**Linux/Shell Scripting**

**Baic Commands in Linux:**

Pwd ---- print the present working directory.

Touch ---- creating a new files.

Cat ----- reading the files(display the content in the files)

Ls ----- list out the files.

Ls -l ----- list out the files with brief info about the files(more details)

Ls -a ---- list out the hidden files also(. Files)

Ls -la ----- list out the files -hiden files and more info about the files

Echo ----- it prints what we gave

Ex:- echo “my name is teja” > file1

Here we create and save a file with file1 with the content

echo “my friend name is siva” >>file1

now here we update the file1 with >>.

Vi / vim ------- edit the files

Nano ----- used to edit the files.

Date ---- print the todays date.

Cal --- print present month and highlight the todays date.

Cal -y ---- print all months and highlight the todays date.

Cal 1 2023---- print the Jan month

Cal -A 1 12 2022 ----- print one month after 12th month of 2022 yr

Cal -B 1 12 2022 ----- before

Clear ----- clear the terminal

Man ------ it is useful to documentation and user manuals

**Navigating the files using commands:**

cd ---- change the directory

cd .. ----- change the directory to previous (go back to the one directory)

mkdir name ---- create a emtry directory

rmdir name ----- remove a empty directory

rm -r filename ------- command to remove non-empty directories.

touch name ----- creating a file.

rm name ------- remove the files.

cp name name1 ----- copy the files from name to name1

mv name name1 ------ move the files from name to name / it is also used to renaming

**Searching:**

Grep word file1 ---- it is used to search the word in a files. Ex:- grep teja.txt

Egrep “i.g” teja.txt

--🡪we use the meta characters to find the exact words

Egrep “i+g” teja.txt

Egrep “i\*g” teja.txt

Egrep “s{2}” teja.txt

🡪”^” –- caret –represents the start of the line

🡪”$” --- doller—represents the end of the line

Egrep “^m” teja.txt

Egrep “$m” teja.txt

Passwd ----- to change the password

Diff ------ it is used to difference b/w the files

Find command🡪find the location of the file or group of files

Ex: find . -name “teja.txt”

Find . -name “\*.txt”

History command ------ get the history

Head -n 5 ~/.bash\_history

Tail -n 5 ~/.bash\_history

**Saving the results to a file:**

We used the command to automatically save the list or anything to a new file..like

Ls > ST ------- saves the list of the particular directory to ST file

Pwd > ST ------- save the present working directory of a directory to ST file

Pwd >> ST -------- now update the ST file only

**File Permissions:**

* rwx rw- r-- 🡪 this is the file
* drwx xr- x-- 🡪 this is the directory

u 🡪 user

g 🡪 group

o 🡪 other peoples

if we do--------------

chmod o+w ST 🡪 u ll give the permission (write) to the other peoples

chmod g+x ST 🡪 u ll give the permissions(execute) to the group

chmod g-w ST 🡪 remove the permission(write) to the group

And:-

4 ----- stands for “read”

2 ----- stands for “write”

1 ----- stands for “executable”

0 ----- stands for “no permissions”

Ex:

$ Chmod 754 ST.txt

Here 7,5,4 represents the individual permissions for user ,group and other peoples

7 --------- 4+2+1=7 🡪 all permissions (rwx)

5 --------- 4+1=5 🡪 only read and excecutable permissionsv(rx)

4 --------- 4=4 🡪 only read permissionsv (r)

**Processers:**

Ps ----- ps command is used to list the currently running processes and their PIDs(Process Identifier) along with some other information depends on different options

Ps ax -------- it is used to list the all processes

**Pipeline:**

$ date | cut --delimiter=” “ –field=1

* it prints the 1st field of the date and it was separated by space

$ date | tee teja.txt | cut –delimiter=”:” -filed=2

**Shell Scripting:**

**Shell:**A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Ex: hello.sh ----shell script save with .sh extension.

#!/bin/bash 🡪 it is called **Shebang** Constructor.

Date

echo $date

NAME="sriteja"

echo $NAME

pwd

cal

Ex2: teja.sh 🡪 input and output command

#!/bin/bash

echo "what is ur name"

read PERSON

echo " hi, $PERSON"

**Using Arrays:**

If you are using the **bash** shell, here is the syntax of array initialization −

array\_name=(value1 ... valuen)

## Accessing Array Values

After you have set any array variable, you access it as follows −

${array\_name[index]}

Here *array\_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept −

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Index: ${NAME[0]}"

echo "Second Index: ${NAME[1]}"

The above example will generate the following result –

$./test.sh

First Index: Zara

Second Index: Qadir

**Using operators:**

We will now discuss the following operators −

* Arithmetic Operators
* Relational Operators
* Boolean Operators
* String Operators
* File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

Ex:- print the sum of two numbers🡪 operators.sh

#!/bin/bash

Sum=’expr 2 + 4’

Echo “Total Sum = $sum”

Output:-

Total Sum = 6

The following points need to be considered while adding -

* There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
* The complete expression should be enclosed between **‘ ‘**, called the backtick.

A=10 and B=20

**Arthemetic Operator:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

**Relational Operator:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a -eq $b ] is not true. |
| **-ne** | Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. | [ $a -ne $b ] is true. |
| **-gt** | Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true. | [ $a -gt $b ] is not true. |
| **-lt** | Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true. | [ $a -lt $b ] is true. |
| **-ge** | Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -ge $b ] is not true. |
| **-le** | Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -le $b ] is true. |

**Boolean Operators:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **!** | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| **-o** | This is logical **OR**. If one of the operands is true, then the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

**Conditional Statements:**

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

* The **if...else** statement
* The **case...esac** statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement −

* [if...fi statement](https://www.tutorialspoint.com/unix/if-fi-statement.htm)
* [if...else...fi statement](https://www.tutorialspoint.com/unix/if-else-statement.htm)
* [if...elif...else...fi statement](https://www.tutorialspoint.com/unix/if-elif-statement.htm)

**GIT**

Version Control System

**Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS −

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

**Git Fetch** is the command that tells the local repository that there are changes available in the remote repository without bringing the changes into the local repository.

**Git Pull** on the other hand brings the copy of the remote directory changes into the local repository

**staging area** can be described as **a preview of your next commit**. When you create a git commit, Git takes changes that are in the staging area and make them as a new commit. You are allowed to add and remove changes from the staging area. The staging area can be considered as a real area where git stores the changes.

**How to Update Git Repo?**

1. Step 1: Copy Git Remote Repo URL. First, open the GitHub host service and go to the remote branch tab. ...
2. Step 2: Launch Git Bash. Next, open up the “Git Bash” utilizing the “Startup” menu:
3. Step 3: Clone Git Remote Repo. ...
4. Step 4: Add Remote Repo. ...
5. Step 5: Update Git Repo.

## ****How to Update Git Repo?****

To update the repository, first, we will clone the Git remote repository on Git local repository and then add an upstream remote with the given URL through the “**$ git remote add upstream <remote-URL>**” command.

Next, fetch and download the content from Git remote repository using the “**$ git pull upstream <branch>**” command.

It will instantly update the Git repository.

**Git Commands:**

Git init ---------initializing the files

Git add -------- by this command to add the files from local to staging repository

Git Commit ------- by this command to add the files from staging to remote repository

Git push -------- it is used to push the code to remote repository

Git clone -------- fetch the code from the remote to local repository

Git branch name ------ create the new branch

Git Checkout ----------- it is used to moved from one branch to another branch

**Docker:**

**Docker** is a container management service. The keywords of Docker are **develop, ship** and **run** anywhere. The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere.

**Docker Hub** is a registry service on the cloud that allows you to download Docker images that are built by other communities. You can also upload your own Docker built images to Docker hub.we will see how to download and the use the Jenkins Docker image from Docker hub.

This will be used to download the Jenkins image onto the local Ubuntu server.

$ docker pull jenkins

**Image:** In Docker, everything is based on **Images**. An image is a combination of a file system and parameters. Let’s take an example of the following command in Docker.

$Docker build ImageName ---- to create a image

$docker run hello-world

* The Docker command is specific and tells the Docker program on the Operating System that something needs to be done.
* The **run** command is used to mention that we want to create an instance of an image, which is then called a **container**.
* Finally, "hello-world" represents the image from which the container is made.

## Displaying Docker Images

To see the list of Docker images on the system, you can issue the following command.

docker images

## Removing Docker Images

The Docker images on the system can be removed via the **docker rmi** command. Let’s look at this command in more detail.

$ docker rmi ImageID

🡪The output will show only the Image ID’s of the images on the Docker host.

sudo docker images -q

🡪This command is used see the details of an image or container.

docker inspect Repository

**Containers:** are instances of Docker images that can be run using the Docker **run** command. The basic purpose of Docker is to run containers. Let’s discuss how to work with containers.

Docker ps ------ it shows the all the running containers available in system

Docker ps -a ------- it shows the all the containers available in ur system.

## docker history

With this command, you can see all the commands that were run with an image via a container.

### **Syntax**

$docker history ImageID

* **ImageID** − This is the Image ID for which you want to see all the commands that were run against it.

## docker top

With this command, you can see the top processes within a container.

### **Syntax**

$docker top ContainerID

* **ContainerID** − This is the Container ID for which you want to see the top processes.

## docker stop

This command is used to stop a running container.

### **Syntax**

docker stop ContainerID

* **ContainerID** − This is the Container ID which needs to be stopped.

## docker rm

This command is used to delete a container.

### **Syntax**

docker rm ContainerID

## docker kill

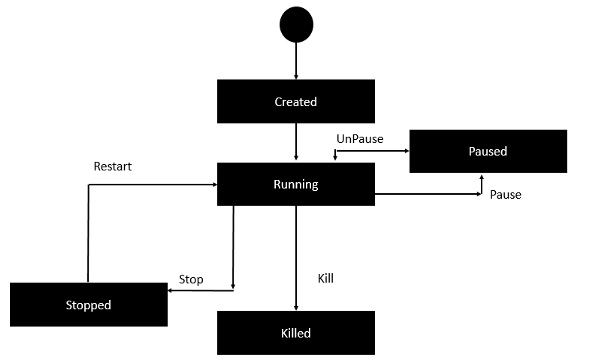
This command is used to kill the processes in a running container.

### **Syntax**

docker kill ContainerID

Docker – Container Lifecycle

The following illustration explains the entire lifecycle of a Docker container.



* Initially, the Docker container will be in the **created** state.
* Then the Docker container goes into the running state when the Docker **run** command is used.
* The Docker **kill** command is used to kill an existing Docker container.
* The Docker **pause** command is used to pause an existing Docker container.
* The Docker **stop** command is used to pause an existing Docker container.
* The Docker **run** command is used to put a container back from a **stopped** state to a **running** state.

**Creating a Docker Image for your Application**

1. Write a Dockerfile for your application.
2. Build the image with docker build command.
3. Host your Docker image on a registry.
4. Pull and run the image on the target machine.

**Dockerfile:** A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

Docker also gives you the capability to create your own Docker images, and it can be done with the help of **Docker Files**. A Docker File is a simple text file with instructions on how to build your images.

$ sudo vi Dockerfile (must use the Capital ‘D’)

#This is a sample Image

FROM ubuntu

MAINTAINER demousr@gmail.com

RUN apt-get update

RUN apt-get install –y nginx

CMD [“echo”,”Image created”]

\*The above code write in the ‘Dockerfile’

\*The Docker File can be built with the following command –

$ docker build

### **Syntax**

docker build -t ImageName:TagName dir .

**Change the port number:**

To change the docker port number for the jenkins image in the docker host:

sudo

after that we are using the Jenkins in the docker by using the dockerhost:8080 port number.

**Docker Commands in Dockerfile:**

* 1. CMD -used to execute the command directly using echo

CMD [“echo” , “helloworld”]

* 1. ENTRYPOINT – it is also used to print the value by using the echo

ENTRYPOINT [“echo”]

* 1. ENV -it is used to environment variables to container

ENV var1=”” ,var2=””

* 1. WORKDIR -it is used to present working directory

WORKDIR dirname

**Docker info** command

-The command will provide all relative information on the Docker component installed on the Docker Host.

## Creating a Volume

A volume can be created beforehand using the **docker** command. Let’s learn more about this command.

### **Syntax**

docker volume create –-name=volumename –-opt options

### **Options**

* **name** − This is the name of the volume which needs to be created.
* **opt** − These are options you can provide while creating the volume.

## Listing all the Volumes

You can also list all the **docker volumes** on a **docker host**. More details on this command is given below −

### **Syntax**

docker volume ls

**Docker Networks:**

**Command:** : to display the list of n/w

$ **Docker network ls**

Docker provides several networking options that allow containers to communicate with each other and with the outside world

1.**Bridge Network**: bridge network is the default networking mode in Docker

2.**Host Network**: containers share the network namespace with the host machine.

3.**Null Network**: The none network mode disables networking for a container

4.**Custom Network**: Docker also allows you to create custom networks with user-defined configurations.

5.**Overlay Network**: Overlay networks enable communication between containers running on different Docker hosts or across multiple Swarm nodes.

**Docker Volume:**

Docker, a volume is a directory or filesystem that is managed by Docker and is used to persist data generated by containers. Volumes allow you to store and share data between containers, as well as persist data even after a container is stopped or deleted. They provide a convenient way to manage and maintain data separately from the container itself

**Host-mounted volumes**: With host-mounted volumes, a directory on the host machine is mounted into the container as a volume.

**Named volumes**: Named volumes are managed by Docker and have a specific name assigned to them.

**Anonymous volumes**: Anonymous volumes are similar to named volumes, but they are not given an explicit name.

**Ansible:**

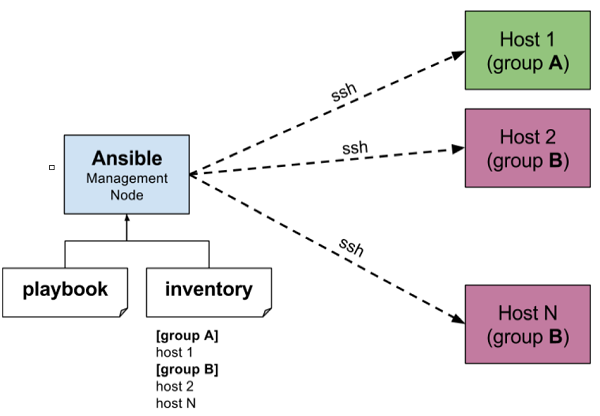
**Ansible** is simple open source IT engine which automates application deployment, intra service orchestration, cloud provisioning and many other IT tools.

Ansible is easy to deploy because it does not use any agents or custom security infrastructure.

Ansible uses playbook to describe automation jobs, and playbook uses very simple language i.e. **YAML**

The picture given below shows the working of Ansible.

**Ansible works** by connecting to your nodes and pushing out small programs, called "**Ansible** modules" to them. **Ansible** then executes these modules (over SSH by default), and removes them when finished. Your library of modules can reside on any machine, and there are no servers, daemons, or databases required.



**YAML Language:**

Ansible uses YAML syntax for expressing Ansible playbooks. This chapter provides an overview of YAML. Ansible uses YAML because it is very easy for humans to understand, read and write when compared to other data formats like XML and JSON.

Every **YAML** file optionally starts with “---” and ends with “...”.

## Understanding YAML

We learned the different ways in which the YAML data is represented.

### **key-value pair**

YAML uses simple key-value pair to represent the data. The dictionary is represented in key: value pair.

**Note** − There should be space between : and value.

**Ex:**

--- #Optional YAML start syntax

james:

name: james john

rollNo: 34

div: B

sex: male

… #Optional YAML end syntax

**Represents a list:**

---

countries:

- America

- China

- Canada

- Iceland

…

# **Ansible - Ad hoc Commands:**

Ad hoc commands are commands which can be run individually to perform quick functions. These commands need not be performed later.

For example, you have to reboot all your company servers. For this, you will run the Adhoc commands from ‘**/usr/bin/ansible**’.

These ad-hoc commands are not used for configuration management and deployment, because these commands are of one time usage.

ansible-playbook is used for configuration management and deployment.

**Ansible Playbooks:**

Playbooks are the files where Ansible code is written. Playbooks are written in YAML format. YAML stands for Yet Another Markup Language. **Playbooks** are one of the core features of Ansible and tell Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks.

Playbooks contain the steps which the user wants to execute on a particular machine. Playbooks are run sequentially. Playbooks are the building blocks for all the use cases of Ansible.

## Playbook Structure

Each playbook is an aggregation of one or more plays in it. Playbooks are structured using Plays. There can be more than one play inside a playbook.

1.start with --- and end with …

2.to write the comments with #

3.members of the list begin with –

4.key: value pairs

<key>: <values>

**Simple Playbook:**

**---**

* Name: First playbook pins test

Hosts: servers

Tasks:

- Name: ping test

ping

**…**

**Ansible Roles:**

Roles provide a framework for fully independent, or interdependent collections of variables, tasks, files, templates, and modules.

In Ansible, the role is the primary mechanism for breaking a playbook into multiple files. This simplifies writing **complex playbooks**, and it makes them easier to reuse. The breaking of playbook allows you to logically break the playbook into reusable components.

Each role is basically limited to a particular functionality or desired output, with all the necessary steps to provide that result either within that role itself or in other roles listed as dependencies.

Roles are not playbooks. Roles are small functionality which can be independently used but have to be used within playbooks. There is no way to directly execute a role. Roles have no explicit setting for which host the role will apply to.t

**Jenkins:**

**Jenkins:**

Jenkin is a java Application

Jenkins is a powerful application that allows continuous integration and continuous delivery of projects,

**Continuous Integration** is a process of integrating code changes from multiple developers in a single project many times.(Or)

CI - Continuous Integration A process where team members integrate their work continuously in a shared repository   
Can be daily or as needed Every integration or checkin in the repo is validated by automated build,automated unit or integration tests

**Continuous Delivery** is a process, where code changes are automatically built, tested, and prepared for a release to production.

**Blue Ocean:** a new user experience for Jenkins and providing an interactive view for Jenkins jobs and pipelines.

**Jenkinsfile:** Pipeline as a code. A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control.

**Pipeline:** Pipeline is a workflow of group of events or jobs that are chained and interlinked with each other.to write the pipelines we use the Groovy programming language.

**(Or)**

A Pipeline is a user defined model of a CD pipeline, A Pipelines code defines your entire build process, which typically includes stages for building an application, testing it & deliveries it.

These are two types:

1.Scripted Pipeline

2.Declarative Pipeline

**1.Scripted Pipeline:**

Node {

Def name

Stage(‘Build’) {

Echo “Building’”

}

Stage(‘Deploy) {

Echo “Deploying”

}

Stage(‘Test’) {

Echo “Testing”

}

}

**2.Declarative Pipeline:**

pipeline {

agent any

stages {

stage('Hello') {

steps {

echo 'Hello World'

}

}

stage('Build') {

steps {

echo 'Building'

}

}

stage('Deploy') {

steps {

echo 'Deploying'

}

}

stage('Test') {

steps {

echo 'Testing'

}

}

}

}\

To change port number in amazon linux machine to use: **vi /usr/lib/systemd/system/jenkins.service**

**Types of Jenkins Jobs:**

1.Freestyle Job

2.Maven

3.Pipeline

4.Externel Job

5.Multi Configuration project

6.Multi Branch pipeline

7.Folder

8.Organization folder

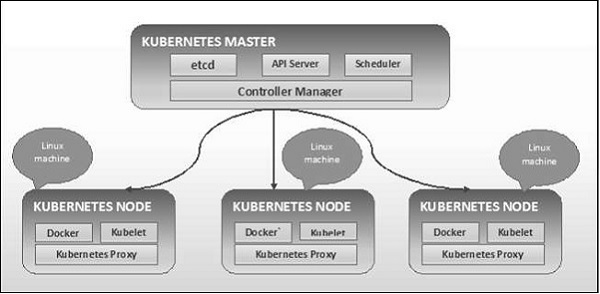
**Kubernetes:**

**Kubernetes** in an open source container management tool hosted by Cloud Native Computing Foundation (CNCF). This is also known as the enhanced version of Borg which was developed at Google to manage both long running processes and batch jobs, which was earlier handled by separate systems.

Kubernetes comes with a capability of automating deployment, scaling of application, and operations of application containers across clusters. It is capable of creating container centric infrastructure.

## Kubernetes - Cluster Architecture

As seen in the following diagram, Kubernetes follows client-server architecture. Wherein, we have master installed on one machine and the node on separate Linux machines.



## Kubernetes - Master Machine Components

### **etcd**

It stores the configuration information which can be used by each of the nodes in the cluster.

### **API Server**

Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it.

### **Controller Manager**

This component is responsible for most of the collectors that regulates the state of cluster and performs a task. In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server.

The controller manager runs different kind of controllers to handle nodes, endpoints, etc. The key controllers are replication controller, endpoint controller, namespace controller, and service account controller.

### **Scheduler**

This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload.

## Kubernetes - Node Components

### **Docker**

The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

### **Kubelet Service**

This is a small service in each node responsible for relaying information to and from control plane service. It interacts with **etcd** store to read configuration details and wright values.

### **Kubernetes Proxy Service**

This is a proxy service which runs on each node and helps in making services available to the external host.

**Node:** A node is a working machine in Kubernetes cluster which is also known as a minion. They are working units which can be physical, VM, or a cloud instance.

**Services:** A service can be defined as a logical set of pods. It can be defined as an abstraction on the top of the pod which provides a single IP address and DNS name by which pods can be accessed. With Service, it is very easy to manage load balancing configuration. It helps pods to scale very easily.

**Pod:** A pod is a collection of containers and its storage inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it

**Namespace:** Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.

**Containerization:** Kubernetes containerization is the utilization of the Kubernetes open source tool to automate the deployment, scaling, and management of containers without launching virtual machines for any applications.

**Kubernetes Image:**

Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure. As of now, we are only supporting Kubernetes to support Docker images. Each container in a pod has its Docker image running inside it.

In order to pull the image and create a container, we will run the following command.

$ kubectl create –f Tesing\_for\_Image\_pull

Once we fetch the log, we will get the output as successful.

$ kubectl log Tesing\_for\_Image\_pull

**Minikube**: a ONE Node cluster, where the master and worker processes are on the same machine. **(or)** It is a tool and it is used to run the Kubernetes clusters locally i.e Laptop

**Kubectl:** the command line tool for Kubernetes, then enables the interaction with the cluster: to create pods, services and other components.

**Kubernetes Jobs:** The main function of a job is to create one or more pod and tracks about the success of pods. They ensure that the specified number of pods are completed successfully. When a specified number of successful run of pods is completed, then the job is considered complete.

We will create the job using the following command with yaml which is saved with the name **py.yaml**.

$ kubectl create –f py.yaml

The above command will create a job. If you want to check the status of a job, use the following command.

$ kubectl describe jobs/py

Labels

Labels are key-value pairs which are attached to pods, replication controller and services. They are used as identifying attributes for objects such as pods and replication controller. They can be added to an object at creation time and can be added or modified at the run time.

Selectors

Labels do not provide uniqueness. In general, we can say many objects can carry the same labels. Labels selector are core grouping primitive in Kubernetes. They are used by the users to select a set of objects.

Kubernetes API currently supports two type of selectors −

* Equality-based selectors
* Set-based selectors

**Replication Controller:**

Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time.

**Replica Set:**

Replica Set ensures how many replica of pod should be running. It can be considered as a replacement of replication controller. The key difference between the replica set and the replication controller is, the replication controller only supports equality-based selector whereas the replica set supports set-based selector.

**Deployments:**

Deployments are upgraded and higher version of replication controller. They manage the deployment of replica sets which is also an upgraded version of the replication controller. They have the capability to update the replica set and are also capable of rolling back to the previous version.

**Volumes:**

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost

**Secrets:**

Secrets can be defined as Kubernetes objects used to store sensitive data such as user name and passwords with encryption.

There are multiple ways of creating secrets in Kubernetes.

* Creating from txt files.
* Creating from yaml file.

**Diff b/w Docker and K8S:**

|  |  |
| --- | --- |
| **Docker** | **Kubernetes** |
| 1. Docker has the ability to reduce the size of development by providing a smaller footprint of the operating system via containers. | * Continues development, integration and deployment * Containerized infrastructure |
| 2.With containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications. | * Application-centric management * Auto-scalable infrastructure |
| 3.You can deploy Docker containers anywhere, on any physical and virtual machines and even on the cloud. | * Environment consistency across development testing and production * Loosely coupled infrastructure, where each component can act as a separate unit |
| 4.Since Docker containers are pretty lightweight, they are very easily scalable. | * Higher density of resource utilization * Predictable infrastructure which is going to be created |