudo docker ps

**#Remove container**: First stop the running container and remove it permanently

sudo docker stop my-webserver

sudo docker rm my-webserver

sudo docker ps -a

**Step 5: Clean Up Images**

#Remove custom image

sudo docker rmi my-custom-image

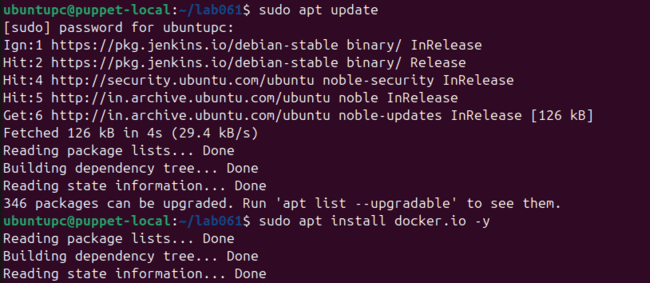
#Remove nginx base image

sudo docker rmi nginx:alpine

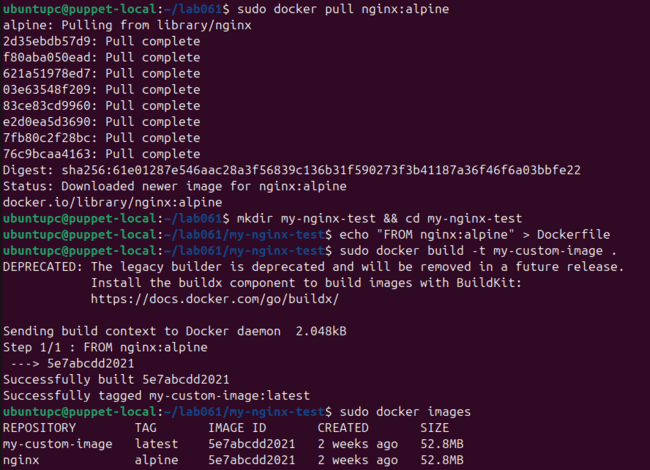
#Go back to home directory

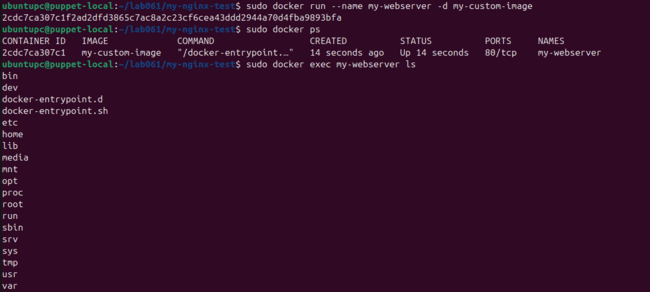
cd ..

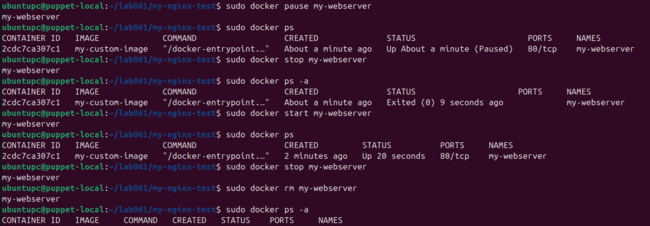
**Output:**

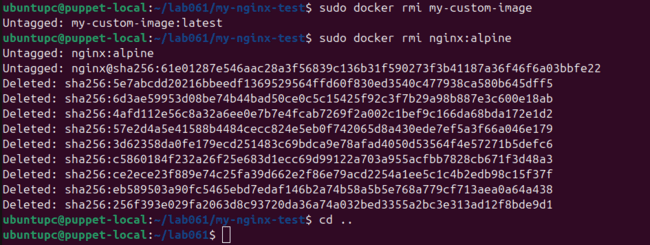
****

****

****

****

****

****

**Output Analysis:**

In this experiment, we successfully installed Docker and confirmed it using version and service commands. We downloaded an image from Docker Hub, created our own custom image through a Dockerfile, and verified both using docker images. Next, we created and ran a container from our image and confirmed its running state through docker ps. We interacted with the container using commands like docker exec, proving that we can run commands inside a live container. Then, we tested the full container life cycle by pausing, unpausing, stopping, starting, and finally removing it, while verifying each state change through appropriate Docker commands. In the end, we deleted the images and cleaned up the system, confirming that Docker gives full control over image and container management with simple lifecycle commands.

**Conclusion:**

This experiment helped us understand Docker architecture and the complete container life cycle in a practical way. We learned how to install Docker, pull and build images, create and run containers, and manage their states from start to removal. By executing multiple Docker commands, we observed how easily containers can be controlled, reused, and cleaned up compared to traditional environments. Overall, this experiment clearly demonstrated that Docker is a powerful DevOps tool that simplifies application deployment, ensures consistency across systems, and makes environment management faster, lightweight, and highly efficient.

**WEEK-7**

**Aim:** To learn Docker file instructions, build an image for a sample web application using Docker file.

**Description:**

Docker is an open-source platform that simplifies building, shipping, and running applications by using containers. It streamlines the development-to-production lifecycle by allowing developers to package an application with all its dependencies, such as libraries, system tools, and code into a single, standardized unit.

**Dockerfile** is a plain text file containing a list of instructions for building a Docker image.

**Dockerfile Instruction** is a single command written in a Dockerfile that tells Docker what to do during the image build process.

Common instructions include:

* **FROM**: Specifies the starting point or "base image."
* **WORKDIR**: Sets the working directory inside the container.
* **COPY**: Copies files from the host to the container.
* **RUN**: Executes commands inside the container.
* **EXPOSE**: Documents which port the application will use.
* **CMD**: Default command to run the container application.

**Procedure:**

**Step 1: Install Docker**

#Update system packages to prepare for Docker installation

sudo apt update

#Install the Docker engine

sudo apt install docker.io -y

#Verify Docker installation

sudo docker --version

**Step 2: Create Project Files**

#Create project directory and navigate into it

mkdir myapp

cd myapp

#Create Flask application file app.py and write the python code

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def home():

return "Hello from Dockerized Flask app!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

#Create requirements.txt file for Python dependencies

flask

**Step 3: Create Dockerfile**

#Create Dockerfile (No File Extension) to define image build instructions

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["python", "app.py"]

**Step 4: Build Docker Image**

#Build an image from the Dockerfile

sudo docker build -t flask-docker-sample .

**Step 5: Run and Test the Container**

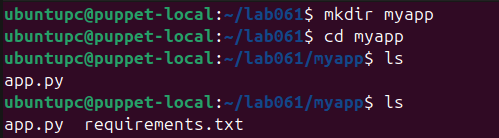
#Run the container and expose it on port 5000

sudo docker run -p 5000:5000 flask-docker-sample

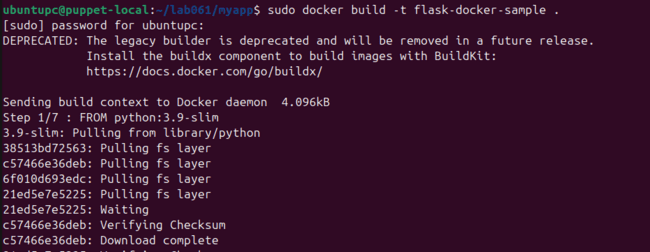
#Open your web browser and go to this address:

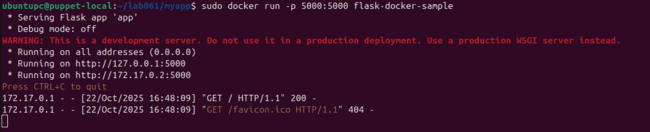
http://localhost:5000

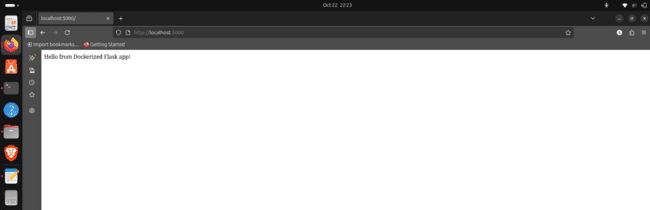
**Output:**

****

****

****

****

****

**Output Analysis:**

In this experiment, we created a sample Flask application and wrote a Dockerfile with instructions such as FROM, WORKDIR, COPY, RUN, EXPOSE, and CMD. Then we built a Docker image and verified it was created successfully. Next, we ran a container from that image and mapped port 5000 so the application could be accessed from a browser. When we opened http://localhost:5000, we saw the message “Hello from Dockerized Flask app!”, confirming that the Flask app was successfully containerized. This output proved that our Dockerfile worked correctly and that Docker can automatically build and run applications from simple instructions.

**Conclusion:**

This experiment helped us understand how Dockerfile instructions help automate the creation of application images, making containerization repeatable and error-free. By writing a Dockerfile, building the image, and running it as a container, we experienced how easily Docker can package and deliver a web application. This experiment clearly demonstrated that Dockerfiles play a key role in DevOps by simplifying builds, ensuring consistency, and speeding up deployments.

**WEEK-8**

**Aim:** To deploy a containerized application on Kubernetes cluster.

**Description:**

This experiment demonstrates deploying a simple containerized web application onto a local Kubernetes cluster. Kubernetes is an open-source container orchestration system that automates the deployment, scaling, and management of applications. It groups containers into "pods" and schedules them across a cluster, handling tasks like load balancing, self-healing, and resource allocation to ensure applications run reliably and efficiently. The setup involves installing Docker (the container runtime), Minikube (to create the local cluster), and kubectl (the command-line tool). A basic index.html file is created and then packaged into an Nginx container image using a Dockerfile. Two essential YAML configuration files are written: a Deployment to define the desired state and a Service to expose the application to the network. The application is deployed using kubectl apply and finally accessed in a browser using the minikube service command, verifying the successful deployment.

**Procedure:**

**Step 1: Install Required Tools**

**#Install Docker**

curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh get-docker.sh

sudo usermod -aG docker $USER

newgrp docker

docker run hello-world

**#Install kubectl**

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x ./kubectl

sudo mkdir -p /usr/local/bin

sudo mv ./kubectl /usr/local/bin/kubectl

kubectl version --client

**#Install Minikube**

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

minikube version

**Step 2: Start Your Kubernetes Cluster**

#Start your local Kubernetes cluster using Docker as the driver

minikube start --driver docker

#Verify that your Minikube cluster is running

minikube status

**Step 3: Create Your Application Files**

#Make a new folder for your project and move into it

mkdir html-k8s && cd html-k8s

#Create the 'index.html' file with your website content

cat > index.html << EOF

<html>

<head><title>K8s HTML Page</title> </head>

<body>

<h1>Hello from Kubernetes on Linux!</h1>

<p>Served by Nginx inside a container.</p>

</body>

</html>

EOF

#Create the 'Dockerfile' to define how to build your container

cat > Dockerfile << EOF

FROM nginx:alpine

COPY index.html /usr/share/nginx/html/index.html

EOF

**Step 4: Build the Docker Image**

#Point your terminal to use the Docker environment inside Minikube

eval $(minikube docker-env)

#Build the container image from your Dockerfile

docker build -t html-k8s-page:v1 .

**Step 5: Create Kubernetes Configuration Files**

#Create the 'deployment.yaml' file to tell Kubernetes how to run your image

cat > deployment.yaml << EOF

apiVersion: apps/v1

kind: Deployment

metadata:

name: html-page

spec:

replicas: 2

selector:

matchLabels:

app: html-page

template:

metadata:

labels:

app: html-page

spec:

containers:

- name: nginx

image: html-k8s-page:v1

imagePullPolicy: IfNotPresent

ports:

- containerPort: 80

EOF

#Create the 'service.yaml' file to expose your application

cat > service.yaml << EOF

apiVersion: v1

kind: Service

metadata:

name: html-page-service

spec:

selector:

app: html-page

ports:

- protocol: TCP

port: 80

targetPort: 80

type: NodePort

EOF

**Step 6: Deploy Your Application**

#Apply the deployment and service configurations to the cluster

kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

**Step 7: Verify the Deployment**

#Check the status of your application pods to see if they are 'Running'

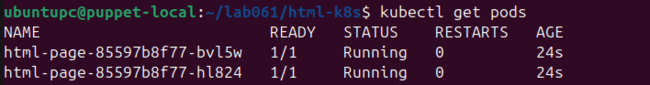
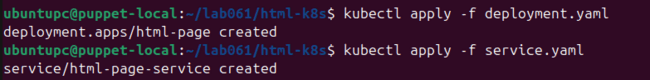
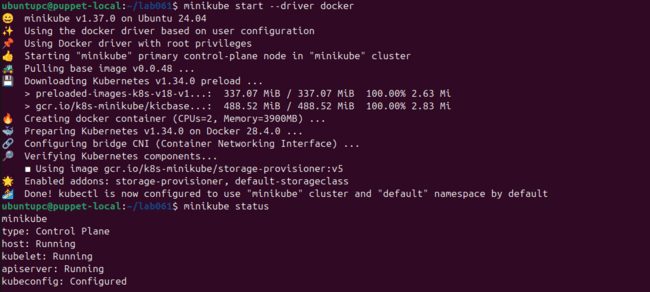
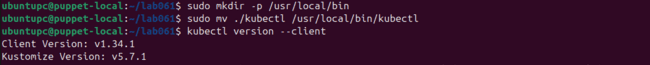
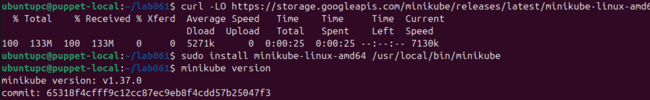
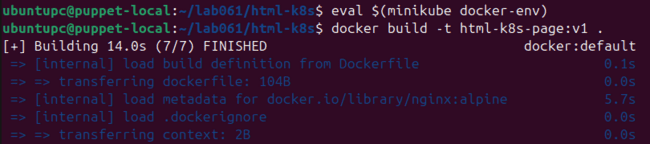
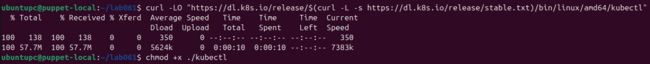
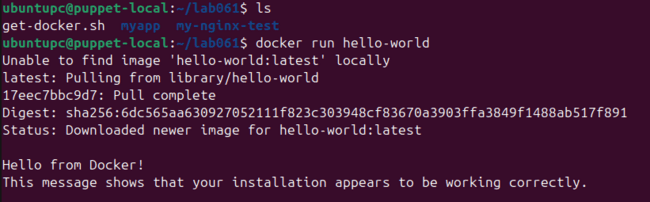
kubectl get pods

**Step 8: Access Your Application**

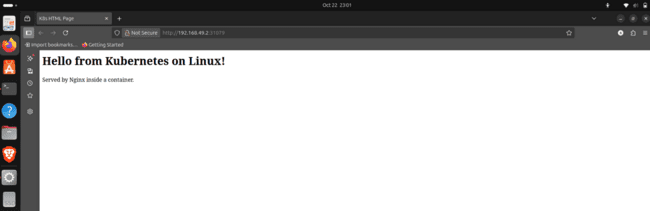
#Ask Minikube to open your new service in a web browser

minikube service html-page-service

**Output:**

****

****

****

**Output Analysis:**

In this experiment, the outputs confirmed every tool was installed correctly, including Docker, Minikube, and kubectl. The minikube status output showed the cluster was "Running" , and the docker build output showed the Nginx image was created. The kubectl get pods command verified that two application pods were "Running" , and the kubectl get svc command showed the service was active. Finally, the minikube service command successfully opened the application in a browser, which displayed the "Hello from Kubernetes on Linux!" message, proving the deployment worked.

**Conclusion:**

This experiment helped us understand the workflow of deploying a containerized application on a local Kubernetes cluster. We successfully used Minikube to create a single-node cluster and kubectl to interact with it. The process involved writing a Dockerfile to package a simple HTML web page into an Nginx container, and then building that image directly within Minikube's Docker environment. By writing and applying a Deployment manifest, we learned how Kubernetes automates the creation and management of our application's pods to maintain a desired number of replicas. Furthermore, creating a **Service** manifest (type NodePort) demonstrated how to expose those pods and make the application accessible from outside the cluster. Finally, accessing the web page in the browser confirmed the entire end-to-end orchestration was successful.

**WEEK-9**

**Aim:** To install and configure Pull based Software Configuration Management and provisioning tools using

Puppet.

**Description:**

Puppet is an open-source configuration management tool that automates provisioning and managing systems to keep them in a desired state. It uses a declarative language, meaning you define the desired end state of your system, not the specific steps to get there.

This experiment sets up a local environment where your machine acts as both server and client. The server role is the Puppet Master, the central controller that stores configuration files (called manifests) and compiles them into a catalog defining the desired state for a client. The client role is the Puppet Agent, a service running on the managed machine that sends facts about itself to the Master and requests this configuration catalog. This setup demonstrates a pull-based configuration, as the Agent initiates contact with the Master to get its certificate signed and "pull" its instructions. The Agent then compares the catalog to its current state and applies the required changes. In real life, a system administrator uses a Puppet Master to define and enforce consistent configurations (like software versions) across hundreds of servers, each of which runs a Puppet Agent to automatically apply those settings.

**Procedure:**

**#**Find your Ubuntu system version by running the command: lsb\_release -a

**Step 0: The Cleanup** *(Optional: Only needed if cleaning a previous install)*

sudo systemctl stop puppetserver puppet

sudo apt-get purge puppetserver puppet-agent puppet7-release -y

sudo apt-get autoremove -y

sudo rm -rf /etc/puppetlabs/

rm -f ~/puppet7-release-\*.deb

**Step 1: Fresh Setup and Installation**

#Create and enter your new directory

mkdir sohanLab

cd sohanLab

#Download the correct Puppet 7 installer for Ubuntu 20.04 (focal)

wget <https://apt.puppet.com/puppet7-release-focal.deb>

#Install the software

sudo dpkg -i puppet7-release-focal.deb

sudo apt-get update

sudo apt-get install puppetserver puppet-agent -y

**Step 2: One-Time Configuration**

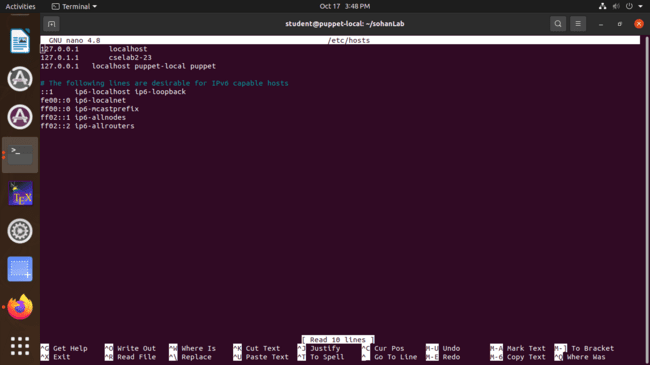
#Change your computer's name to 'puppet-local'

sudo hostnamectl set-hostname puppet-local

#Open the 'hosts' file to link your hostname to your local IP

sudo nano /etc/hosts

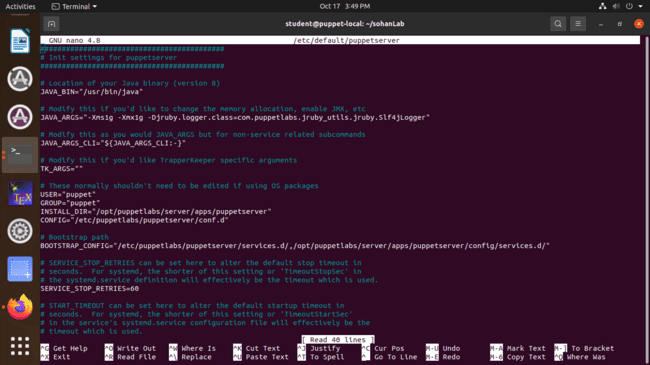
(At very top of the file add, **127.0.0.1 localhost puppet-local puppet**)

****

#Reduce Puppet Server’s memory usage

sudo nano /etc/default/puppetserver

(Inside the file, change JAVA\_ARGS to 1g, instead of 2g)

****

**Step 3: The Final Execution**

**(Following three commands are optional on new system)**

sudo systemctl stop puppetserver puppet

sudo rm -rf /etc/puppetlabs/puppet/ssl

sudo rm -rf /etc/puppetlabs/puppetserver/ca

#Generate a new, clean Certificate Authority for the master to sign requests

sudo /opt/puppetlabs/bin/puppetserver ca setup

#Start the main Puppet master server

sudo systemctl start puppetserver

#Start the local Puppet agent service

sudo systemctl start puppet

#Open a new, blank file in the text editor to create your Puppet instruction file (manifest)

sudo nano /etc/puppetlabs/code/environments/production/manifests/site.pp

(Inside this file, you add the code to create /tmp/sohan\_lab\_test.txt)

node 'puppet-local' {

file { '/tmp/sohan\_lab\_test.txt':

ensure => file,

content => "This file was created by a local Puppet Master and Agent!\n",

mode => '0644',

}

}

**#######Type this command in a new “AGENT” terminal**

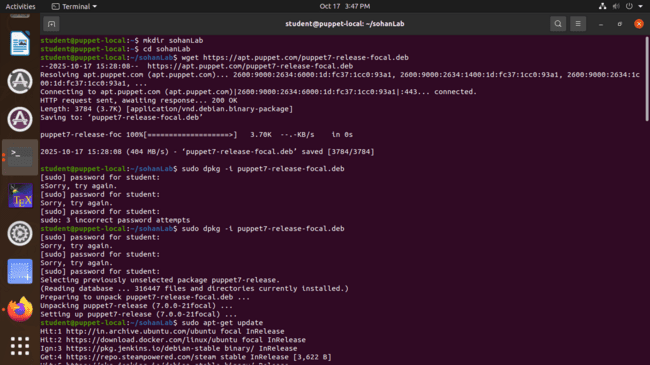
#Tell the agent to check in with the master, get its certificate, and apply its configuration

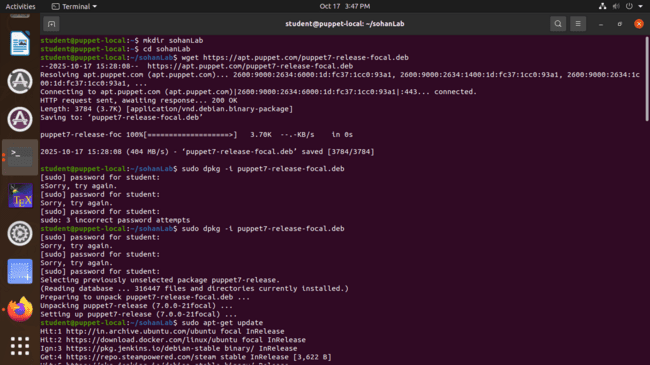
sudo /opt/puppetlabs/bin/puppet agent -t

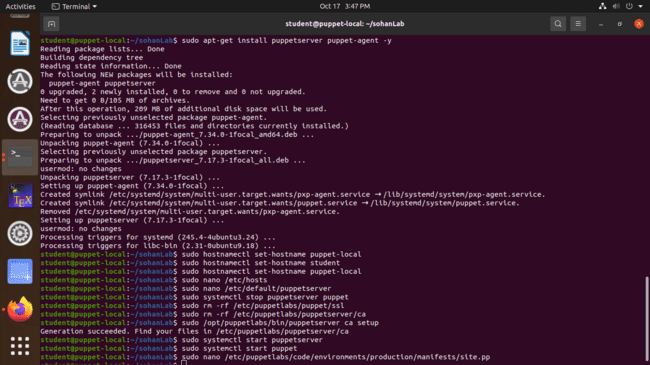
**#(ANY TERMINAL)** Display the contents of the file that Puppet was supposed to create

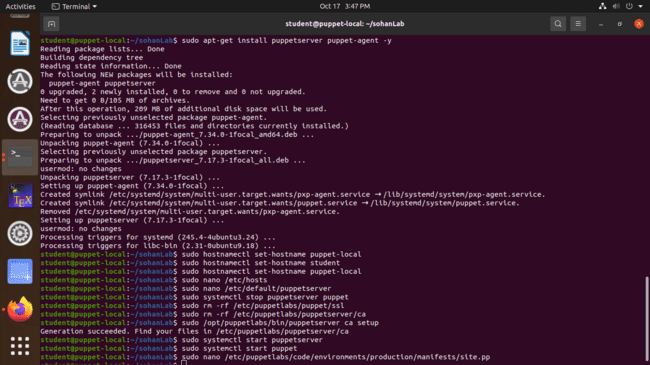
cat /tmp/sohan\_lab\_test.txt

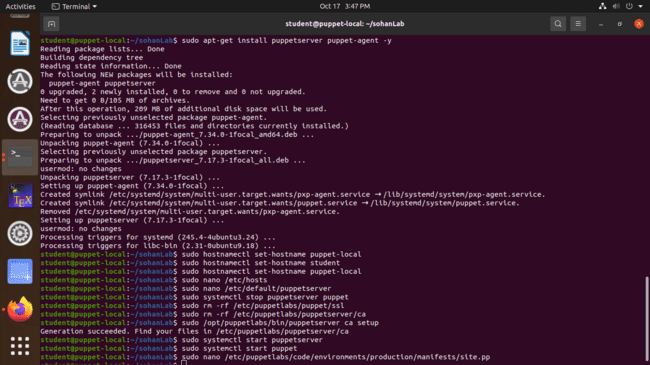
**Output:**

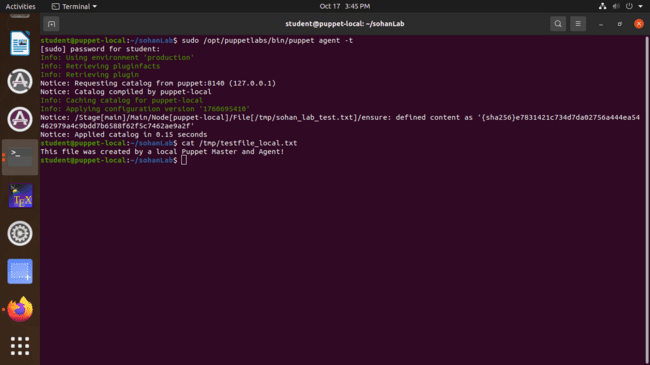
****

****

****

****

****

****

**Output Analysis:**

In this experiment, the final output was the successful creation of the file /tmp/sohan\_lab\_test.txt. When we used the cat command to view its contents, it displayed "This file was created by a local Puppet Mater and Agent!" , which perfectly matched the content defined in our Puppet manifest. This result confirms that the entire workflow was successful: the Agent ran, requested its configuration, received the catalog from the Master, and correctly applied the changes to the system to match the desired state.

**Conclusion:**

This experiment helped us successfully install and configure a complete pull-based configuration management system using Puppet on a single machine. We learned how to set up both a Puppet Master and a Puppet Agent, and how critical it is to properly configure the local environment so they can find each other. We also learned how to define a desired state using a manifest file and how the Master uses a Certificate Authority to securely sign the Agent's certificate request. By running the agent, we successfully demonstrated its ability to automatically "pull" the configuration from the Master and apply it.

**WEEK-10**

**Aim:** To learn Software Configuration Management and provisioning using Puppet Blocks(Manifest, Modules, Classes, Function)

**Description:**

Puppet is an open-source configuration management tool that automates provisioning and managing systems to keep them in a desired state.

Its code is organized into blocks:

* Manifest: A .pp file defining system state
* Module: A folder structure organizing related classes and functions
* Class: A container of resources for a specific configuration
* Function: A reusable block of logic to perform tasks or display messages.

This experiment demonstrates Software Configuration Management using Puppet by automating the setup of an Apache web server. It uses a Module to organize a Class named webserver. This class, defined in a Manifest file, contains resources to install the 'apache2' package, run the 'apache2' service, and create an index.html file. A custom Function is also defined to print a success message when the configuration is applied. Applying the main site.pp manifest causes Puppet to automatically build the server, which is then verified by checking the webpage and service status.

**Procedure:**

**Step 1: Install Puppet**

**#**Find your Ubuntu system version by running the command: lsb\_release -a

wget https://apt.puppet.com/puppet8-release-noble.deb

sudo dpkg -i puppet8-release-noble.deb

sudo apt-get update

sudo apt-get install puppet-agent

echo 'export PATH=$PATH:/opt/puppetlabs/bin' >> ~/.bashrc

source ~/.bashrc

puppet --version

**Step 2: Create the Project Structure**

#Go to your home directory

cd ~

#Create the module's manifests directory

mkdir -p ~/puppet\_experiment/modules/webserver/manifests

#Create the module's custom functions directory

mkdir -p ~/puppet\_experiment/modules/webserver/lib/puppet/functions/webserver

#Move into the main project directory

cd ~/puppet\_experiment

**Step 3: Define the Class (Manifest)**

#Open the class manifest file init.pp in the nano editor

nano modules/webserver/manifests/init.pp

# init.pp

# Purpose: Define webserver class to install and configure Apache

class webserver {

# Install Apache

package { 'apache2':

ensure => installed,

}

# Start and enable Apache service

service { 'apache2':

ensure => running,

enable => true,

}

# Create a sample HTML page

file { '/var/www/html/index.html':

ensure => file,

content => '<h1>Welcome to Puppet Web Server</h1>',

}

# Call a function to display a notification

notify { webserver::greet('System Administrator'): }

}

**Step 4: Create the Function**

#Open the Ruby function file greet.rb in the nano editor

nano modules/webserver/lib/puppet/functions/webserver/greet.rb

Puppet::Functions.create\_function(:'webserver::greet') do

def greet(role)

"Hello, #{role}! Puppet has successfully applied the webserver configuration."

end

end

**Step 5: Create the Main Manifest (site.pp)**

#Open the main site.pp manifest file in the nano editor

nano site.pp

# site.pp

include webserver

**Step 6: Apply the Manifest**

#Apply the configuration using the full path to Puppet

sudo /opt/puppetlabs/bin/puppet apply site.pp --modulepath=./modules

**Step 7: Verify the Configuration**

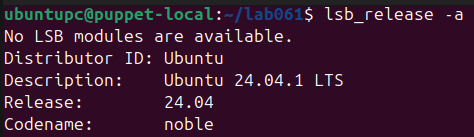
#Display the content of the web page created by Puppet

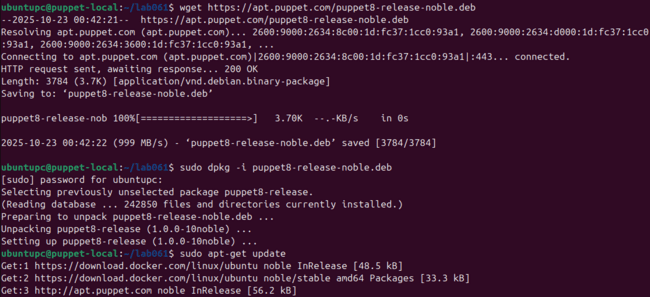
cat /var/www/html/index.html

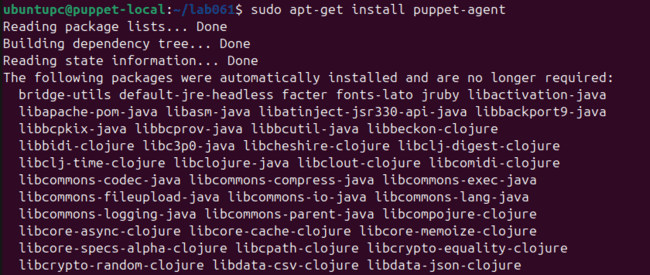
#Check the running status of the Apache service

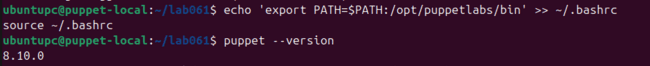
sudo systemctl status apache2

**Output:**

****

****

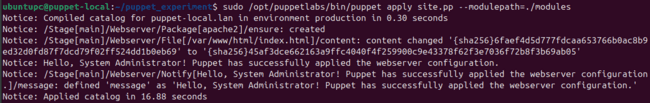
****

****

****

****

****

****

****

**Output Analysis:**

In this experiment, the output from the puppet apply command confirms the successful execution of the manifest. It shows the index.html file's content being changed and the custom notify message being triggered and displayed as "Hello, System Administrator!..." . The subsequent verification steps confirm this success, as cat displays the correct HTML content and ps aux lists the running apache2 processes, proving that Puppet automatically configured the system to the desired state.

**Conclusion:**

This experiment helped us understand Software Configuration Management by practically using Puppet's core building blocks: Manifests, Modules, Classes, and Functions. We successfully demonstrated how these components work together to automate the provisioning of an Apache web server. The process confirmed that Puppet can efficiently apply a defined state, enforce consistency, and reduce manual effort, highlighting its power in automating infrastructure management.