**“Partitioning”** means to divide a single hard drive into many logical drives.

**“file system**” in order to store the data. File system is applied on the partition by formatting it with a particular type of file system.

**Mounting :** Attaching a directory to the file system in order to access the partition and it's file system is known as mounting.

# fdisk -l

# fdisk /dev/sdc

8e for Linux LVM, 82 for Linux Swap and 83 for Linux normal partition

# partprobe /partx -a/kpartx /dev/sdc1 (to update the partitioning information in partition table)

# mkfs.ext2/ext3/ext4/xfs/vfat <device name> ( for example/dev/sdc1)

# mount -a (permanent mount)

# mount (to verify whether it is mounted or not)

# umount -a

# fuser -cu <device name> (to check the users who are accessing that partition)

# lsof <device name> (to check the files which are open in that mount point)

# fuser -ck <opened file name with path> (to kill that opened files)

# df -hT (to see device name, file system type, size, used, available size, use% and mount point)

# du -h <filename or directory name> (to see the size of the file or all the file sizes in that directory)

# ncdu (to list biggest files and directories, we have to install the **ncdu** package before executing this)

# mount -l (to list all the mounted partitions along with their labels)

(i) If the size of the RAM is less than or equal to 2GB, then the size of the swap = 2 X RAM size.

(ii) If the size of the RAM is more than 2GB, then the size of the swap = 2GB + RAM size.

# free -m (to see the present swap size)

# swapon -s (to see the swap usage)

# mkswap <device or partition name> (to format the partition with swap file system

# swapon <device or partition name> (to activate the swap space)

**Attributes of the file system:**

1. Inode number:

Inode numbers are the objects the Linux O/S uses to record the information about the file.

(a) Inode first part contains information about the file, owner, its size and its permissions.

(b) Inode second part contains pointer to data blocks associated with the file content.

(ii) File name

(iii) data block

By running the **# fsck <device or partition name>**command we can check the integrity of the file system.

By using **# gparted**  command we can extend the root partition, otherwise we cannot extend the file systems which is not on LVM

# fdisk -l (to see how many disk are attached to the system)

**Journeling :** It is a dedicated area in the file system where all the changes are tracked when the system crashed.

# tune2fs -l /dev/sdb1 (to check whether the journaling is there or not)

# tune2fs -j /dev/sdb1 (to convert ext2 file system to ext3 file system)

# tune2fs -O dir\_index, has\_journal, unit\_bg /dev/sdb1 (to convert ext2 file system to ext4 file system)

# tune2fs -O extents, dir\_index, unit\_bg /dev/sdb1 (to convert ext3 file system to ext4 file system)

# dump2fs <device or partition name> (to check the metadata of the partition and repair the metadata)

# journalctl(It tracks all the log files between two different timings and by default saved in **/run/log** location)

# journalctl -n 5 (to display last five lines of all the log files)

# journalctl -p err (to display all the error messages)

# journalctl -f (to watch journalctl messages continuously)

# journalctl --since<today> or <yesterday> (to see all the journalctl messages since today or yesterday)

# journalctl --since "date" --until "date" (to see the journal messages between the specified two dates)

# journalctl -pid=1 (to see the pid=1 process name)

# auditctl (to see the audit report)

# sar (sar utility is to watch the system activity report like CPU, memory,...etc.,)

# smartctl -H <hard disk name> (to check the health of the specified hard disk)

# smartctl -i <hard disk name> (to see the information of the specified hard disk)

# smartctl -a <hard disk name> (it gives more information of the specified hard disk

Using this LVM we can extend or reduce the file systems as per requirement without loss of any data

* The size of the logical volume can be increased online, no downtime required.
* Check current size of the logical volume by **# lvdisplay <LV name>**and the size of the file system by **# df -hT** command.
* Increase the size of the logical volume by **# lvextend or # lvresize** commands.
* Then finally update the file system by **# resize2fs or # xfs\_growfs** commands.

Logical volume size cannot be reduced online and it requires downtime

# umount <file system mount point>

# e2fsck <device or partition name>

# lvreduce -L -<size in MB></dev/vgname/lvname>

# resize2fs </dev/vgname/lvname>

# lvdisplay </dev/vgname/lvname> (to check the size of the logical volume>

# mount -a (to mount the file system)

# df -hT (to check the size of the file system)

# lsusb or # fdisk -l (to know the pen drive name)

# mount <pen drive name><mount point>

( to mount the pen drive on the above created mount point)

The CD/DVD ROM device name in Linux is /dev/cdrom

# mkdir /mnt/mycdrom (to create the mount point for CD/DVD)

# mount -t iso9660 /root/rhel6.iso /iso -o ro, loop (to mount the .iso image files

RAID : It provides fault tolerance, load balancing using stripping, mirroring and parity concepts.

(i) RAID - 0 ---- Stripping ---- Minimum 2 disks required

(ii) RAID - 1 ---- Mirroring ---- Minimum 2 disks required

(iii) RAID - (1+0) --- Mirroring + Stripping ---- Minimum 4 disks required

(iv) RAID - (0+1) --- Stripping + Mirroring ---- Minimum 4 disks required

(v) RAID - 5 ---- Stripping with parity ---- Minimum 3 disks required

# mdadm -Cv /dev/md0 -n 2 /dev/sdb /dev/sdc -l 0 (to create the RAID - 0 using disk - 1 and disk - 2)

Link file is a short cut file to the original file

Soft link is nothing but a short cut file. . If original file is deleted, no use of short cut file. ie., we cannot access the original data by selecting the link file.

Hard link in nothing but a backup file. If the original file is deleted, there is no effect on hard link file. ie., we can access the original file data even though the link file is deleted.

find command required the specific location. Without specific location we cannot find the files or directories.

# find / -mmin 20 (to search for files/directories which are modified within 20 minutes, +20 ----> above 20 minutes and -20 -----> below 20 minutes)

# find / -mtime 2 (to search files/directories which are modified within 2 days)

* User names and user id are stored in **/etc/passwd** file.
* User's passwords are stored in  **/etc/shadow** file in an encrypted form.

# date -d "+ 45 days"

# uptime (to see the information from how long the system is running, how many users login and load average)

**grep:** It is used to search a word or sentence in file (ie., inside the file)

# date + %R (to display the time only)

# date + %x (to display the date only)

# whatis <command> (to see the short description of that command)

# whereis<command> (to see the location of that command and location of the document of that command)

# reset (to refresh the terminal)

# stat <file name/directory name> (to see the statistics of the file or directory)

# chmod u+s <file name> (to set the suid on that file)

# chmod g+s <directory name> (to set the sgid on that directory)

Sticky bit: Avoid accidental deletes

# chmod o+t <directory name> (to set the sticky bit permission on that directory)

**How to recover the root password if missed or deleted**

Then press 'b' to boot the system in single user mode.

Then prompt appears and type **# passwd root** command.

**How to restrict the users from login?**

Put the user's hostnames as entries in **/etc/hosts.deny**

**Global profile :**

The location of the global profile is **/etc/bashrc**

**Local profile :**

The location of the profile is **.bash\_profile**

Using Access Control list we assign the permissions to some particular users to access the files and directories. you need to enable ACL while mounting the partition.

# getfacl check ACL permissions

# setfacl assign permissions

**How will you lock a user, if he enters wrong password 3 times?**

# vi /etc/pam.d/system-auth

**auth required pam\_tally.so no\_magic\_root  
account required pam\_tally.so deny=3 no\_magic\_root lock\_time=180**

# faillog -u <user name> (to see the specified users failed login attempts)

By configuring the disk quotas we can restrict the user to use unlimited space on the file system and also to restrict the unlimited files in the file system. We can configure the disk quotas in ways. They are,

(i) user quotas

(ii) group quotas

by **# yum install quota\* -y** command.

# quotaon (to enable the quota)

# quotaoff (to disable the quota)

# edquota (to edit or modify the quota)

# repquota (to display or report the present quota)

# quotacheck (to create a quota database)

\* quotas cab be applied on file systems only.

# chpasswd (to change multiple user's passwords)

A  **Network Protocol** defines rules and conventions for communication between the network devices.

|  |  |
| --- | --- |
| TCP/IP | UDP |
| Transmission Control Protocol | User Datagram Protocol |
| It is connection oriented | It is connection less |
| Reliable | Non-Reliable |
| TCP Acknowledgement will be sent / received | No Acknowledgement |
| Slow communication | Fast communication |
| Protocol No. for TCP is 6 | Protocol No. for UDP is 17 |
| HTTP, FTP, SMTP, ....etc., uses TCP | DNS, DHCP, ....etc., uses UDP |

The IP address sub components are Classes of an IP address, Subnet masks and Gateway.

A subnet mask allows the users to identify which part of an IP address is reserved for the network and which part is available for host use.

A Gateway is the network point that provides entrance into another network.

ethtool <device name> (to check the network cable is connected or not)

|  |
| --- |
| **/etc/sysconfig/network-scripts** is the directory which contains the NIC configuration information. |
| **/etc/sysconfig/network-scripts/ifcfg-<device name>** is the file which contains the NIC configuration details. |
| **/etc/resolve.conf** is the file which contains DNS server IP and domain name location. |
| **/etc/hostname** is the hostname configuration file. |
| **/etc/hosts** is the file which contains the local DNS server IP address. |

Collection of multiple NIC cards and make them as single connection (virtual) NIC card is called bonding. **or Teaming or Bridging**

(a) Mode 0 -----> Round Robbin : It provides load balancing and fault tolerance.

(b) Mode 1 -----> Activebackup

(c) Mode 3 -----> Broadcasting

# vim /etc/sysconfig/network-scripts/ifcfg-bond0

**What is the command to check all the open ports of remote machine?**

# nmap <IP address or hostname of the remote system>  
**What is the command to check all the listening ports and services of your machine?**

**#**netstat -ntulpRunLevels:

0 Power off  
 1 Single user  
 2 Multi user without network  
 3 Multiuser with network  
 4 Development purpose  
 5 GUI  
 6 Restart  
**troubleshoot if the network is not reaching?**

network cable is connected or not by **# ethtool <NIC device name>** command.

if connected then check the IP address is assigned or not by **# ifconfig <NIC device name>** command.

Then check the system uptime by **# uptime** command.

Then check the network services status by **# service network status** and **# serviceNetworkManager status** commands.

Then check the network service at Run Level by **# Chkconfig --list network** command.

Then check whether the source network and destination network are in the same domain or not.

Then finally check the routing table by **# route -n** command.

# netstat -r (to check the default gateway and routing table)

# route (to check the default gateway with routing table)

# ip route (to display the NIC device with default gateway)

# hostname (to see the hostname with fully qualified domain name)

# hostname -i (to see the IP address of the system)

# hostname -d (to check the domain name of the system)

# hostname -s (to check the hostname without domain

# dig or # host or #nslookup (all are used to resolve the name to IP and IP to name)

# nmcli connection show --activate (it shows which connection is active currently)

# nmcli connection up <connection name> (to activate or up the specified connection)

# nmcli connection down <connection name> (to disable or down the specified connection)

# nmcli connection show <connection name> (to see the information about the specified NIC device)

# tracepath (it displays the routing information)

# route -n (to check the gateway)

# service --status-all (to see the list of all the processes which are currently running)

# ls /etc/init.d (is the location of all the services and deamons in RHEL - 6)

# /etc/rc.local (is the last script to be run when the system is booting)

# netstat -r (to see all routing table information)

# netstat -i (to see all the NIC cards information)

# ip link (to check the network connection)

In Linux systems the booting is done in 6 stages.

* BIOS
* MBR
* GRUB
* Kernel
* Init
* Runlevel

# who -r (to see the present run level of the system)

**change the default run level # vim /etc/inittab** command **# init 6**

**# reboot** command will not send the kill signals to the system and it will kill all the running processes and services forcefully and then restart the system.

**# init 6** command will send the kill signals to the system and it will stop all the processes and services one by one and then restart the system.

The drivers is Linux system are known as Modules or Kernel Modules.

# lsmod (to list all the currently loaded modules)

# modprobe -r <module name> (to remove the specified module)

# modprobe <module name> (to install or re-install the module)

# rmmod <module name> (to remove the specified module)

# lsmod |grep -i usb (to see the USB module is loaded or not)

Run level is nothing but to put the system in different levels to perform different maintenance modes.

cron jobs are scheduling jobs automatically at a particular time, day of the week, week of the month and month of the year.

at jobs are executes only once.

* **/etc/crontab** -----> is the file which stores all the scheduled jobs.
* **/etc/cron.deny** -----> is the file used to restrict the users from using cron jobs.
* **/etc/cron.allow -----**> is used to allow only users whose names are mentioned in this file to use cron jobs and this file does not exist by default.
* # crontab -e (to edit the cron job editor to create or remove the cron jobs)
* <minutes><hours><day of the month><month of the year><day of the week><job or script>

# crontab -l -u <user name> (to check the specified user's assigned cron jobs)

# crontab -e (to edit the crontab)

Telnet is a mechanism to connect and to administrate the remote system from local system.

**ways we can connect the remote system?**

(i) telnet (ii) ssh

(iii) rlogin (iv) rcp

(v) ftp (vi) scp

(vii) sftp (viii) tftp

**# ssh-keygen**

Then the public and private keys are generated in **/home/<user name>/.ssh** location. ie., **.ssh** directory in users home directory are **id\_rsa** (private key) and **id\_rsa.pub** (public key).

Then copy the public key **id\_rsa.pub** on the remote system by executing the below command.

**# ssh-copy-id -i <user name>@<IP address of the remote system>**

Go to remote system and check whether the above key is copied or not by **# cat /home/<user name>/.ssh/authorized\_keys** file. And the private key should be in our system.

Whenever we are trying to establish a connection the public key on remote system should be matched with the private key on our system. otherwise there is no connection is established.

(vi) If both public and private keys are matched then connection will be established and first time it will ask the password. Once the connection is established, next time onwards it won't ask any passwords.

**# ssh <user name>@<remote hostname or IP address>** (first time it will asks the password)

(vii) The authentication is done through the public and private keys, so this type of authentication is called keybased authentication.

**prevent the remote login root user**

**# vim /etc/ssh/sshd\_config**

**PermitRootLogin yes/no**

**PermitEmptyPassword no**

**PasswordAuthentication no**

**MaxAuthTries <no.>** (type any numeric value equal to Max. users to allowat a time in place of <no.>

# nmap -p 22 <IP address of the remote host> (to see the ssh is running or not on remote system)

# scp <source file name with full path><IP address of the remote system>:<destination location>

**rsync** is a very good program for backing up or mirroring a directory tree of files from one machine to another machine and for keeping the two machines " **in sync** ".

If **rsync** is combined with ssh, it makes a great utility to sync the data securely otherwise by sniffing any one can see our data ie., no security for our data.

**Platform Backup Tools**

Windows ntbackup

Linux tar, cpio, dd, dump, restore

dump is a command used to take a backup of file systems only.

The options are, -0----->full backup

-(1 - 9) -----> incremental backups

# restore <options><dump backup file> (to restore the backup contents if that data is lost)

There are mainly three types of backups available.

(i) Full backup (Entire file system backup)

(ii) Incremental backup (backup from the last full backup or incremental backup)

(iii) Cumulative or differential backup (backup from last full backup or cumulative backup)

The point - in - time copy of the file system is called the snap shot.

# ps (to see the active process in the system)

# top (It will show a dynamic real-time view of a running system.

# nice (to run a program with modified scheduling priority ie., it runs the process with an adjustable niceness)

# renice (to alter the scheduling priority of one or more running processes)

# ps -aux

(it displays all the terminals processes information including background processes with user names)

# ps -ef (it displays the total processes information with parent process ID (PPID))

# ps -o pid, comm, %mem, %cpu (to display process id, command, %memory and %cpu utilization nothing but filtering the output)

# ps -Ao pid, comm, %mem, %cpu (to display the same information as above but including some more information)

# ps -o pid, comm, %mem, %cpu |sort -k <no.> -r |head -n 10

(to display which process is utilizingmore memory or cpu in reverse order where -k means field, <no.> means field no. and -r reverse order)

# ps -o pid, comm, %mem, %cpu |sort -k 3 -r |head -n 10 (to display the process which occupies more memory and cpu utilization in reverse order)

There are six process states and they are,

(i) Running process (the process which is in running state and is indicated by " **r** " ).

(ii) Sleeping process (the process which is in sleeping state and is indicated by " **s** " )

(iii) Waiting process (the process which is in waiting state and is indicated by " **w "** ).

(iv) Stopping process (the process which is in stopping state and is indicated by " **T "** ).

(v) Orphan process (the process which is running without parent process and is indicated by " **o "** ).

(vi) Zombie process (the process which is running without child process and is indicated by " **Z** " ).

# nice -n <nice value range from -20 to 19><command> (to set a priority to a process before starting it)

**What is the command to see the complete information on virtual memory?**

# vmstat is the command to the complete information on virtual memory like no of processes, memory usage, paging memory, block I/O (input /output), traps, disk and CPU activity.

# vmstat 2 10 (It will give the report for every 2 seconds upto 10 times)

The fields ae, r -----> how many waiting processes

b -----> how many processes are busy

swapd -----> how much virtual memory used

free -----> how much memory is freely available

buffer -----> how much temporary memory using

caching -----> how much caching still using

swapin -----> how much data transferred from RAM to swap

swapout ---> how much data transferred from swap to RAM

bi -----> how much block input

bo -----> how much block output

system in ---> the no. of interrupts

system cs ---> the no. of contexts changed

# vmstat -a (to see the active and inactive processes)

# vmstat -d (to see the statistics of the disk used)

# cat /proc/meminfo (to see the present memory information)

**What is the command to see the I/O statistics?**

# iostat (to see the Input and Output statistics in the Linux system)

\* This command is used to monitoring the system input and output statistics and processes transfer rate.

\* It is also used to monitor how many kilo bytes read per second and how many kilo bytes read and write, shows CPU load average statistics since the last reboot in first line and most current data is shown in the second line.

SAR stands for System Activity Report.The SAR configuration is stored in **/etc/sysconfig/sysstat** file.

# iostat 5 2 (to monitor the input and output statistics for every 5 seconds upto 10 times)

# lscpu (to see the no. of CPUs present in the system)

# lsusb (to see the no. of USB devices present in the system)

# lsblk (to see all the partitions or block devices information)

# dmidecode (to see the complete hardware information of the system)

# dmidecode -t memory (to see the memory information of the system)

# dmidecode -t bios (to see the system's bios information)

# dmidecode -t system (to see the system's information)

# dmidecode -t processor to see the processor's (CPU's) information of the system)

# dmidecode -t 1 (to check the System's Serial No. information)

# dmidecode -t 4 (to see the processor's (CPU's) information)

# dmidecode -t 16 (to check the Max. RAM capacity of the system)

# dmidecode -t 17 (to check how much RAM the system is using)

# pidstat (to monitoring the individual tasks currently being managed by the Linux kernel)

# nfsiostat (to monitor the NFS input and output statistics)

# cifsiostat (to monitor the Samba input and output statistics)

# stat <file name or directory name> (to see the statistics of the file or directory)

**Forward Lookup Zone** is used to resolve**Hostnames** to **IP addresses**.

**Reverse Lookup Zone** is used to resolve **IP addresses** to **Hostnames.**

# service iptables restart (to restart the IP tables in RHEL - 6)

# service iptables save (to save the IP tables in RHEL - 6)

# firewall-cmd --permanent --add-service=dns (to add the DNS service to firewall in RHEL - 7)

# firewall-cmd --complete-reload (to reload the firewall in RHEL - 7)

(xi) Check whether the DNS is resolving or not.

# iptables -A INPUT -i eth0 -p tcp -m tcp --deport 80 -j ACCEPT

# iptables -A OUTPUT -i eth0 -p tcp -m tcp --deport 80 -j ACCEPT

# firewall-cmd --permanent --add-service=http

# firewall-cmd --complete-reload

Take a backup of the database.

**# mysqldump -u root -p <database name>><file name with full path>**

There are four types of firewalls.

(i) **Packet firewalls :**

* It works atPhysical, Data Link and Network Layers.
* It works fast and efficiently.
* It treats each packet in isolation.

(ii) **Statefull firewalls :**

* It identifies a packets connection state.
* It maintains packets history in the state tables.

(iii) **Application layer firewalls :**

* It inspects and filter packets on OSI layer upto Application Layer.
* It identifies if protocols are being misused.

(iv) **Proxies firewalls :**

* It acts as an intermediary.
* It operates at Application Layer.
* It won't allow direct connections.
* # iptables -A INPUT -i eth0 -p tcp --deport 22 -j ACCEPT (to add the rules to the existing iptables to allow ssh)
* where -A ---> Add or append a rule to the INPUT chain for incoming traffic.
* -i eth0 ---> Incoming packets through the interface eth0 will be verified against this added new rule.
* -p tcp -deport 22 ---> protocol is tcp and the destination port is 22.
* -j ACCEPT ---> Accept the packet.
* # iptables -A INPUT -p tcp -m state --state NEW -m tcp --deport 80 -j ACCEPT
* (to allow http traffic)
* # iptables -A IN PUT -p udp -m state --state NEW -m udp --deport 161 -j ACCEPT
* (to allow SNMP traffic through port number 161)
* # iptables -P INPUT DROP (to block the input traffic)
* # iptables -P FORWARD DROP (to block the forward traffic)
* # iptables -p OUTPUT DROP (to block the output traffic)
* (where P is capital letter)
* # iptables -A INPUT -s 9.9.9.9 -j DROP (to block the 9.9.9.9 input traffic)
* # iptables -L (to see the list of the IP tables)
* # iptables -F (to flush the iptable rules nothing but deleting all the rules)

**What is server hardening?**

(i) To checking our system is reaching to standards required by the organization.

(ii) That is minimum password length, minimum size of root partition.

1. Minimum free space and password expiry and all other security standards.

Decommission means the process of removing the old system from the production environment and Recommission means the process of putting the new system into the production environment.

(i) **grep** means Globally search for Regular Expression.

(ii) Using grep we can filter the results to get a particular information.

Pipes are nothing but adding two commands and make as one command.

Filters are nothing but filtering the results what we required.

**A record :**

* It is the Address records also known as host records
* Points to the IP address reflecting the domain
* Used for forward lookup of any domain name

**CNAME Record :**

* It is short abbreviation for Canonical Name
* Provides an alias name for same hostname
* Helps create subdomains

Ansible – 2.7 2.5

Ansible’s “raw” module (for executing commands in a quick and dirty way) and the script module don’t even need Python installed.

$ ansible myhost --sudo -m raw -a "yum install -y python2"

$ ansible all -m ping --ask-pass

**--ask-vault-pass**

ansible-galaxy **[**delete|import|info|init|install|list|login|remove|search|setup**]** **[**--help**]** **[**options**]**

Ansible Installations

---------------------------

<https://www.softwaretestinghelp.com/ansible-tutorial-1/>

inventory 🡪 /etc/ansible/hosts

conf 🡪/etc/ansible/

**$ ansible webservers –m ping**

**$ ansible webservers --list-hosts**

#1) Setup Module

To get information about the network or hardware or OS version or memory related information

**$ ansible webservers –m setup**

#### #2) Command Module

The command module simply executes a specific command on the target machine and gives the output.

**$ ansible webservers –m command - an ‘uptime’**

**$ ansible webservers –m command –a ‘hostname’**

#### #3) Shell Module

To execute any command in the shell of your choice you can use the Shell module.

The shell module commands are run in /bin/sh shell

**$ ansible webservers -m shell -a 'ls -l > temp.txt'**

#### #4) User Module

Using this module one can create or delete users.

**$ ansible webservers -m user -a 'name=user1 password=user1' --become**

**$ ansible webservers -m user -a 'name=user1 state=absent' --become**

* **become** – Privilege to the superuser to run the command

#### #5) File Module

* This module is used to create files, directories, set, or change file permissions and ownership etc

**$ ansible webservers -m file -a 'dest=/home/ansible/niranjan.txt state=touch mode=600 owner=ansible group=ansible'**

Create a directory

To create a directory using the file module, you need to set two parameters.

* Path(alias – name, dest) – This is the absolute path of the directory to be created.
* State – You should enter the value as ‘directory.’ By default, the value is ‘file’.

**$ ansible webservers -m file -a "dest=/home/ansible/vndir state=directory mode=755"**

Delete a file

**$ ansible webservers -m file -a "dest=/home/ansible/niranjan.txt state=absent"**

#6) Copy Module

It is used for copying files to multiple target machines.

**$ ansible webservers -m copy -a "src=sample.txt dest=/home/ansible/sample.txt"**

 Managing Software Packages

If you need to install software packages through ‘yum’ or ‘apt’ you can use the below commands.

**$ ansible webservers –m yum -a “name=git state=present” --become**

Check if the package is installed & update it to the latest version.

**$ ansible webservers -m yum -a “name=git state=latest”**

Install Apache Webserver

**$ ansible webservers -m yum -a "name=httpd state=present" –become**

Check if Maven is installed or not.

**$ ansible webservers -m yum -a "name=maven state=absent" –become**

Managing Services Module

To manage services with ansible, we use a module **‘service’.**

**Starting a service**

**$ ansible webservers -m service -a “name=httpd state=started” --become**

Playbooks define variables, configurations, deployment steps, assign roles, perform multiple tasks. For **E.g.** COPY / DELETE Files and Folders, install packages, start services. So primarily playbooks are defined to orchestrate the steps to multiple machines or servers and get them all to a certain desired state.

**To run any playbook use the following command**

**$ ansible-playbook <playbook.yml>**

**To check the playbook for syntax errors**

**$ ansible-playbook <playbook.yml> --syntax-check**

**To view hosts list**

**$ ansible-playbook <playbook.yml> --list-hosts**

Create multiple directories. To create multiple directories with one single task you can use the loop **with\_items** statement.

---

- hosts: webservers

become: true

tasks:

- name: Create multiple directories

file: path={{item}} state=directory

with\_items:

- '/home/ansible/vn1'

- '/home/ansible/vn2'

- '/home/ansible/vn3'

Using **replace**module we can replace a word with another word. The replace module will need 3 parameters i.e. ‘path’, ‘regexp’ (to find the particular word) and ‘replace’ (providing another word for replacement).

Using the Ansible **archive**module you can compress files or folders to ‘zip’, ‘.gz’, or ‘bz2’ format.

Using the system date and timestamp helps in certain status or logging purposes. The Ansible facts provide access to remote or target servers date and time. So we can use the **debug module**to print the output along with the **var**attribute as shown below.

---

- hosts: webservers

become: true

tasks:

- name: Date and Time Example in Ansible

debug:

var=ansible\_date\_time.time

- hosts: all

tasks:

- name: Ansible timestamp filename example

command: touch niranjan{{ansible\_date\_time.date}}.log

The above playbook will create a dynamic file based on the current date for **E.g**. niranjan2018-07-15.log

We can also capture the output of any task to a register variable.

**Example 13:** Install Apache server. Save the below code and run playbook as shown below.

---

- hosts: webservers

become: true

tasks:

- name: Install Package

yum: name=httpd state=present

- name: Start httpd service

service: name=httpd state=started

The tasks performed are to download the maven file from the URL using the **get\_url** module, extract the file downloaded, move it to a smaller directory, update and run the profile where the maven is added to the path.

---

- hosts: webservers

become: true

tasks:

- name: Download Maven

get\_url: url=http://www-us.apache.org/dist/maven/maven-3/3.5.3/binaries/apache-maven-3.5.3-bin.tar.gz dest=/opt/niranjan/apache-maven-3.5.3-bin.tar.gz

- name: Extract Maven

command: tar xvf /opt/niranjan/apache-maven-3.5.3-bin.tar.gz -C /opt/niranjan

- name: Move to a smaller directory

command: mv /opt/niranjan/apache-maven-3.5.3 /opt/niranjan/maven

- name: Update Profile

copy: content="export M2\_HOME=/opt/niranjan/maven \n" dest=/etc/profile.d/maven.sh

**# lineinfile is used to add additional or append lines to existing files.**

**- lineinfile:**

path: /etc/profile.d/maven.sh

line: 'export PATH=${M2\_HOME}/bin:${PATH}'

- name: Source profile

shell: source /etc/profile.d/maven.sh

You can use **pre\_tasks and post\_tasks**to run certain tasks before or after running the main task

Normally in a playbook, you have so many tasks that are executed. What if you need to execute only a certain task? Tags are the answer to it.

---

- name: Pre , Post tasks and Tags example

hosts: localhost

become: true

tags:

- niranjan

pre\_tasks:

- debug: msg="Started task with tag - niranjan.

tasks:

- name: Going to execute the main task

debug: msg="Currently in the target server"

post\_tasks:

- debug: msg="Completed task with tag - niranjan.

- name: Play without tags

hosts: localhost

become: true

tasks:

- name: Command to list files

shell: ls -lrt > niranjan.txt

Let’s see what happens while running the playbook with the –list-tags option

**$ ansible-playbook preposttagseg.yml --list-tags**

If you need to execute the tasks with the tag niranjan then the command to run would be:

**$ ansible-playbook preposttagseg.yml --tags niranjan**

So we need to restart service only if the changes are done to configuration files. **Handlers** provide that feature.

So the proper flow with handlers would be to have a **notify** option.

---

- hosts: webservers

become: true

tasks:

- name: Install httpd package

yum: name=httpd state=latest

- name: Copy the httpd configuration file

copy: src=/home/ansible/httpd.final dest=/etc/httpd/conf/httpd.conf

- name: Copy index.html file

copy: src=/home/ansible/index.html dest=/var/www/html

notify:

- restart httpd

- name: Start httpd service

service: name=httpd state=started enabled=yes

handlers:

- name: restart httpd

service: name=httpd state=restarted

Ansible Vault

Most of the times when sensitive or confidential data need to be protected in the playbook, then it can be encrypted rather than just keeping it in a text file which is readable by all. Ansible Vault allows you to encrypt the playbook to protect the confidential data.

To create new encrypted files with vault use the **ansible-vault create**command.

**$ ansible-vault create jobagreement.yml**

**$ ansible-vault view jobagreement.yml**

**Changing password of the encrypted files**

Use the command **ansible-vault rekey**to change the password of the file.

**$ ansible-vault rekey jobagreement.yml**

Use the option –ask-vault-pass with the ansible-playbook command.

**$ ansible-playbook users.yml --ask-vault-pass**

Use the command ansible-vault decrypt command.

**$ ansible-vault decrypt jobagreement.yml**

With Ansible roles you can group your variables, tasks, handlers etc., which increase reusability and most certainly reduce syntax errors. It helps to de-clutter the whole code.

In order to create roles, you use the **ansible-galaxy**command which has all the templates to create it.

**So for Example in Continuous Delivery where I am deploying a new build of my J2EE application (WAR file) to tomcat my steps would be as follows:**

* Stop the application
* Uninstall the application
* Deploy the new build of an application
* Start the application

So I would be creating a role with at least 4 tasks and one main file calling it. This way I am making my code more modular and reusable. So let’s call this role as **tomcat** and create it.

**$ cd /etc/ansible/roles**

**$ sudo ansible-galaxy init tomcat --offline**

**The main components we will use in this section include:**

* **tasks/main.yml** – This is the starting point for tasks created for the role. You can use the main.yml file to point to the other task files.
* **vars** – This is to define any variables used.
* **meta** – This is to define information about yourself or the author.

<https://www.softwaretestinghelp.com/ansible-roles-jenkins-integration-ec2-modules/>

**Step 1: Create the main list of all tasks**

Edit the **tasks/main.yml** file and add the below code. As per the example scenario mentioned above, we are defining 4 different tasks. In most cases, the deploy application will also start the application so the last one of the starting application may not be required.

**---**

**# tasks file for tomcat**

**- import\_tasks: stop\_app.yml**

**- import\_tasks: uninstall\_app.yml**

**- import\_tasks: deploy\_app.yml**

**- import\_tasks: start\_app.yml**

**Step 2: Create all the 4 files as per the scenario**

In the below tasks **action: ec2\_facts**is used to get facts from remote EC2 instances and call them in plays/tasks

**tasks/stop\_app.yml file**

---

- name: Gather EC2 instance metadata

action: ec2\_facts

- name: Stop application on {{ansible\_hostname}}

command: wget "http://{{tomcat\_user}}:{{tomcat\_pwd}}@{{ansible\_ec2\_public\_ipv4}}:8080/manager/text/**stop**?path=/HelloWorld-Maven" -O - -q

**tasks/uninstall\_app.yml**

---

- name: Gather EC2 instance metadata

action: ec2\_facts

- name: Undeploy application on {{ansible\_hostname}}

command: wget "http://{{tomcat\_user}}:{{tomcat\_pwd}}@{{ansible\_ec2\_public\_ipv4}}:8080/manager/text/undeploy?path=/HelloWorld-Maven" -O - -q

**tasks/deploy\_app.yml**

---

- name: Deploy the new WAR file to target servers

copy: src=/var/lib/jenkins/workspace/Demo-Maven-Project/target/HelloWorld-Maven.war dest=/home/ansible/tomcat/webapps

In the deploy code if JENKINS is used to build the WAR file then you need to provide the source directory of the WAR file in the **copy** module and the destination is the target server webapps location of tomcat.

**task/start\_app.yml**

---

- name: Gather EC2 instance metadata

action: ec2\_facts

- name: Start application on {{ansible\_hostname}}

command: wget "http://{{tomcat\_user}}:{{tomcat\_pwd}}@{{ansible\_ec2\_public\_ipv4}}:8080/manager/text/start?path=/HelloWorld-Maven" -O - -q

**Step 3: Define Variables**

Edit the**vars/main.yml**file and add the code as shown below.

---

# vars file for tomcat

tomcat\_user: tomcat

tomcat\_pwd: tomcat

**Step 4: Define information in the meta folder**

Edit the meta/main.yml file and add your information like author, description, and company.

galaxy\_info:

author: V Niranjan

description: Devops specialist

company: <Company Name>

**Step 5: Create a main site.yml file**

Lastly, create the main site.yml file to call the role created which in turn will help to deploy the application to the servers or a list of hosts as per the inventory file. Create the file as **/etc/ansible/site.yml**

---

- hosts: webservers

become: true

roles:

- apache

**Step 6: Run playbook file site.yml**

$ ansible-playbook site.yml

### Provision an AWS EC2 instance using Ansible

do not forget to install **boto**which is a pre-requisite and also ensure to export the user “AWS\_ACCESS\_KEY\_ID” and “AWS\_SECRET\_ACCESS\_KEY”.

**The code below will show you how to create an EC2 instance along with creating a security group and key pair.**

* Create a security group
* Create key pair and the PEM file
* Create EC2 instance
* Save the EC2 instance IP address to the ansible inventory file

---

- hosts: localhost

become: true

gather\_facts: False

vars:

region: ap-south-1

instance\_type: t2.micro

ami: ami-5b673c34 # RedHat Linux 7.5

hosts\_file: /etc/ansible/hosts

tasks:

- name: Create security group

ec2\_group:

aws\_access\_key: <access key>

aws\_secret\_key: <Secret key>

name: "vniranjan"

description: "V Niranjan Security Group"

region: "{{ region }}"

rules:

- proto: tcp

from\_port: 22

to\_port: 22

cidr\_ip: 0.0.0.0/0

- name: Create an EC2 key

ec2\_key:

aws\_access\_key: <access key>

aws\_secret\_key: <Secret key>

name: "vniranjan"

region: "{{ region }}"

register: ec2\_key

- name: Save private key (PEM file)

copy: content="{{ec2\_key.key.private\_key}}" dest=/home/ansible/vniranjan.pem mode=0600

when: ec2\_key.changed

- name: Create an ec2 instance

ec2:

aws\_access\_key: <access key>

aws\_secret\_key: <secret key>

key\_name: vniranjan

group: vniranjan # security group name

instance\_type: "{{ instance\_type}}"

image: "{{ ami }}"

wait: true

region: "{{ region }}"

count: 1 # default

count\_tag:

Name: Demo

instance\_tags:

Name: Demo

register: ec2

- name: Save IP to inventory file

lineinfile:

dest: "{{hosts\_file}}"

insertafter: '\[webservers\]'

line: "{{item.private\_ip}}"

with\_items: "{{ec2.instances}}"

**/etc/ansible/hosts inventory file updated with private IP**

**Login to the instance**

ssh -i “vniranjan.pem” [ec2-user@ec2-13-126-30-207.ap-south-1.compute.amazonaws.com](mailto:ec2-user@ec2-13-126-30-207.ap-south-1.compute.amazonaws.com)

Terminating EC2 Instances

In this section let’s know more about terminating EC2 instances.

**In the following screen you can see that there are 2 instances running and the steps for terminating would be in the following order:**

* Terminate the EC2 instances using Instance id’s
* Remove security group
* Remove key pai

---

- hosts: localhost

gather\_facts: false

connection: local

vars:

instance\_ids:

- 'i-05945003313d20603' # Replace these with your EC2 instance id’s

- 'i-0ce5ce5820bddf610'

region: ap-south-1

keypair\_name: vniranjan

securitygroup\_name: vniranjan

tasks:

- name: Terminate EC2 instance

ec2:

aws\_access\_key: <access key>

aws\_secret\_key: <Secret key>

instance\_ids: '{{instance\_ids}}'

region: '{{region}}'

state: absent

wait: true

- name: Remove EC2 Key

ec2\_key:

aws\_access\_key: <access key>

aws\_secret\_key: <Secret key>

name: '{{keypair\_name}}'

state: absent

region: '{{region}}'

- name: Remove Security Group

ec2\_group:

aws\_access\_key: <access key>

aws\_secret\_key: <Secret key>

name: '{{securitygroup\_name}}'

state: absent

region: '{{region}}'

### 2. Manage AWS S3 Objects with Ansible

Ansible S3 module can be used to get or put a file to or from an S3 bucket. To use this module we will need to install and configure **boto** module of python which acts as an API(Application program interface) to access AWS. This has to be installed on the **Ansible control machine.**

**Example 1: Create an empty bucket with a folder**

---

- hosts: localhost

become: true

tasks:

- name: Create an S3 bucket

s3: aws\_access\_key=<Access Key> aws\_secret\_key=<Secret Key> bucket=ansiblevnbucket object

**Get (download) the file from S3 bucket**

object=/niru.txt dest=/home/ansible/niranjan.txt **mode=get**

**Delete an object or file from S3 bucket**

---

object=/development/niranjan.txt **mode=delobj**

### 3. Jenkins Integration with Ansible

In this section, we will see how Jenkins can be integrated with Ansible. The WAR file built using the build process will be used to deploy to Tomcat on the target machine using Ansible. We will be calling the Ansible role created in the previous section in Jenkins using the Ansible plugin.

So once the build is done the deployment of WAR file will be automatically triggered using Ansible.

**Step 1: Create a Jenkins job and configure the SCM repo using the code in GitHub**

**Step 2: Configure the build**

**Build : pom.xml**

**Gloals : clean install**

**Step 3: Create roles directory within the Jenkins workspace**

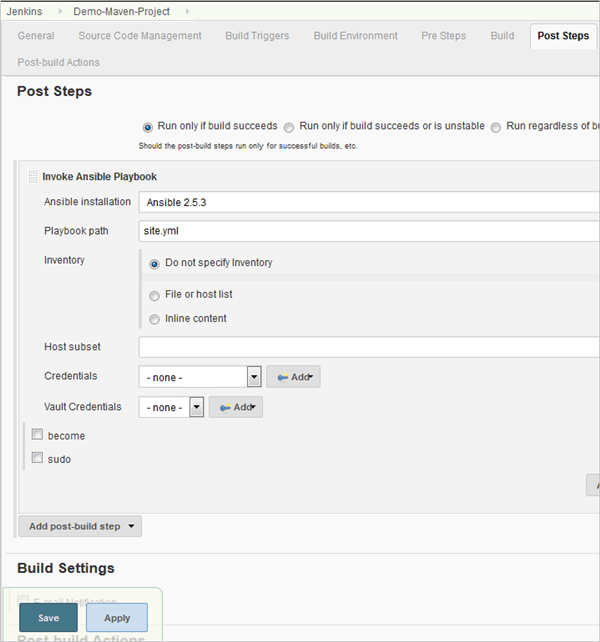
**Step 4: Create the tomcat role in the Jenkins workspace location using the command shown below**

**$ sudo ansible-galaxy init tomcat –offline**

Follow the procedure as in the previous section to create all the files for **tasks, vars, meta and the main site.yml.**

The main **site.yml** file is created in **/var/lib/Jenkins/workspace/<Jenkins-Job-Name>**directory.

**Step 5: Configure the Jenkins post-build step to invoke the Ansible playbook and call the site.yml file. Save the job.**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2018/07/8.Configure-the-jenkins-post-build-step.png)

**Step 6: Trigger the build job and launch the Tomcat URL to verify if the application is deployed correctly.**

Jinga Templates:

 Ansible Templates can help us make our configuration files dynamic and re-usable.

The gist of template files is to take what would normally be a static / concrete file, then replace all the hardcoded bits that make it difficult to share, and then pass in the required values as variables to enable us to declare the file once, but re-use it many times.

Ansible Templates use the excellent [Jinja templating library](http://jinja.pocoo.org/). As such, all our Ansible Templates must have the .j2 file extension.

At its most basic, our Jinja template files allow us to turn something like this:

# templates/our-template-name.j2

server {

root /var/www/{{ domain }}/web;

}

**Why Ansible?**

**============**

It’s agentless.  Unlike Puppet, Chef, Salt, etc.. Ansible operates only over SSH (or optionally ZeroMQ), so there’s none of that crap PKI that you have to deal with using Puppe

**Playbook Mode**

 - This executes a series of commands in order, according to a playbook.

**Non-playbook mode**

 - This executes an ansible module command on a target host.

**https://linuxhint.com/ansible-tutorial-beginners/**

**Dry Run**

Ansible supports running a playbook in dry run mode (also called Check Mode), in this mode, Ansible will **not** make any changes to your host, but simply report what changes would have been made if the playbook was run without this flag.

# ansible-playbook --check playbook.yml

In Ansible, even though include: is technically a language construct, it behaves exactly like a module would. You include it as a task with some key-value pair parameters.

**Docker –v18.09**

## List Docker CLI commands

docker

docker container --help

## Display Docker version and info

docker --version

docker version

docker info

## Execute Docker image

docker run hello-world

## List Docker images

docker image ls

## List Docker containers (running, all, all in quiet mode)

docker container ls

docker container ls --all

docker container ls -aq

ls --all

It’s time to begin building an app the Docker way.

* Stack
* Services
* **Container** (you are here)

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

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

Now run the build command. This creates a Docker image, which we’re going to tag using -t so it has a friendly name.

docker build -t friendlyhello .

**Proxy server settings**

Proxy servers can block connections to your web app once it’s up and running.

# Set proxy server, replace host:port with values for your servers

ENV http\_proxy host:port

ENV https\_proxy host:port

**DNS settings**

DNS misconfigurations can generate problems with pip. You need to set your own DNS server address to make pip work properly.

{

"dns": ["your\_dns\_address", "8.8.8.8"]

}

Before proceeding, save daemon.json and restart the docker service.

sudo service docker restart

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

docker run -p 4000:80 friendlyhello

docker run –p publish  value

docker container stop <Container NAME or ID> to stop the container.

A registry is a collection of repositories, and a repository is a collection of images

Log in to the Docker public registry on your local machine.

$ docker login

docker tag image username/repository:tag

docker tag friendlyhello gordon/get-started:part2

Upload your tagged image to the repository:

docker push username/repository:tag

docker run -d -p 4000:80 friendlyhello # Same thing, but in detached mode

docker container ls -a # List all containers, even those not running

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry

we scale our application and enable load-balancing. To do this, we must go one level up in the hierarchy of a distributed application: the **service**.

In a distributed application, different pieces of the app are called “services.” Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.  to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file.

A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "4000:80"

networks:

- webnet

networks:

webnet:

Run your new load-balanced app

Before we can use the docker stack deploy command we first run:

docker swarm init

You need to give your app a name. Here, it is set to getstartedlab:

docker stack deploy -c docker-compose.yml getstartedlab

docker service ls Get the service ID for the one service in our application:

A single container running in a service is called a **task**.  Tasks are given unique IDs that numerically increment, up to the number of replicas you defined in docker-compose.yml.

List the tasks for your service:

docker service ps getstartedlab\_web

* Take the app down with docker stack rm:
* docker stack rm getstartedlab
* Take down the swarm.

docker swarm leave --force

docker stack ls # List stacks or apps

docker stack deploy -c <composefile> <appname> # Run the specified Compose file

docker service ls # List running services associated with an app

docker service ps <service> # List tasks associated with an app

docker inspect <task or container> # Inspect task or container

docker container ls -q # List container IDs

docker stack rm <appname> # Tear down an application

docker swarm leave --force # Take down a single node swarm from the manage

 you deploy this application onto a cluster, running it on multiple machines. Multi-container, multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a **swarm**.

A swarm is a group of machines that are running Docker and joined into a cluster.

The basic concept is simple enough: run docker swarm init to enable swarm mode and make your current machine a swarm manager, then run docker swarm join on other machines to have them join the swarm as workers.

create a couple of VMs using docker-machine, using the VirtualBox driver:

docker-machine create --driver virtualbox myvm1

docker-machine create --driver virtualbox myvm2

Use this command to list the machines and get their IP addresses.

docker-machine ls

You can send commands to your VMs using docker-machine ssh. Instruct myvm1 to become a swarm manager with docker swarm init and look for output like this:

$ docker-machine ssh myvm1 "docker swarm init --advertise-addr <myvm1 ip>"

Run docker-machine env myvm1 to get the command to configure your shell to talk to myvm1.

Just like before, run the following command to deploy the app on myvm1.

docker stack deploy -c docker-compose.yml getstartedlab

A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together.

* **Docker Engine** − It is used for building Docker images and creating Docker containers.
* **Docker Hub** − This is the registry which is used to host various Docker images.
* **Docker Compose** − This is used to define applications using multiple Docker containers.

**update** − The **update** option is used ensure that all packages are updated on the Linux system.

Docker run –it Ubuntu bash

An image is a combination of a file system and parameters

docker images -q

This command is used to return only the Image ID’s of the images.

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

Syntax

docker inspect Repository

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

Syntax

docker history ImageID

docker top

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

Syntax

docker top ContainerID

docker stats(cpu,memory)

This command is used to provide the statistics of a running container.

Syntax

docker stats ContainerID

docker attach

This command is used to attach to a running container.

Syntax

docker attach ContainerID

docker pause

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

Syntax

docker pause ContainerID

docker unpause

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

Syntax

docker unpause ContainerID

docker kill

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

Syntax

docker kill ContainerID

service docker stop

This command is used to stop the Docker **daemon** process.

Syntax

service docker sto

service docker start

This command is used to start the Docker daemon process.

Syntax

service docker start

Now there is an easier way to attach to containers and exit them cleanly without the need of destroying them. One way of achieving this is by using the **nsenter** command.

Before we run the **nsenter** command, you need to first install the **nsenter**image. It can be done by using the following command −

docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter

nsenter

This method allows one to attach to a container without exiting the container.

Syntax

nsenter –m –u –n –p –i –t containerID command

Options

* **-u** is used to mention the **Uts namespace**
* **-m** is used to mention the **mount namespace**
* **-n** is used to mention the **network namespace**
* **-p** is used to mention the **process namespace**
* **-i** s to make the container run in interactive mode.
* **-t** is used to connect the I/O streams of the container to the host OS.
* **containerID** − This is the ID of the container.
* **Command** − This is the command to run within the container.

sudo nsenter –m –u –n –p –i –t 2978 /bin/bash

The **RUN** command is used to run instructions against the image.

Use the Docker **run** command to download the private registry. This can be done using the following command.

sudo docker run –d –p 5000:5000 –-name registry registry:2

CMD [“apache2ctl”, “-D”, “FOREGROUND”]

the CMD command is used to run apache2 in the background.

CMD Instruction

This command is used to execute a command at runtime when the container is executed.

Syntax

CMD command param1

CMD [“echo” , “hello world”]

ENTRYPOINT

This command can also be used to execute commands at runtime for the container. But we can be more flexible with the ENTRYPOINT command.

Syntax

ENTRYPOINT command param1

ENTRYPOINT [“echo”]

Docker run image helloworld

ENV

This command is used to set environment variables in the container.

Syntax

ENV key value

WORKDIR

This command is used to set the working directory of the container.

Syntax

WORKDIR dirname

Container Linking allows multiple containers to link with each other.

 launch the destination container, but this time, we will link it with our source container.

Changing the Storage Driver for a Container

If you wanted to change to the storage driver used for a container, you can do so when launching the container. This can be done by using the **–volume-driver** parameter when using the **docker run** command. An example is given below −

sudo docker run –d –volume-driver=flocker

–v /home/demo:/var/jenkins\_home –p 8080:8080 –p 50000:50000 jenkins

The **–volume-driver** option is used to specify another storage driver for the container.

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

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

This command can be used to list all the networks associated with Docker on the host.

Syntax

docker network ls

sudo docker network inspect bridge

Creating Your Own New Network

One can create a network in Docker before launching containers. This can be done with the following command −

Syntax

docker network create –-driver drivername name

## Daemon Logging

At the daemon logging level, there are four levels of logging available −

* **Debug** − It details all the possible information handled by the daemon process.
* **Info** − It details all the errors + Information handled by the daemon process.
* **Errors** − It details all the errors handled by the daemon process.
* **Fatal** − It only details all the fatal errors handled by the daemon process.

sudo dockerd –l debug &

* **dockerd** is the executable for the **docker daemon process**.
* The **–l** option is used to specify the logging level. In our case, we are putting this as debug
* **&** is used to come back to the command prompt after the logging has been enabled.

Docker logs containerID

**Docker Compose** is used to run multiple containers as a single service.

sudo ./docker-compose up

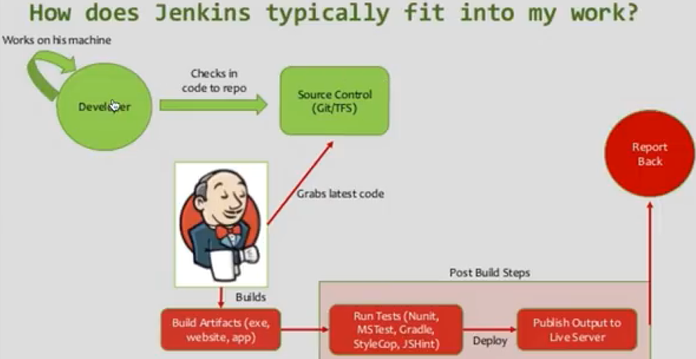
Versions:  
------------

Jenkins: 2.150.1

Java : 1.8

Tomcat: 8.0

Git : 2.14



Manage Jenkins

=============

1. Configure System 🡪 Configure global settings and paths
2. Home Directory
3. Maven Project Configuration
4. # Of executors –default(2)

This refers to the total number of concurrent job executions that can take place on the Jenkins machine.

Your maximum number of concurrent jobs will be (executors on master) + (number of remote slaves) \* (executors per slave).

1. Labels
2. Quiet Period
3. Environment Var’s
4. Tools Locations
5. Time Stamper
6. Jenkins URL
7. Git Hub Servers
8. Docker Label
9. Docker Registry URL
10. Registry credentials
11. Shell
12. Extended Email Notification

|  |
| --- |
|  |

1. Configure Global Security

Enable Security

**Access Control** -🡪Security Realm

Jenkins Own User db

LDAP

Unix user/group db

**Authorization** 🡪

Anyone can do anything

Legacy mode

Loggiedin users can do anything

Matrix based security

Project based matrix authorization strategy

Role based strategy :

**Agents** 🡪

Tcp for jnlp agents

Fixed

Random

Disable

API Token

Sshd server

1. Configure Credentials

Restrictions 🡪 Excludes,Includes

1. Global tool Configuration

Maven

Git

Java

Gradle

Docker

1. Reload Configuration From Disk
2. Manage Plugins

Update

Available

Install

advanced

1. System Information

System properties

Environment vars

Plugins

1. System Log

Log Recorder

1. Load Statistics

#of Online executors

#of busy executors

# of available executors

Queue length

1. Jenkins CLI
2. **java -jar** [**jenkins-cli.jar**](http://3.16.168.90:8080/jnlpJars/jenkins-cli.jar) **-s http://3.16.255.196:8080/ help**
3. Script Console

Groovy script console

1. Manage Nodes

Add,remove,control,monitor various nodes

1. **Manage and Assign Roles**
2. Manage Roles

* Project Roles, Global Roles , Slave Roles

**Roles : Admin,Credentials,Agent,Job,Run,view,SCM,Lockable Resources**

1. Assign Roles

* Global Roles,Item Roles,Node Roles

1. Role Strategy Macros
2. About Jenkins
3. Manage Old Data
4. Manage Users
5. Prepare for shutdown

1.freeStyle Project

-----------------------

1. **General**

Description

Discard old builds

Delivery pipeline configuration

This project is parameterized

Execute concurrent build necessary

1. SCM

None

Git -🡪 Repositories,Branches

mercurial

1. Build Triggers

Triggers build remotely -🡪 Authentication Token

Build after another projects are built 🡪 Projects to watch ,Trigger obnly if build is stable,unstable,fails

Build periodically 🡪 schedule

Github hook trigger for git scm poling

Poll scm 🡪 Schedule

1. Build environment
2. Build

Conditional step

Multiple conditional step

Execute shell command

Windows batch command

6. Post build actions

Archive artifacts

Build other projects

Publish junit test result report

Deploy war to container

Email notification

Delete workspace when build is done

Java -jar Jenkins.war httpPort=9090

Export JENKINS\_HOMe = /users/rk/jenkins

Localhost:8080/Jenkins/restart

Env Variables :

------------------

1.Jenkins\_home

2. user

3.lang

4.Path

5.PWD

6.Logname

7.shell

8.Shlvl

9.WorkSpace directory : $(JENKINS\_HOME)/workspace/($ITEM\_FULLNAME)

10.Build Record Root Directory : ($ITEM\_RootDIR)/builds

11. Project Name

12. BuildNo

13. Build Status

14. Build URL

System Message : <h1>hello system message</h1>

GIT

-----

Git init

Git status

Git add .

Git commit –m “add files”

git remote add origin <https://github.com/kotacharan/jenkins.git>

git push –u origin master

Catlight

---------

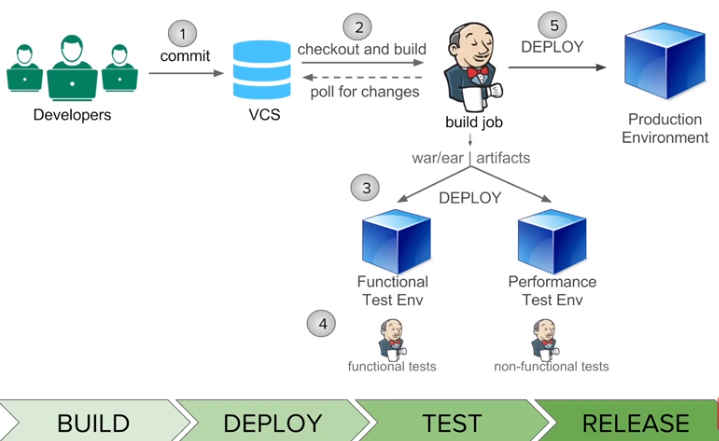
Jenkins build monitor

To monitor the builds

Pipeline:

======

**Build 🡪 Deploy 🡪 Test 🡪 Release**



**1.Deploy to a Container:**

Post Build Action

\*\*/\*.war

**2. Send Email Notification**

Post build Action

Extreme Notification Plugin

**3. Delivery Pipeline view**

**4. Build Pipeline view**

**5. Blue Ocean**

**6. Build Monitor View**

Jenkins Home

------------------

1. Users
2. User Content
3. Updates
4. Secrets
5. Plugins
6. Nodes
7. Logs
8. Jobs

Builds Types

----------------

1. Last failed build
2. Last stable
3. Last successful
4. Last unstable
5. Last un successful

Plugins:

------------

1. Role Based authorization strategy
2. Jenkins Pipeline (or simply "Pipeline") is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins.
3. A continuous delivery pipeline is an automated expression of your process for getting software from version control right through to your users and customers.

Jenkins Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code". The definition of a Jenkins Pipeline is typically written into a text file (called a Jenkinsfile) which in turn is checked into a project’s source control repository.

Java

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker { image 'maven:3.3.3' } }

stages {

stage('build') {

steps {

sh 'mvn --version'

}

}

}

}

[Toggle Scripted Pipeline](https://jenkins.io/doc/pipeline/tour/hello-world/) *(Advanced)*

*Jenkinsfile (Scripted Pipeline)*

/\* Requires the Docker Pipeline plugin \*/

node('docker') {

checkout scm

stage('Build') {

docker.image('maven:3.3.3').inside {

sh 'mvn --version'

}

}

}

Pipelines are made up of multiple steps that allow you to build, test and deploy applications.

On Linux, BSD, and Mac OS (Unix-like) systems, the sh step is used to execute a shell command in a Pipeline

Windows-based systems should use the bat step for executing batch commands.

There are some powerful steps that "wrap" other steps which can easily solve problems like retrying (retry) steps until successful or exiting if a step takes too long (timeout).

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

retry(3) {

sh './flakey-deploy.sh'

}

timeout(time: 3, unit: 'MINUTES') {

sh './health-check.sh'

}

}

}

}

}

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh './flakey-deploy.sh'

}

}

}

}

}

}

When the Pipeline has finished executing, you may need to run clean-up steps or perform some actions based on the outcome of the Pipeline. These actions can be performed in the post section.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Test') {

steps {

sh 'echo "Fail!"; exit 1'

}

}

}

post {

always {

echo 'This will always run'

}

success {

echo 'This will run only if successful'

}

failure {

echo 'This will run only if failed'

}

unstable {

echo 'This will run only if the run was marked as unstable'

}

changed {

echo 'This will run only if the state of the Pipeline has changed'

echo 'For example, if the Pipeline was previously failing but is now successful'

}

}

}

A [workspace](https://jenkins.io/doc/book/glossary/#workspace) is allocated which will contain files checked out from source control as well as any additional working files for the Pipeline.

Environment variables can be set globally, like the example below, or per stage. As you might expect, setting environment variables per stage means they will only apply to the stage in which they’re defined.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

environment {

DISABLE\_AUTH = 'true'

DB\_ENGINE = 'sqlite'

}

stages {

stage('Build') {

steps {

sh 'printenv'

}

}

}

}

Another common use for environment variables is to set or override "dummy" credentials in build or test scripts. Because it’s (obviously) a bad idea to put credentials directly into a Jenkinsfile,

To collect our test results and artifacts, we will use the post section.

post {

always {

junit 'build/reports/\*\*/\*.xml'

}

}

When there are test failures, it is often useful to grab built artifacts from Jenkins for local analysis and investigation. This is made practical by Jenkins’s built-in support for storing "artifacts", files generated during the execution of the Pipeline.

This is easily done with the archiveArtifacts step and a file-globbing expression

post {

always {

archiveArtifacts artifacts: 'build/libs/\*\*/\*.jar', fingerprint: true

junit 'build/reports/\*\*/\*.xml'

}

}

If more than one parameter is specified in the archiveArtifacts step, then each parameter’s name must explicitly be specified in the step code - i.e. artifacts for the artifact’s path and file name and fingerprint to choose this option. If you only need to specify the artifacts' path and file name/s, then you can omit the parameter name artifacts - e.g.  
archiveArtifacts 'build/libs/\*\*/\*.jar'

Since the post section of a Pipeline is guaranteed to run at the end of a Pipeline’s execution, we can add some notification or other steps to perform finalization, notification, or other end-of-Pipeline tasks.

deleteDir() /\* clean up our workspace \*/

Email

post {

failure {

mail to: 'team@example.com',

subject: "Failed Pipeline: **${**currentBuild.fullDisplayName**}**",

body: "Something is wrong with **${**env.BUILD\_URL**}**"

}

}

### Slack

post {

success {

slackSend channel: '#ops-room',

color: 'good',

message: "The pipeline **${**currentBuild.fullDisplayName**}** completed successfully."

}

}

The most basic continuous delivery pipeline will have, at minimum, three stages which should be defined in a Jenkinsfile: Build, Test, and Deploy. For this section we will focus primarily on the Deploy stage, but it should be noted that stable Build and Test stages are an important precursor to any deployment activity.

One common pattern is to extend the number of stages to capture additional deployment environments, like "staging" or "production", as shown in the following snippet.

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

## Asking for human input to proceed to deployment

 to judge if the application is in a good enough state to "promote" to the production environment.

This can be accomplished with the input step.

the "Sanity check" stage actually blocks for input and won’t proceed without a person confirming the progress.

stage('Sanity check') {

steps {

input "Does the staging environment look ok?"

}

}

#### On macOS and Linux

docker run \

--rm \

-u root \

-p 8080:8080 \

-v jenkins-data:/var/jenkins\_home \

-v /var/run/docker.sock:/var/run/docker.sock \

-v "$HOME":/home \

jenkinsci/blueocean

#### On Windows

docker run ^

--rm ^

-u root ^

-p 8080:8080 ^

-v jenkins-data:/var/jenkins\_home ^

-v /var/run/docker.sock:/var/run/docker.sock ^

-v "%HOMEPATH%":/home ^

jenkinsci/blueocean

This means you could access the Jenkins/Blue Ocean container (through a separate terminal/command prompt window) with a docker exec command like:

docker exec -it jenkins-tutorials bash

Run the same docker run …​ command you ran for [macOS, Linux](https://jenkins.io/doc/tutorials/build-a-java-app-with-maven/#on-macos-and-linux) or [Windows](https://jenkins.io/doc/tutorials/build-a-java-app-with-maven/#on-windows) above.  
**Note:** This process also updates the jenkinsci/blueocean Docker image, if an updated one is available.

pipeline {

agent {

docker {

image 'maven:3-alpine'

args '-v /root/.m2:/root/.m2'

}

}

stages {

stage('Build') {

steps {

sh 'mvn -B -DskipTests clean package'

}

}

}

}

As of version 2.5 of the Pipeline plugin, Pipeline supports two discrete syntaxes which are detailed below. For the pros and cons of each

Declarative Pipeline is a relatively recent addition to Jenkins Pipeline [[1](https://jenkins.io/doc/book/pipeline/syntax/#_footnotedef_1)] which presents a more simplified and opinionated syntax on top of the Pipeline sub-systems.

The top-level of the Pipeline must be a block, specifically: pipeline { }

Blocks must only consist of [Sections](https://jenkins.io/doc/book/pipeline/syntax/#declarative-sections), [Directives](https://jenkins.io/doc/book/pipeline/syntax/#declarative-directives), [Steps](https://jenkins.io/doc/book/pipeline/syntax/#declarative-steps), or assignment statements.

Sections in Declarative Pipeline typically contain one or more [Directives](https://jenkins.io/doc/book/pipeline/syntax/#declarative-directives) or [Steps](https://jenkins.io/doc/book/pipeline/syntax/#declarative-steps).

The agent section specifies where the entire Pipeline, or a specific stage

|  |  |
| --- | --- |
| **Required** | Yes |
| **Parameters** | [Described below](https://jenkins.io/doc/book/pipeline/syntax/#agent-parameters) |
| **Allowed** | In the top-level pipeline block and each stage block. |

**any**

Execute the Pipeline, or stage, on any available agent. For example: agent any

**none**

When applied at the top-level of the pipeline block no global agent will be allocated for the entire Pipeline run and each stage section will need to contain its own agent section. For example: agent none

**label**

Execute the Pipeline, or stage, on an agent available in the Jenkins environment with the provided label. For example: agent { label 'my-defined-label' }

**node**

agent { node { label 'labelName' } } behaves the same as agent { label 'labelName' }, but node allows for additional options (such as customWorkspace).

**docker**

Execute the Pipeline, or stage, with the given container which will be dynamically provisioned on a [node](https://jenkins.io/doc/book/glossary/#node) pre-configured to accept Docker-based Pipelines,

 agent { docker 'maven:3-alpine' } or

agent {

docker {

image 'maven:3-alpine'

label 'my-defined-label'

args '-v /tmp:/tmp'

}

 on a node matching the optionally defined label parameter. docker also optionally accepts an argsparameter

docker also optionally accepts a registryUrl and registryCredentialsId parameters which will help to specify the Docker Registry to use and its credentials.

agent {

docker {

image 'myregistry.com/node'

label 'my-defined-label'

registryUrl 'https://myregistry.com/'

registryCredentialsId 'myPredefinedCredentialsInJenkins'

}

}

**dockerfile**

Execute the Pipeline, or stage, with a container built from a Dockerfile contained in the source repository.  In order to use this option, the Jenkinsfile must be loaded from either a Multibranch Pipeline, or a "Pipeline from SCM."

 this is the Dockerfile in the root of the source repository: agent { dockerfile true }

If building a Dockerfile in another directory, use the dir option: agent { dockerfile { dir 'someSubDir' } }

agent {

// Equivalent to "docker build -f Dockerfile.build --build-arg version=1.0.2 ./build/

dockerfile {

filename 'Dockerfile.build'

dir 'build'

label 'my-defined-label'

additionalBuildArgs '--build-arg version=1.0.2'

args '-v /tmp:/tmp'

}

}

**label**

A string. The label on which to run the Pipeline or individual stage.

This option is valid for node, docker and dockerfile, and is required for node.

**customWorkspace**

A string. Run the Pipeline or individual stage this agent is applied to within this custom workspace, rather than the default.

**args**

A string. Runtime arguments to pass to docker run.

[**postcondition**](https://jenkins.io/doc/book/pipeline/syntax/#post-conditions)blocks: always, changed, fixed, regression, aborted, failure, success, unstable, unsuccessful, and cleanup

**fixed**

Only run the steps in post if the current Pipeline’s or stage’s run is successful and the previous run failed or was unstable.

**regression**

Only run the steps in post if the current Pipeline’s or stage’s run’s status is failure, unstable, or aborted and the previous run was successful.

**aborted**

Only run the steps in post if the current Pipeline’s or stage’s run has an "aborted" status, usually due to the Pipeline being manually aborted. This is typically denoted by gray in the web UI.

**cleanup**

Run the steps in this post condition after every other post condition has been evaluated, regardless of the Pipeline or stage’s status.

**unsuccessful**

Only run the steps in post if the current Pipeline’s or stage’s run has not a "success" status. This is typically denoted in the web UI depending on the status previously mentioned

the credentials() method will ensure that the environment variable specified contains the Secret Text contents

stage('Example') {

environment {

AN\_ACCESS\_KEY = credentials('my-prefined-secret-text')

}

steps {

sh 'printenv'

}

}

The environment block has a helper method credentials() defined which can be used to access pre-defined Credentials by their identifier in the Jenkins environment

The options directive allows configuring Pipeline-specific options from within the Pipeline itself.

**buildDiscarder**

Persist artifacts and console output for the specific number of recent Pipeline runs. For example: options { buildDiscarder(logRotator(numToKeepStr: '1')) }

**checkoutToSubdirectory**

Perform the automatic source control checkout in a subdirectory of the workspace. For example: options { checkoutToSubdirectory('foo') }

**disableConcurrentBuilds**

Disallow concurrent executions of the Pipeline. Can be useful for preventing simultaneous accesses to shared resources, etc. For example: options { disableConcurrentBuilds() }

**newContainerPerStage**

Used with docker or dockerfile top-level agent. When specified, each stage will run in a new container instance on the same node, rather than all stages running in the same container instance.

**overrideIndexTriggers**

Allows overriding default treatment of branch indexing triggers. If branch indexing triggers are disabled at the multibranch or organization label, options { overrideIndexTriggers(true) } will enable them for this job only. Otherwise, options { overrideIndexTriggers(false) } will disable branch indexing triggers for this job only.

**preserveStashes**

Preserve stashes from completed builds, for use with stage restarting. For example: options { preserveStashes() } to preserve the stashes from the most recent completed build, or options { preserveStashes(buildCount: 5) } to preserve the stashes from the five most recent completed builds.

**quietPeriod**

Set the quiet period, in seconds, for the Pipeline, overriding the global default. For example: options { quietPeriod(30) }

**retry**

On failure, retry the entire Pipeline the specified number of times. For example: options { retry(3) }

**skipDefaultCheckout**

Skip checking out code from source control by default in the agent directive. For example: options { skipDefaultCheckout() }

**skipStagesAfterUnstable**

Skip stages once the build status has gone to UNSTABLE. For example: options { skipStagesAfterUnstable() }

**timeout**

Set a timeout period for the Pipeline run, after which Jenkins should abort the Pipeline. For example: options { timeout(time: 1, unit: 'HOURS') }

**timestamps**

Prepend all console output generated by the Pipeline run with the time at which the line was emitted. For example: options { timestamps() }

**parallelsAlwaysFailFast**

Set failfast true for all subsequent parallel stages in the pipeline. For example: options { parallelsAlwaysFailFast() }

The parameters directive provides a list of parameters which a user should provide when triggering the Pipeline. The values for these user-specified parameters are made available to Pipeline steps via the params object,

Available Parameters

**string**

A parameter of a string type, for example: parameters { string(name: 'DEPLOY\_ENV', defaultValue: 'staging', description: '') }

**text**

A text parameter, which can contain multiple lines, for example: parameters { text(name: 'DEPLOY\_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '') }

**booleanParam**

A boolean parameter, for example: parameters { booleanParam(name: 'DEBUG\_BUILD', defaultValue: true, description: '') }

**choice**

A choice parameter, for example: parameters { choice(name: 'CHOICES', choices: ['one', 'two', 'three'], description: '') }

**file**

A file parameter, which specifies a file to be submitted by the user when scheduling a build, for example: parameters { file(name: 'FILE', description: 'Some file to upload') }

**password**

A password parameter, for example: parameters { password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password') }

***Jenkinsfile (Declarative Pipeline)***

pipeline {

agent any

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some information about the person')

booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this value')

choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick something')

password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a password')

file(name: "FILE", description: "Choose a file to upload")

}

stages {

stage('Example') {

steps {

echo "Hello **${**params.PERSON**}**"

echo "Biography: **${**params.BIOGRAPHY**}**"

echo "Toggle: **${**params.TOGGLE**}**"

echo "Choice: **${**params.CHOICE**}**"

echo "Password: **${**params.PASSWORD**}**"

}

}

}

}

The triggers directive defines the automated ways in which the Pipeline should be re-triggered.

The triggers currently available are cron, pollSCM and upstream.

**cron**

Accepts a cron-style string to define a regular interval at which the Pipeline should be re-triggered, for example: triggers { cron('H \*/4 \* \* 1-5') }

**pollSCM**

Accepts a cron-style string to define a regular interval at which Jenkins should check for new source changes. If new changes exist, the Pipeline will be re-triggered. For example: triggers { pollSCM('H \*/4 \* \* 1-5') }

**upstream**

Accepts a comma separated string of jobs and a threshold. When any job in the string finishes with the minimum threshold, the Pipeline will be re-triggered. For example: triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }

A section defining tools to auto-install and put on the PATH. This is ignored if agent none is specified.

agent any

tools {

maven 'apache-maven-3.0.1'

}

The input directive on a stage allows you to prompt for input, using the [input step](https://jenkins.io/doc/pipeline/steps/pipeline-input-step/#input-wait-for-interactive-input).

Configuration options

**message**

Required. This will be presented to the user when they go to submit the input.

**id**

An optional identifier for this input. Defaults to the stage name.

**ok**

Optional text for the "ok" button on the input form.

**submitter**

An optional comma-separated list of users or external group names who are allowed to submit this input. Defaults to allowing any user.

**submitterParameter**

An optional name of an environment variable to set with the submitter name, if present.

**parameters**

An optional list of parameters to prompt the submitter to provide. See [parameters](https://jenkins.io/doc/book/pipeline/syntax/#parameters) for more information.

stage('Example') {

input {

message "Should we continue?"

ok "Yes, we should."

submitter "alice,bob"

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

}

The when directive allows the Pipeline to determine whether the stage should be executed depending on the given condition.

If the when directive contains more than one condition, all the child conditions must return true for the stage to execute. This is the same as if the child conditions were nested in an allOf condition (see the [examples](https://jenkins.io/doc/book/pipeline/syntax/#when-example)below). If an anyOf condition is used, note that the condition skips remaining tests as soon as the first "true" condition is found.

More complex conditional structures can be built using the nesting conditions: not, allOf, or anyOf.

Built-in Conditions

**branch**

Execute the stage when the branch being built matches the branch pattern given, for example: when { branch 'master' }. Note that this only works on a multibranch Pipeline.

**buildingTag**

Execute the stage when the build is building a tag. Example: when { buildingTag() }

**changelog**

Execute the stage if the build’s SCM changelog contains a given regular expression pattern, for example: when { changelog '.\*^\\[DEPENDENCY\\] .+$' }

**changeset**

Execute the stage if the build’s SCM changeset contains one or more files matching the given string or glob. Example: when { changeset "\*\*/\*.js" }

By default the path matching will be case insensitive, this can be turned off with the caseSensitive parameter, for example: when { changeset glob: "ReadMe.\*", caseSensitive: true }

**changeRequest**

Executes the stage if the current build is for a "change request" (a.k.a. Pull Request on GitHub and Bitbucket, Merge Request on GitLab or Change in Gerrit etc.). When no parameters are passed the stage runs on every change request, for example: when { changeRequest() }.

By adding a filter attribute with parameter to the change request, the stage can be made to run only on matching change requests. Possible attributes are id, target, branch, fork, url, title, author, authorDisplayName, and authorEmail. Each of these corresponds to a CHANGE\_\* environment variable, for example: when { changeRequest target: 'master' }.

The optional parameter comparator may be added after an attribute to specify how any patterns are evaluated for a match: EQUALS for a simple string comparison (the default), GLOB for an ANT style path glob (same as for example changeset), or REGEXP for regular expression matching. Example: when { changeRequest authorEmail: "[\\w\_-.]+@example.com", comparator: 'REGEXP' }

**environment**

Execute the stage when the specified environment variable is set to the given value, for example: when { environment name: 'DEPLOY\_TO', value: 'production' }

**equals**

Execute the stage when the expected value is equal to the actual value, for example: when { equals expected: 2, actual: currentBuild.number }

**expression**

Execute the stage when the specified Groovy expression evaluates to true, for example: when { expression { return params.DEBUG\_BUILD } } Note that when returning strings from your expressions they must be converted to booleans or return null to evaluate to false. Simply returning "0" or "false" will still evaluate to "true".

**tag**

Execute the stage if the TAG\_NAME variable matches the given pattern. Example: when { tag "release-\*" }. If an empty pattern is provided the stage will execute if the TAG\_NAME variable exists (same as buildingTag()).

The optional parameter comparator may be added after an attribute to specify how any patterns are evaluated for a match: EQUALS for a simple string comparison, GLOB (the default) for an ANT style path glob (same as for example changeset), or REGEXP for regular expression matching. For example: when { tag pattern: "release-\\d+", comparator: "REGEXP"}

**not**

Execute the stage when the nested condition is false. Must contain one condition. For example: when { not { branch 'master' } }

**allOf**

Execute the stage when all of the nested conditions are true. Must contain at least one condition. For example: when { allOf { branch 'master'; environment name: 'DEPLOY\_TO', value: 'production' } }

**anyOf**

Execute the stage when at least one of the nested conditions is true. Must contain at least one condition. For example: when { anyOf { branch 'master'; branch 'staging' } }

**triggeredBy**

Execute the stage when the current build has been triggered by the param given. For example: ` when { triggeredBy 'SCMTrigger' }`

Stages in Declarative Pipeline may declare a list of nested stages to be run within them in sequential order.  Note that a stage must have one and only one of steps, parallel, or stages,

the last for sequential stages. It is not possible to nest a parallel block within a stage directive if that stage directive is nested within a parallel block itself. However, a stage directive within a parallel block can use all other functionality of a stage, including agent, tools, when, etc

Stages in Declarative Pipeline may declare a number of nested stages within a parallel block, which will be executed in parallel

The script step takes a block of [Scripted Pipeline](https://jenkins.io/doc/book/pipeline/syntax/#scripted-pipeline) and executes that in the Declarative Pipeline.

Whenever you create any Pipeline project in Blue Ocean, Jenkins actually creates this as a multibranch Pipeline project behind the scenes.

 "Pipeline stub" in each branch and proceeds to run each Pipeline against its respective branch, whose build results are shown on the **Activity** page of the main Blue Ocean interface.

ITEM

------

1. FreeStyle
2. Maven
3. Pipeline
4. MultiBranch

View

------

1. List
2. My View
3. Delivery Pipeline View
4. Global View
5. Build Pipeline view

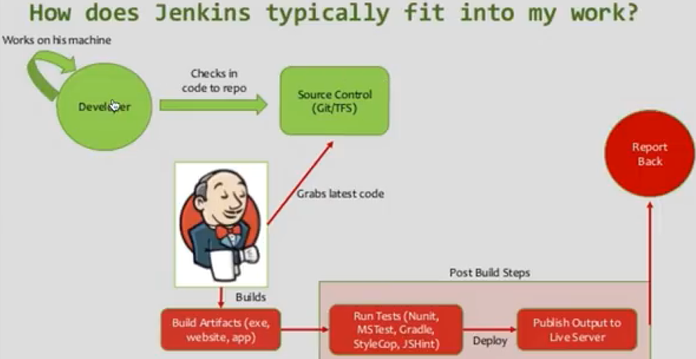
Versions:  
------------

Jenkins: 2.150.1

Java : 1.8

Tomcat: 8.0

Git : 2.14



Manage Jenkins

=============

1. Configure System
2. Configure Global Security
3. Configure Credentials
4. Global tool Configuration
5. Reload Configuration From Disk
6. Manage Plugins
7. System Information
8. System Log
9. Load Statistics
10. Jenkins CLI
11. Script Console
12. Manage Nodes
13. **Manage and Assign Roles**
14. Manage Roles

* Project Roles, Global Roles , Slave Roles

**Roles : Admin,Credentials,Agent,Job,Run,view,SCM,Lockable Resources**

1. Assign Roles

* Global Roles,Item Roles,Node Roles

1. Role Strategy Macros
2. About Jenkins
3. Manage Old Data
4. Manage Users
5. Prepare for shutdown

Java -jar Jenkins.war httpPort=9090

Export JENKINS\_HOMe = /users/rk/jenkins

Localhost:8080/Jenkins/restart

Env Variables :

------------------

1.Jenkins\_home

2. user

3.lang

4.Path

5.PWD

6.Logname

7.shell

8.Shlvl

9.WorkSpace directory : $(JENKINS\_HOME)/workspace/($ITEM\_FULLNAME)

10.Build Record Root Directory : ($ITEM\_RootDIR)/builds

11. Project Name

12. BuildNo

13. Build Status

14. Build URL

System Message : <h1>hello system message</h1>

GIT

-----

Git init

Git status

Git add .

Git commit –m “add files”

git remote add origin <https://github.com/kotacharan/jenkins.git>

git push –u origin master

Catlight

---------

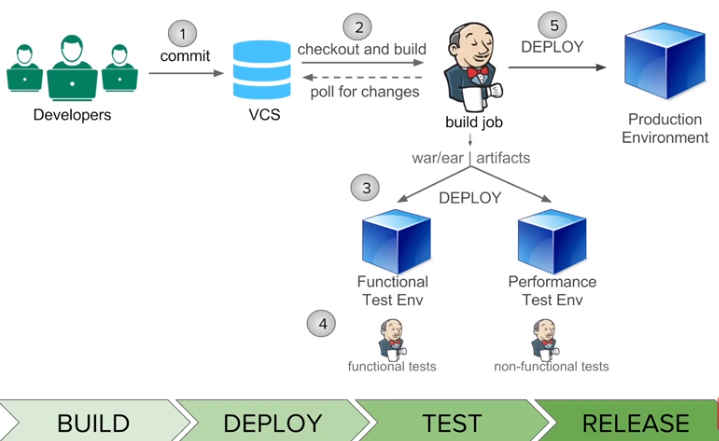
Jenkins build monitor

To monitor the builds

Pipeline:

======

**Build 🡪 Deploy 🡪 Test 🡪 Release**



**1.Deploy to a Container:**

Post Build Action

\*\*/\*.war

**2. Send Email Notification**

Post build Action

Extreme Notification Plugin

**3. Delivery Pipeline view**

**4. Build Pipeline view**

**5. Blue Ocean**

**6. Build Monitor View**

Jenkins Home

------------------

1. Users
2. User Content
3. Updates
4. Secrets
5. Plugins
6. Nodes
7. Logs
8. Jobs

Builds Types

----------------

1. Last failed build
2. Last stable
3. Last successful
4. Last unstable
5. Last un successful

Plugins:

------------

1. Role Based authorization strategy
2. Jenkins Pipeline (or simply "Pipeline") is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins.
3. A continuous delivery pipeline is an automated expression of your process for getting software from version control right through to your users and customers.

Jenkins Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code". The definition of a Jenkins Pipeline is typically written into a text file (called a Jenkinsfile) which in turn is checked into a project’s source control repository.

Java

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker { image 'maven:3.3.3' } }

stages {

stage('build') {

steps {

sh 'mvn --version'

}

}

}

}

[Toggle Scripted Pipeline](https://jenkins.io/doc/pipeline/tour/hello-world/) *(Advanced)*

*Jenkinsfile (Scripted Pipeline)*

/\* Requires the Docker Pipeline plugin \*/

node('docker') {

checkout scm

stage('Build') {

docker.image('maven:3.3.3').inside {

sh 'mvn --version'

}

}

}

Pipelines are made up of multiple steps that allow you to build, test and deploy applications.

On Linux, BSD, and Mac OS (Unix-like) systems, the sh step is used to execute a shell command in a Pipeline

Windows-based systems should use the bat step for executing batch commands.

There are some powerful steps that "wrap" other steps which can easily solve problems like retrying (retry) steps until successful or exiting if a step takes too long (timeout).

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

retry(3) {

sh './flakey-deploy.sh'

}

timeout(time: 3, unit: 'MINUTES') {

sh './health-check.sh'

}

}

}

}

}

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh './flakey-deploy.sh'

}

}

}

}

}

}

When the Pipeline has finished executing, you may need to run clean-up steps or perform some actions based on the outcome of the Pipeline. These actions can be performed in the post section.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Test') {

steps {

sh 'echo "Fail!"; exit 1'

}

}

}

post {

always {

echo 'This will always run'

}

success {

echo 'This will run only if successful'

}

failure {

echo 'This will run only if failed'

}

unstable {

echo 'This will run only if the run was marked as unstable'

}

changed {

echo 'This will run only if the state of the Pipeline has changed'

echo 'For example, if the Pipeline was previously failing but is now successful'

}

}

}

A [workspace](https://jenkins.io/doc/book/glossary/#workspace) is allocated which will contain files checked out from source control as well as any additional working files for the Pipeline.

Environment variables can be set globally, like the example below, or per stage. As you might expect, setting environment variables per stage means they will only apply to the stage in which they’re defined.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

environment {

DISABLE\_AUTH = 'true'

DB\_ENGINE = 'sqlite'

}

stages {

stage('Build') {

steps {

sh 'printenv'

}

}

}

}

Another common use for environment variables is to set or override "dummy" credentials in build or test scripts. Because it’s (obviously) a bad idea to put credentials directly into a Jenkinsfile,

To collect our test results and artifacts, we will use the post section.

post {

always {

junit 'build/reports/\*\*/\*.xml'

}

}

When there are test failures, it is often useful to grab built artifacts from Jenkins for local analysis and investigation. This is made practical by Jenkins’s built-in support for storing "artifacts", files generated during the execution of the Pipeline.

This is easily done with the archiveArtifacts step and a file-globbing expression

post {

always {

archiveArtifacts artifacts: 'build/libs/\*\*/\*.jar', fingerprint: true

junit 'build/reports/\*\*/\*.xml'

}

}

If more than one parameter is specified in the archiveArtifacts step, then each parameter’s name must explicitly be specified in the step code - i.e. artifacts for the artifact’s path and file name and fingerprint to choose this option. If you only need to specify the artifacts' path and file name/s, then you can omit the parameter name artifacts - e.g.  
archiveArtifacts 'build/libs/\*\*/\*.jar'

Since the post section of a Pipeline is guaranteed to run at the end of a Pipeline’s execution, we can add some notification or other steps to perform finalization, notification, or other end-of-Pipeline tasks.

deleteDir() /\* clean up our workspace \*/

Email

post {

failure {

mail to: 'team@example.com',

subject: "Failed Pipeline: **${**currentBuild.fullDisplayName**}**",

body: "Something is wrong with **${**env.BUILD\_URL**}**"

}

}

### Slack

post {

success {

slackSend channel: '#ops-room',

color: 'good',

message: "The pipeline **${**currentBuild.fullDisplayName**}** completed successfully."

}

}

The most basic continuous delivery pipeline will have, at minimum, three stages which should be defined in a Jenkinsfile: Build, Test, and Deploy. For this section we will focus primarily on the Deploy stage, but it should be noted that stable Build and Test stages are an important precursor to any deployment activity.

One common pattern is to extend the number of stages to capture additional deployment environments, like "staging" or "production", as shown in the following snippet.

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

## Asking for human input to proceed to deployment

 to judge if the application is in a good enough state to "promote" to the production environment.

This can be accomplished with the input step.

the "Sanity check" stage actually blocks for input and won’t proceed without a person confirming the progress.

stage('Sanity check') {

steps {

input "Does the staging environment look ok?"

}

}

CI -- > Integration, building and testing code with in the development environment

CI pipeline should run on every code push/commit

CD 🡪 Taking CI builds and run through the deployment procedures on prod or non-prod env’s

Pipeline steps

------------------

1. **Push the code to source code repo**

--- > Developers are committing their code to the mainline/feature branches. Then feature braches need to merged to the mainline

--- > CI tool monitors the code repo and when a change is detected runs ci pipeline steps

---- > CI pipeline runs different steps of tasks with the goal to verify that the code works as expected. If the end results is partial trust that required additional manual testing

**Promotion** -- > if the whole pipeline is run without any failure, the commit is promoted to a release. Candidate that requires manual verifications

**Failure Notification 🡪** If any of the steps failed the process is aborted and a failure notification is sent to the developers that made the commit and all other interested parties

1. **Static analysis**

--- > Highlighting possible coding errors to make sure that agreed formatting is followed

Check style and Find bugs for Java and PMD for a variety of languages

---- > It is often first step in the pipeline, that is execution tends to be very fast and most cases faster than any other step we have in the pipeline

--- > CI tool is in-charge detecting the change in the code repository, checking out the code and running the rest of the steps

--- > static analysis is performed without executing the code

--- > fail note

1. Pre deployment testing

* > Pre deployment tests are mandatory.

--- > unit tests are always fall in to this category and few others might be run as well.

For Eg,you can execute the Functional Testing without deploying the code run them now

--- > Tests that do not require code to be deployed are run in this step

1. Packaging and deployment to the test environment

--- > Once we did all types of verifications that could be done without actually deploying the application, it’s time to package it.

--- > In java we would create jar/war/ear

----- > tar/zip compressions of a file easier transfers to servers

--- > once deployment package is created, proceed to deploy test env

----- > compile the code and create Docker container with all the dependencies (libraries, runtime, application server and so on)

---- > In case of any failure of any of the steps in the deployment pipeline,failure notification is sent and the process is aborted

---- > The package or the container is deployed to one or more test env’s

1. **Post deployment testing**

Once deployed to the test env, we can execute the rest of the tests,those that could not be run without deploying the application

Include functional, integration and performance testing

Once post deploy tests are finished, ci pipeline is completed.

Packages/artifacts we generate during packaging and deployment to test environment are waiting for further usually manual and verifications

**Continuous Delivery**

---------------------------

Cd same as ci,ci assumes there are manual validations to be performed after successful implementation of cd pipeline results in packages/artifacts being ready to be deployed to production

**Continues Deployment**

------------------------------

Fully automated process. Ends to the application to be deployed to production

Entire cd pipeline, deployment to production, post deployment testing

Maven

---------

POM : XML representation of project resources like source code, test code,dependenices(exetranal jar’s)

Details description of project, Including information about versioning, configuration management, dependencies, application and testing resources, team members and structure.

POM file should be located in ROOt directory of project belongs to.

Project : group Id,Artifact Id,Version

Elements

----------

1.Dependencies

2. Repo’s

3. Plugin’s repo’s

4. build

5. report

Mvn clean package

1. Clean the target dir and package the project as jar
2. Jar available as artifact\_id\trget folder
3. Test reports as target\surefire reportd
4. Maven compiles source code and test the source code
5. Run the test cases
6. Created the package

Core concepts:

----------------

Build life cycle(default,clean,site)mvn clean package site

-🡪 build phases

--🡪goals

Generate-sources,Clean,compile,test compile,test,package,integration test,install,deploy

Install 🡪 copy artifact to the local repo

Deploy 🡪 copied to remote repo

Mvn clean install/deploy

Dependencies(external jars files)

build plugins(extra goals to achieve in build phase,add a plugin to the POM file)

common plugins: clean,compiler,surefire,jar,war,Javadoc,

build profiles(build project for diff env’s)

Settings.xml

--------------

Configure settings for maven across all maven pom files.configure:

1. Location of local repo
2. Active build Profile

Maven Installation dir : $M2\_Home/conf/settings.xml

User’s Home dir : ${User.Home}/.me/settings.xml(It over rides the other file)

Build Profiles

----------------

Set of configuration values used to set or override default values of maven build

Customized build for diff env’s

EX : path of DB Server for dev,test,prod

Specified in POM.xml file using activeProfiles/profiles elements

Types

---------

1. Per Project -- > Defined in project pom.xml
2. Per user 🡪 defined maven settings.xml (.m2/)
3. Global 🡪 Defined maven global settings.xml file(/conf/settings.xml)

Profile Activation

----------------------

1. Explicitly using command console input

Create project

* Environment specific files
* Src/main/resources
* Env.prop’s 🡪 no profile mentioned
* Env.test.prop’s 🡪 test configuration test profile is mentioned
* Env.prod.pro’s 🡪 prod “

1. Through maven settings(.m2/)

<activeProfile>Test</activeProfile>

1. Based on default variable

Update test profile mentioned in pom.xml. add activation element to profile element

<Profile><id>test</id></profile>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

</profile>

1. Os settings

Activation element to include os details.this triggers when the system in windows

<activation>

<os>

<name>Windows</name>

<family>windows

</os>

</activation>

Repository

---------------

Place, i.e directory,where all project jar’s,library jar’s,plugin’s,project specific artifacts are stored

3 types

1. Local 🡪 User Home Dir
2. Remote 🡪Developers custom repo,project jar’s ,lib’s

POM.xml

<Repositories>

<url>

1. Central 🡪 Maven community

Snapshot

--------------

Special version, indicates current development copy. Maven checks new snapshot version in remote repo for every build. Automatically fetches latest every time

Release Version

---------------------

If once version is downloaded never try to download a newer 1.o.if it need to be downloaded it must be upgraded to 1.1

Pom.xml

----------

<dependencies>

<group id,artifact id>

<version>

<scope>test

<system path>$basrDir\src\lib\lp.jar

External Dependencies

-------------------------------

Dependencies not find in remote and central then use this as <system path>.its relative to the project location

Dependency Scope

-----------------------

1. Compile

Dependency is available in the class path of the project .default scope

1. Provided

Dependency is provided by jdk/web server/container at runtime

1. Runtime

Dependency not required for compilation but needed for execution

1. Test

Dependency need only for test compilation and execution phases

1. System

Provide the system path

Super(Base) POM inheritance

---------------------------------------

Inherits from super pom

Child pom file add this top of a pom file

<parent>

<g,a,v,relative path>

</parent>

**Pipeline:**

-------------

1. Pipeline script 🡪 Groovy Script
2. Pipeline script from SCM 🡪 Declarative script

The**reuseNode** is a Boolean and on returning true, the docker container would run on the agent specified at the top-level of the pipeline, in this case the agent specified at the top-level is ‘any’ which means that the container would be executed on any available node. By default this Boolean returns false.

There are some restrictions while using the parallel directive:

* A stage can either have a parallel or steps block, **but not both**
* Within a parallel directive you cannot nest another parallel directive
* If a stage has a parallel directive then you cannot define ‘agent’ or ‘tool’ directives
* There are two ways for setting up the slaves.
* 1. Using username and password
* Head over to Jenkins dashboard –> Manage Jenkins –> Manage Nodes
* Select new node option.
* 2. Using ssh keys.
* Add The Private Key To Jenkins Credential List
* Go to jenkins dashboard –> credentials –> Global credentials –> add credentials , select and enter all the credentials as shown below and click ok.

It is very important to have [Jenkins](https://jenkins.io/) backup with its data and configurations. It includes, job configs, build logs, plugins, plugin configuration etc.

[Jenkins Thin Backup](https://wiki.jenkins.io/display/JENKINS/thinBackup) is a popular plugin for backing up Jenkins. It backs up all the data based on your schedule and it handles the backup retention as well.

Webserver 🡪 uses http only

App server 🡪 uses http,tcp,rmi

Tomcat dir’s

----------------

Webapps

Bin(startup.sh,shutdown.sh)

Logs(catalina.out,localhost.dt.log)

Conf(server.xml,tomcat-users.xml)

Lib(library)

Temp

Work

Git

-------

Git status

Git add .(stage to the file)

Git commit –m “hello”

Git mergetool(merging conflicts)

Git branch namr

Git checkout brname

Git rm

Git log filename—author =name/ --oneline/shortlog

Master 🡪root dir

Tags🡪minor releases,read only

Git branch –a

Git branch –v(display info)

Git branch –r(remote branches)

Git push origin master

Git branch –d rel(delete branch)

Git diff r1 r2 (diff branches)

Git clean(remove files)

Git pull = fetch + merge

Stash

===

Stores unstaged changes

Git stash

Git stash list(list stashes)

Git stash apply stash@{0}

Git stash drop stash@{0}

Git add remote origin <https://git.git>

Git push –u origin master