1. DevOps and Build Release Engineer

# In

Ls ==> list the files and directories

ls –a ==> to view the hidden file and directories

ls –l ==> listing files and directories in long list format

ls –lt ==> recently modified files will show above

ls –lrt ==> recently modified files will show bottom

total in ls -lrt ==> disk usage of listed files ( sequence - permissions, linkcount, owner, group, filesize, modifieddate, filename)

pwd ==> present working directory

touch filename ==> to create file ex: touch vkm --- vkm named file is created

touch {vkm1,vkm2,vkm3} ==> to create multiple files at a time

**mkdir** directoryname ==> to create directory ex: mkdir temp -- temp named directory is created

mkdir {temp1,temp2,temp3] ==> to create multiple directories in a single lane

mkdir -p dir1/dir2/dir3 ==> to create multiple subdirectories (here dir1 is parent directory)

vi vkm ==> to edit the file

esc :wq! to save the edited file and force quit

esc :q! the edited content will not be saved

%s/string1/string2/g ==> string1 will be replaced by string2 in all the lines

1,3s/s/string1/string2/g ==> string1 will be replaced by string2 in first 3 lines

$s/string1/string2/g ==> string1 will be replaced by string2 in last line

1s/string1/string2/1 ==> the very first string1 will be replaced by string2 in first line

**esc** :line number ==> ex: esc :4 cursor will move to the 4th line

esc dd ==> to delete particular line in vi

esc :set nu ==> set line numbers

esc :set nonu ==> remove line numbers

**cat** vkm ==> to see the content of the file (ls - to see the content of directory)

cat dir1/dir2/dir3/file1 ==> to see the content of a file inside the directory dir3

cat file1;cat file2 ==> to see the content of multiple file

cat -n file ==> to see the content of the file along with numbers

cat \* ==> to see the content of all the files

cd temp ==> enter into directory (change directory)

cd .. ==> come out of directory (cd ../../.. ==> come out as many as directories)

cd - ==> come to previous path ( acts like switch between two channels)

**copy command**

cp f1 f2 ==> copy content of file1 to file2

cp \* destination path ==> it select all the files to be copied

cp source destination (ex: cp temp1/temp2/temp3/file1 dir1/dir2/dir3/ ---- file1 will be copied to dir3)

cp -r ( to copy multiple directory and files -r is used - it indicates recurssive) ex: cp -r dir1 temp1/temp2/temp3/

**move command**

mv ==> to move or rename (it works like cut and paste) there is no source only destination is there.

mv file1 file2 file3 dir1 ==> to move multiple files in dir1

mv dir1/file . ==> to move the file to pwd . is used

suppose we were in some other directory we want move a file from home to that directory we should use command

mv /home/ec2-user/filename . ==> file from home is moved to pwd

\* we dont have any -r command in move command.

**remove/delete command**

rm file1 ==> to remove/delete file

rm -r dir1 dir2 ==> to remove/delete dir1 dir2

rm -rf ==> to remove directories and subdirectories in arguement list

rm file .file ==> to hide the file

## chmod command ==> to change permissions of files or directories

- indicates file

d indicates directory

rwx indicates read write execute (those are permissions)

rwx|rwx|rwx indicates owner group and others permissions to read write and execute (here all permissions is given to all)

chmod 777 ==> command used to give all permissions

chmod -R 777 ==> to change the permissions of directory and the files present inside the directory

chmod u+rwx g+rwx o+rwx filename ==> if we do not have access to see the permission of a file but still we need to change the permission

chmod a+rwx ==> to grant permission for u g o

## Disk Usage

du -sh ==> to check the file size (d-disk u-usage s-size h-human readable format)

du -sh file1 file2 ==> to check the size of two files

du -sh \* ==> check size of all the files present in directory

df -h ==> to check the size of disk (d-disk f=free)

df -h . ==> shows the present disk size we are working

**Echo**

echo "text to be displayed" ==> these command is used to print on the screen

echo "text to be printed" >> file ==> text which is to be printed will be saved in file

echo -e "text to be printed \ntexted to be printed" ==> \n represents next line -- to do this -e command is used

## Word Count

wc -l filename ==> it shows no of lines present in that file

wc -w filename ==> it shows no of words present in that file

wc -c filename ==> it shows no of characters present in that file

ls -lrt > file ==> output of ls -lrt will be stored in file (here content present in file will be overwritten)

ls -lrt >> file ==> content doesnt overwrite

free -h ==> system memory

ls -lrt | tail -1 ==> to display recently modified file

here | (pipe) works as output of one command will passed to a next command as input

## Head & Tail

head -4 file ==> to print first 4 line of a file

tail -4 file ==> to print last 4 line of a file

head -95 file | tail -1 ==> to print 94th line of a file

head -5 file | tail -1 ; head -8 file | tail -1 ==> to print 5th and 8th line at a time.

head -5 file | tail -1 | wc ==> to count how many lines words characters in 4th line of a file

\* to count only lines words and characters separately we use wc -l wc -w and wc -c commands after pipe

## Grep

grep -- global regular expression print

this command searches for a regular pattern of characters and display all lines that contain the patterns

grep "class" file ==> in which of the line 'class' word is present those lines are displayed

grep -n "class" file ==> the line with word class (which we want to search/display) and along with particular line no will displayed

grep -i ==> case sensitive free

grep -e "word1" -e "word2" file ==> to search multiple words/strings and to diplay the lines that contains that word

grep -w "linux" file ==> search only the word not the pattern (here only linux will be displayed and not the pattern of linux)

ex: if linuxos is present it wouldnt search like other grep command

grep "^i" ==> to print the line which is start with letter i

grep -e "^l" -e "-w" -e "h" file1 file2 ==> to print the multiple line starts with multiple characters with different files

grep "i$" ==> to print the line which is ends with letter i

grep -R -l "linux" \* ==> to search/list the word 'linux' present in which file

grep -c "linux" filename ==> to count the no of linux word present in the file

grep -c "^" file name ==> to count the no of lines present in the file

grep -c "^$" filename ==> to count the no of empty lines present in the file

grep -c -v "^$" file name ==> to count the no of non empty lines present in the file (^ starts with nothing $ ends with nothing)

grep -v "linux" filename ==> the lines without the word linux is printed

grep -v "^$" filename ==> it will print only non empty lines

## SED

sed command -- stream editor (without opening the file we can replace and delete the string and lines)

sed 's/linux/windows/g' filename ==> here linux will replaced by windows but it only displays and not edited

sed 's/linux/windows/Ig' file ==> I is used for case sensitive free

sed 's/linux/windows/g' file1 > file2 ==> here linux will replaced by windows and the edited content will saved in file2

sed 's/old line/new line/g' file1 ==> here old line is replaced by new line

sed '1s/linux/windows/g' file ==> here linux will replaced by windows only in first line of the file but it doesnt print only that line

sed '$s/linux/windows/g' file ==> here linux will replaced by windows only in last line of the file but it doesnt print only that line

sed '1,6s/linux/windows/g' file ==> here linux will replaced by windows from 1st to 6th line --''-

sed -i 's/linux/windows/g' filename ==> here the actual file is going edited and it wont display

sed -n '4p' file ==> to print the 4th line of a file and it wont change the actual content of the file

sed -n '$p' file ==> to print last line of a file

sed -n '1,4p' file ==> to print first four lines (1 to 4)

sed -n '1p;4p' file ==> to print only 1st and 4th line (1 and 4)

sed '2d' file ==> to delete the 2nd line and it wont change the actual content of the file

sed -i '2d' file ==> it delete the 2nd line and it changes the actual content of the file

sed '$d' file ==> to delete last line of the file

sed '1,4d' file ==> to delete first to fourth line of a file

sed '1d;4d' file ==> to delete 1st and 4th line of a file

sed '$a line to be inserted' file ==> extra line to be inserted will be printed at the end of the file

sed '1a line to be inserted' file ==> extra line to be inserted will be printed after the first line of a file

## CUT

cut -- command used for selecting a specific column of a file

cut -d " " -f1 file ==> it will print the first column of a file (d - delimiter f1- field 1)

cut -d " " -f1,3 file ==> it will print first and third column only

cut -d " " -f1-3 file ==> it will print first three columns.

limitations

It works only if no of characters of each column is same

## AWK

awk -- is a powerful command it can used to cut the file column wise we can replace strings, we can search for strings and we can write script also.

awk -F " " '{print $1}' file ==> it will print first column of a file (F - field separator)

awk '{print $0}' file ==> it will print all the lines (all columns)

awk '{print NR, $0}' file ==> it will print all the lines with line no

awk '{print NR "-" $0}' file ==> it will print all the lines with line no along with -

awk -F " " '{print $1,$3}' file ==> it will print 1st and 3rd column of a file

awk -F ":" '{print $1}' file ==> here : is field separator and we have only two fields one is left to the : and another right to the :

if we have two : then there is three field separator

awk -F " " '{print $NF}' file ==> it print last column of a file

awk -F " " '{print $(NF-1)}' file ==> it print last second column of a file

awk '{print $3,$NF}' file ==> it print 3rd and last word/column of all the line

awk '/linux/ {print}' file ==> it will print the line which has word linux

awk 'NR==1, NR==3 {print $0}' file ==> it will print first 3 rows along with all the column of a file

## Soft link & Hard Link

Is a shortcut to a file, if you make any changes in original file it gets reflected in the link, if I delete original file soft link wont work.

Ln –s filename soft link name U can create a file or dir

**Hard link**

Ln filename hardlinkname

Is a shortcut to a file, if you make any changes in original file it gets reflected in the link, and if I delete the original file, hard link wil work because it points to inode of a file.

* + **Xargs**

Is used to pass as a args to a next command

Output of one command will be passing as args to next command. how to find files which are modified 90 dyas back and del that.

Find . –type f –mtime +90 | xargs rm

* + **Exec command**

Find . –type f –mtime +3 –exec

Is used to replace the current shell with the command without spawing a new process and used to assign file descriptor to file

## Umask

Is used to set default permission on a system. It is reverse of chmod.

Umask 000 –who were creates the file on a system will have all permissions.

Umask 777 means- no permission to a file

## Ping

Is used to check whether the server is up and down Ping ip or hostname

## telnet

It uses port 23, is used to authentic account credentials on remote server used to break firewall

telnet ip or hostname

## rsync

is used to copy files from one server to another server and also within the server, while copying the data if copy is stopped due to network issues in between, if I use ‗scp‘ it will start copying from begin

if I use rsync it will start copying where it is stopped. Rsync file user@server2:/home/test/../

If you want to copy in server Rsync src dest

* + su

super user

su –root or su –

used to login as root user

root has highest permission

* + **sudo** superuser does

is used to run command with root permissions sudo username

## who

is used to check who are all logged into system who

who | wc -l

## whoami

is used to check, who you logged in name whoami

## uname

to check linux version

* + uname –a

complete information about linux

* + uname –v

version and time of linux

* + ps

used to list current running process on a system ps –ef | grep ―system32‖

## kill

used to stop process

-9 is a special signal to terminate that process Kill -9 PID

Killall -9 pname

Sudo service servername/Pname stop Sudo service servername/Pname start Sudo service servername/Pname restart

Pkill -9 –u ―username‖ – stop process started by particular user

Ps –u username or ps –ef | grep ―system32‖ – list process started by particular user

How do you run script or command in a background we need to see & at the end of the command.

How do you run process in foreground Fg PID

## sort

is used to sort the data sort filename

sort –r filename

* + **uniq**

used to print only unique values (it removes duplicate)

it will work only when the contents/data are consecutive values/info/data uniq will be always used with the sort

**uniq** filename

cat file | sort | uniq

## tee

used to write output of a command to a file as well as display on console/prompt

cat log | tee log1

ls –lrt | tee –a log 🡪used to append to log

## Netstat

Used to check free ports on a system. List all listening ports tcp or udp Netstat –a

Netstat –at Netstat –au Netstat –lt Netstat -lu

* **Mount**

Attaches a filesystem, located on same device or other to a file tree. Mount –t type device dir

Unmounts-detaches the specified file system from file hierarchy

## How do you setup passwordless connection between 2 servers?

Server1 🡪 server2

Ssh-keygen

I will be getting private and public key

From server1 copy the public key of server1 to server2 in .ssh/authorized\_keys Then u can access server2 from server1 passwordless connection.

We can use winscp plugin or software to connect to sever sfrom password less connection.

We need to run ssh keygen on server1 it generates the key we need to copy public key to authorized keys files under .ssh folder. So we can connect s2 from s1 without password.

## Linux Directory Structure

* /bin – binary or executable programs.
* /etc – system configuration files.
* /home – home directory. It is the default current directory.
* /opt – optional or third-party software.
* /tmp – temporary space, typically cleared on reboot.
* /usr – User related programs.
* /var – log files.
* /boot- It contains all the boot-related information files and folders such as conf, grub, etc.
* /dev – It is the location of the device files such as dev/sda1, dev/sda2, etc.
* /lib – It contains kernel modules and a shared library.
* /lost+found – It is used to find recovered bits of corrupted files.
* /media – It contains subdirectories where removal media devices inserted.
* /mnt – It contains temporary mount directories for mounting the file system.
* /proc – It is a virtual and pseudo-file system to contains info about the running processes with a specific process ID or PID.
* /run – It stores volatile runtime data.
* /sbin – binary executable programs for an administrator.
* /srv – It contains server-specific and server-related files.
* /sys – It is a virtual filesystem for modern Linux distributions to store and allows modification of the devices connected to the system.

## Zombie Process:

If child process fails and doesn’t return the status to parent process then its called as zombie process

ps aux | awk '$8=="Z" {print}'

## Orphan Process:

A child process that remains running even after its parent process is terminated or completed without waiting for the child process execution is called an orphan.

ps -eo pid,ppid,cmd | awk '$2==1 && $1!=1 {print}'

## Daemon process:

Daemon processes are started working when the system will be bootstrapped and terminate only when the system is shutdown. It does not have a controlling terminal. It always runs in the background

## Difference between bash profile and bash rc

Both the bash profile and bash rc files are used by the Bash shell to set up the environment and define various settings when a user logs in to a shell session.

The main difference between the two is that the **bash profile file** is typically executed only once when the user logs in, while the **bash rc file** is executed every time a new shell is opened.

The **bash profile file** is used to set up user-specific environment variables, aliases, and functions, and to run any scripts that need to be executed only once, such as installing software or setting up the system for a specific user.

On the other hand, the **bash rc file** is used to set up shell-specific configurations, such as setting the prompt, defining shell functions or aliases, and setting other shell options.

In summary, the **bash profile file** is used for user-specific settings that only need to be executed once, while the **bash rc file** is used for shell-specific settings that need to be executed every time a new shell is opened.

# Shell Scripts

To execute-> ./ex.sh or sh ex.sh or bash ex.sh

* If we wont write she bang it uses default shell.
* 1st line of shell scripts called shebang.

## Notations

**$#** - to display total number of arguments

**$\*** - all arguments passed to a scripts

**$?** - status of last executed command

If it is 0, success. Else failure (1,2,)

**$$** - display the present PID.

**$@** - all arguments passed to a script are stored in array format.

**$!** - PID of current running process, which went into background. #-is used to commenting a code.

&&-AND,||-OR,!-NOT.

## Crontab

Is a scheduler, which is used to schedule scripts on Linux server. Crontab –e ---is used to edit cron job

Crontab –l ---is used to list cron jobs

\* \* \* \* \* /script path Mins|hrs|date|month |day of week

* Schedule to run the script on specified date and time
  + 12th June at 10 am mon

00 10 12 06 01 /script path

* + 13th June at 4 pm tue

00 16 13 06 02 /script path

* + Every day at 2pm

00 14 \* \* \* /script path

* + 03.30 pm only on tue

30 14 \* \* 02 /script path

* + Mon-fri at 4pm

00 16 \* \* 01-05 /script path

* + Mon and Tuesday at 5.30pm

30 17 \* \* 01,02 /script path

* + Mon every one hour

00 \* \* \* 01 /script path

* + Mon every 30 mins

\*/30 \* \* \* 01 /script path

* + Mon at 4pm, script should run at every 5min

\*/5 16 \* \* 01 /script path

* + 05.15pm one the friday

15 17 \* \* 05 /script path

* + Sat 10pm

00 22 \* \* 06 /script path

* + Every month 1st at 11 am

00 11 01 \* \* /script path

or Sudo yum install update And to check crontab use, which crontab

then,

/home/…/.//big3.sh > log1 Crontab –e

\*\*\*\*\*/script path Crontab –l

* + Need to set a crontab every month end at 11 am

00 23 \* \* \* [ `date +%d` -eq `echo \`cal\` | awk ‗{print $NF}‘` ] && job.sh

## Realistic Scripts

* Builds (different compilation or developing files) were failing due to memory issue it was giving an error like ―no space on disk‖. To resolve this problem/issue,
* **I have written a script to check disk memory and send a email notification, saying that memory is 90%, and please take appropriate action.**

Vim memch.sh

#!/bin/bash

Mem=`df –h . | tail -1 | awk –F ― ― ‗{print $4}‘ | sed ‗s/%//g‘` If [ $mem –gt 40 ]

Mail –s ―memory reached‖ –c [pradeep@gmail.com](mailto:pradeep@gmail.com) < filename

Fi

--------filename at the end of mail line is,

You can write a body of mail in that file can read to the mail command. Else U can try like,

Echo ―hi memory reached‖ | Mail –s ―memory reached‖ –c [pradi@gmail.com](mailto:pradi@gmail.com)

To run memch.sh for every min, Crontab –e

\* \* \* \* \* /memch.sh

* Mail command

Is used to email the file or information to the specified email.

Mail –s ―memory reached‖ –c [pradi@gmail.com](mailto:pradi@gmail.com) [pradeep@gmail.com](mailto:pradeep@gmail.com) < filename

Echo ―hi memory reached‖ | Mail –s ―memory reached‖ –c [pradi@gmail.com](mailto:pradi@gmail.com) [pradeep@gmail.com](mailto:pradeep@gmail.com)

Mail –s ―memory reached‖ ―body of mail‖ –c [pradi@gmail.com](mailto:pradi@gmail.com) [pradeep@gmail.com](mailto:pradeep@gmail.com)

## REALISTIC SCRIPT 2 – Service installation

* They wanted me to write a **script** to monitor the ‗**Tomcat‘ services** on the server, if any service or process is stopped accidently by someone, we should get email notification saying that **service is stopped** and also script should restrict the service.-------------------realistic script example-----------------

Services=”ser1 serv2 serv3 ―

For service in $services

Do

Ps -ef | grep “$service”

If [$? -ne 0]

Then

Mail -s “service is stopped and trying to restart

Sudo service $service restart

Fi

Done

## REALISTIC SCRIPT 3 - Logs

**Collect all the logs and deletes files older than X days**

#!/bin/bash

prev\_count=0

fpath=/var/log/app/app\_log.\*

find $fpath -type d -mtime +10 -exec ls -ltrh {} | rm -rf

count=$(cat /tmp/folder.out | wc -l)

if [ "$prev\_count" -lt "$count" ] ; then

MESSAGE="/tmp/file1.out"

TO="daygeek@gmail.com"

echo "Application log folders are deleted older than 15 days" >> $MESSAGE

fi

# GIT

* Git is a distributed version control system that uses a specific architecture to manage repositories and versions of code
* Type: - GIT, SVN, clearcase, TFS, perforce, CVS, Mercury
* **Workspace**
  + It’s a place where we edit modify project related files

## GIT Arch

* + Workspace 🡪 Adding area🡪Commit area
* **Working directory**: The working directory is a copy of a specific version of the project stored on the local machine. It contains the files and folders that make up the current version of the project. Developers can make changes to files in the working directory.
* **Staging area**: The staging area, also known as the index, is a space where changes to files are staged before they are committed to the local repository. The staging area allows developers to review changes before they are committed and to selectively commit changes.
* If I run git add files will be moved to staging area, it‘s a intermediate area, where we can save the changes.
* **Local repo**: A commit is a snapshot of the changes made to the files in the working directory. A commit includes a message that describes the changes made, and it is added to the local repository's version history
* **Git init** (used to create non-bare repo)
* Vi test1 (add some content)

**Git status** (used to check whether files are workspace, staging area or in git repo

* Git add test1 (this will move file from workspace to staging area)
* **Git status** (it will show changes to be committed)
* Git commit –m ― ― (this will move files from staging area to git repo)
* **Git status** (this will show working directory clean)
* **Git log** (used to check the repo history)
* Git log filename (specified filename)
* Git log -2 (last 2 commits)
* **Git checkout commitId** (used to switch to a previous version, used to switch to branch, also to switch to tags)
* **Git checkout master** (gives to the latest version)

## Tags

* Tag is a name given to set of versions of files and directories. It indicates milestones of a project we can easily remember tags in the future. If I want good code in the future, we tag it.
* Command to list tags is,
  + Git tag
* To create a tag,

Git tag tag name (it tags the latest version of code by default)

## Branching

* Is a parallel development, two teams will work on same peace of code on two different branches, later they can integrate the changes by merging
* **Git branch** – to list the branches
* Git branch branch name –to list branch names
* To switch to a branch name
  + Git checkout branch name

## Git merge

Git merge is a command used to combine changes from one branch into another branch. The merge command creates a new commit that combines the changes from two or more branches.

**When you run the merge command, Git will:**

* + Identify the common ancestor of the two branches.
  + Create a new merge commit that combines the changes from the two branches.
  + Apply the changes from the merged branch to the target branch.

## Merge Conflicts

Will occur when the same peace of code is changed on 2 different branches, when we try to merge those two branches, then merging conflict will occur,

To resolve this issue, I don ‘t know whose change should I take to merge, so I contact developers changes the code, person who modified code of branch1 and branch2. Then they will decide and tell us whose changes should I take into merge.

Then I take that change and I commit it. I get to know who modified the code on branch1 and branch2 using git log command.

* **To delete branch or tag**
  + git branch –d branch name
  + git tag –d tag name

## Git Rebase

* Git rebase is a command used to apply changes from one branch to another branch. The rebase command moves the entire branch to a new starting point by applying each commit in the branch on top of the new starting point.

**When you run the rebase command, Git will**:

* Identify the common ancestor of the two branches.
* Remove the commits that exist only in the current branch.
* Apply the commits in the current branch to the new starting point.
* **Git rebase branch name**
  + Latest commit id is called as HEAD,(is a internal tag)

If you want to apply the changes from the **"feature-branch" to the "main"** branch, you would use the following commands:

git checkout main

git rebase feature-branch

Here **Git merge is the best way** to merge the code because rewriting history will be avoided.

**Here are some situations when git rebase can be useful:**

**Updating a feature branch**: When working on a feature branch, you may want to periodically update it with the latest changes from the main branch. You can use the rebase command to apply the changes from the main branch to the feature branch, keeping the feature branch up to date with the latest changes.

**Cleaning up the commit history**: If the commit history of a branch is messy and contains many unnecessary commits, you can use the rebase command to clean up the history and make it more readable.

**Squashing commits**: If you have several small commits that are related to the same feature, you can use the rebase command to squash them into a single commit. This can make the commit history more concise and easier to follow.

**Resolving conflicts**: When two branches have conflicting changes, you can use the rebase command to resolve the conflicts and apply the changes from one branch to the other.

## Difference between merge and rebase

Git merge creates a new merge commit that combines the changes from the source branch and the target branch. It preserves the history of the source branch.

Git rebase, on the other hand, applies the changes from the source branch on top of the target branch. This creates a linear history with no branch points and can make the project history cleaner and easier to follow. However, rebasing can also rewrite the project history and create conflicts if other developers have already pulled the original commits.

**Here are some key differences between merge and rebase:**

* History: Merge creates a branch point in the history, while rebase applies changes on top of the target branch, creating a linear history.
* Commits: Merge creates a new merge commit, while rebase rewrites the commit history.
* Conflicts: Merge can result in merge conflicts, while rebase can result in conflicts when applying changes.
* Branches: Merge is good for combining changes from long-lived feature branches, while rebase is good for cleaning up the commit history and combining the commits

## Git revert

* Git revert is used to create a new commit that undoes the changes made by a previous commit. It is useful when you want to undo changes made to a repository without deleting any of the history.
* When you run git revert, Git will create a new commit that reverses the changes made by a specified commit. The new commit will contain the opposite changes made by the specified commit, effectively undoing the changes.
* git revert <commit>
* Once you run git revert, Git will open up a text editor so you can enter a commit message for the new commit that will be created.
* It's worth noting that git revert does not delete the original commit or the changes made by it. Instead, it creates a new commit that undoes those changes. This means that you can always go back to the original commit if needed.

## Git reset

Git reset command that is used to undo changes and move the current branch to a different commit. Here are the different types of git reset explained with examples:

**Soft reset**: git reset --soft moves the current branch to a different commit, but leaves the changes in the working directory and staging area. This is useful if you want to undo a commit and keep the changes ready to be committed again.

Example: git reset --soft HEAD~1 moves the current branch back one commit, but keeps the changes in the working directory and staging area.

**Mixed reset**: git reset --mixed is the **default** behavior of git reset. It moves the current branch to a different commit and resets the staging area, but leaves the changes in the working directory. This is useful if you want to undo a commit and make changes before committing again.

Example: git reset HEAD~1 moves the current branch back one commit, resets the staging area, but keeps the changes in the working directory.

**Hard reset**: git reset --hard moves the current branch to a different commit and discards all changes in the working directory and staging area. This is useful if you want to completely undo all changes since the last commit.

Example: git reset --hard HEAD~1 moves the current branch back one commit and discards all changes in the working directory and staging area.

## Git stash

* + If I am working on one branch, if I get some critical work, which needs to be fixed on the other branch. I don ‘t wants to commit changes in other branch, as I not completed the work, I will do git stash and I switch to other branch I will fix the issue and I come back to the previous branch to continue my work. I need to run git stash pop.
  + Git stash will save files somewhere in temporary area. It will not store in working space, staging area or git repo.
    - Touch f1 f2 f3
    - Git add \*
    - Git commit –m ―files added‖
    - Switch to other branch and check.
    - Git stash – then come back to other branch and try
    - Git status pop

## Git cherry-pick

* + It allows us to apply a specific commit from one branch onto another branch. It is useful when we need to copy a single commit from one branch to another, without merging the entire branch
  + If I want to commit 3 commits use, git cherry-pick commitid1 commitid2 commitid3
  + This command is useful to undo changes when any commit is accidentally made to the wrong branch. Then, you can switch to the correct branch and use this command to cherry-pick the commit.

## Git conflict

* + Is same as merge conflict
  + Sometime you get merge conflicts when merging or pulling from a branch.
  + Git will then tell you something like conflicts(content) merge conflict in fake file.
  + It also tells you to fix conflicts and commit the changes.

## Git bisect

* + Bisect allows us to efficiently find a specific commit in our project's commit history that introduced a bug or caused a regression
  + Git bisect <subcommand> <options>
  + This command uses binary search algorithm to find which commits in your project’s history introduced a bug.
* There are 2 kinds of repository in git
  + Bare repo
  + Non-bare repo

### Bare repo

* + It acts as a central-repo, we can only push and pull the changes to the repository.
  + In bare repository, you can’t run any git operation on bare repo.
  + It contains only commit history, branch and tag names, and other git objects.
  + Git init –bare my-project.git

### Non bare repo

* + It’s a local repo we can modify the files and push to a central repo we can run all the git commands. Command to create a non-bare repo is git init
  + Command to create a bare repo is
    - Git init

## Git clone

* + Bringing the remote repo to local workspace for the first time called as git clone.
  + Git clone URL

There are several options that you can use with the git clone command, including:

* + **--depth**: limits the number of commits that are cloned, which can make the clone faster and use less disk space.
  + **--branch**: specifies a specific branch to clone, instead of the default branch.
  + **--recursive**: clones submodules along with the main repository.

## Git push

* + Moving the changes from workspace to remote repo or central repo
  + Git push user@severname: /path.././central\_repo

## Git pull

* + Bring the changes from remote repository and merges to local repo automatically.
  + Git pull user@servername: /home/./path/central\_repo

## Git fetch

* + Bring the changes from remote repo and stores it on a separate branch, you can review the changes and merge normally if it is required.
  + Git fetch user@servername:/home.//./central\_repo/
  + Git pull = git fetch + git merge

|  |  |
| --- | --- |
| * + Git Fetch | * + Git Pull |
| * + The Git fetch command only downloads new data from a remote repository. | * + Git pull updates the current HEAD branch with the latest changes from the remote server. |
| * + It does not integrate any of these new data into your working files. | * + Downloads new data and integrate it with the current working files. |
| * + Command - git fetch origin   + git fetch --all | * + Tries to merge remote changes with your local ones.   + Command - git pull origin master |

## Git reflog

* + This command is useful for recovering lost commits, branches, or other references that were accidentally deleted or lost.

## git branch --merged

* + Returns the list of branches that have been merged into the current branch.

## git branch --no-merged

* + Returns the list of branches that have not been merged.
  + A head is nothing but a reference to the last commit object of a branch

## Git squash

* + This allows us to combine multiple commits into a single commit. This is useful for cleaning up our commit history, making it easier to understand the changes that were made to the code over time
  + To squash the last 3 commits into one:
  + git reset --soft HEAD~3
  + git commit -m "New message for the combined commit

## Difference between git and cvs

* + Git
* Is a distributed version control system
* It means whole repository will be there in the local workspace
* If I want to go previous version of file, I can go directly in local workspace itself.
* Git has many advanced features like rebase, fetch, stash, merge etc.
* These advance features are not there in CVS.
* CVS
* Is a centralized repo and also SVN also.
* If I want to go to previous version of a file, I need to checkout from centralised\_repo because initially it will have only one version of a file in local repo.

## MAKE FILE

* + Works on time stamp basis, if target time is less than the dependencies time it will re- generate the target, that means make file will compare target time with its dependencies time.
  + If target time is latest than its dependencies it will not re-generate the target.
  + If there are thousand files, if I change 5 files, only 5 files will get compiled and incorporated at the build.

Patch build

* + It is a critical fix which needs to be deliver to a customer within few hours. Developers will change only required files, make will compile only changed files and changes will get incorporated to the build so patch build will take less time.

Load build

* + We compile source code from the scratch, before we start this build we delete all intermediate files (.o files), so that all files will get compile from scratch. So it take more time.

# Branching Strategy

* Branches can be created for multiple reasons, here we create branches for releases.
* Development team will be going on development branch, once the code is ready for the first release, we create a release1 branch from dev branch and we make a release (we do build and release it) from this release1 branch.
* Whatever the issues specific to this release will be foxed on release1 branch only. It will act as a maintenance branch for release1.
* Simultaneously development will be going on dev branch for 2nd release. Once the code is ready for 2nd release before we create a branch for 2nd release, we merge, release 1 branch to dev branch and then we create release2 branch for dev branch for 2nd release. So, whatever the issues we have seen in 1sr release should not be visible in the 2nd release and so on
* BVT (build verification test) or sanity test.
* Is a basic functionality of a build which should never break.
* Sanity report.
  + ♣ Is a report related to testing and should checkbox for test.
* Release note

♣ Tag name and description and known issues.

* + Build🡪build success🡪BVT or sanity test🡪sanity report🡪release note🡪test team
  + Build🡪build fail🡪dev team

## Maven and ANT difference

|  |  |
| --- | --- |
| **Ant** | **Maven** |
| Ant **doesn't has formal conventions**, so we need to provide information of the project structure in build.xml file. | Maven **has a convention** to place source code, compiled code etc. So we don't need to provide information about the project structure in pom.xml file. |
| Ant is **procedural**, you need to provide information about what to do and when to do through code. You need to provide order. | Maven is **declarative**, everything you define in the pom.xml file. |
| There is **no life cycle** in Ant. | There is **life cycle** in Maven. |
| It is **a tool** box. | It is **a framework**. |
| It is **mainly a build tool**. | It is **mainly a project management tool**. |
| The ant scripts are **not reusable**. | The maven plugins are **reusable**. |
| It is **less preferred** than Maven. | It is **more preferred** than Ant. |

## Maven and ANT?

* Maven is a project management and comprehension tool. Maven provides developers a complete build lifecycle framework. Development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle.
* Apache Ant is a Java based build tool from Apache Software Foundation. Apache Ant's build files are written in XML and they take advantage of being open standard, portable and easy to understand

# Jenkins

* Jenkins is a CI tool and automated framework is used to integrate development changes automatically without manual intervention.
* Other CI tools – Bamboo, Apache gump, Buildbot, Travis CI
  + Go to Jenkins main page
  + Click on new item
  + Give job name
  + Select job type (free style) & OK
  + Configure page
  + Description of job
  + SCM (source code management
  + Select git
  + Path of git central repo, add branches, credentials

Jenkins will create its own workspace and pulls latest code to it.

## Job Types

* Freestyle Project
* Maven Project
* Pipeline
* Multi configuration project
* Multi branch pipeline

## Build Triggers

Is used to trigger the job based on the time, based on commit or where other jobs complete/success.

* **Trigger Build remotely (eg: from scripts):** This build trigger allows external systems to trigger a build by sending an HTTP request to the Jenkins server. You can specify an authentication token to secure the trigger.
* **Build after other projects are built**: This build trigger allows you to trigger a build of one project after another project has been built. You can specify the other project(s) and conditions (such as success or failure) that must be met.
* **Build periodically**: we need to mention time like a crontab, it will trigger the job based on the time., and also it is a scheduler crontab.
* **GitHub hook trigger for GITScm polling**: This trigger uses the Git SCM polling mechanism to detect changes in the repository and start a build.
* **Poll SCM**: will trigger the job based on commits within the specified time job will get triggered based on commit.

If there is a commit in between an hour, job will get triggered else simply without commits it won’t trigger.

## Build step

* This section is used to compile and make a target file, and also compile and generate a binary. We can execute shell or invoke ANT.
* If I use execute shell, I can use shell script which internally calls makefile or I can use invoke ANT to compile java source code.
* Conditional step
* Execute windows batch command
* Execute shell
* Invoke Ant
* Invoke Gradle script
* Run with timeout
* Invoke with top level Maven targets
* Trigger build on other projects

## Post build action

* This is used to post build activities after build a project, like deploying build, running test cases or send a email notification or copying the build to shared path.

## Master slave

* Master: server on which Jenkins has installed is called master.
* Slave: server on which we run the jobs from the master is called slave.
* Why master-slave or slave machines required?
  + To distribute the load to a different server from the master we go for master-slave.
  + We need to run specific job on specific environment for example, java project/java job we need to compile it on server which has java environment we go for master slave/slave machines.

### How to create slave machines/nodes?

* + Jenkins
  + Manage nodes
  + New node
  + Give node name and description
  + Number of executors
  + Label name
  + Root directory
  + Launch method (SSH), HOST, credentials.
  + Number of executors means, we can run specific number of jobs simultaneously on that server depending on capacity of server we specify number of executors.
  + If number of executors is ‗2‘ menas , we can run 2jobs from Jenkins simultaneously.
* How to run this slave on the new job or already created job or on the server
  + Got to configure page of job
  + Configure
  + Restrict this project can run
  + Mention the node name/slave name
* We go for labels to run job on available server, we need to add all server names to a same label. We need to mention the label name in the job instead of node name.
* This will work in round robin passion.

## Parameterized plugins

* + Is used to give or pass parameters or input to a job at the run time. We have deployment job, we run that job based on request from dev team or development team on test team, at the running the deployment job we need to select build number and server name, these 2 parameters will be taken as input to a job.

## Build pipeline (upstream and downstream)

* + Used to trigger one job after the other and we can pass parameter from one job to another using upstream and downstream plugins.
  + In the build now page
  + Configure page for build job
    - Upstream-
    - Downstream-deploy job
  + Configure page for deploy job
    - Upstream-build job
    - Downstream-test job
  + Configure page for build job
    - Upstream-deploy job
    - Downstream-
  + Build job will get triggered by commit, once the build job is success it will automatically trigger deployment job.
  + If deployment job is success it will automatically trigger testing jobs, we can trigger job serially using upstream and downstream plugins.

**Gearman plugins**

* + It is a high availability plugin, if jenkin master goes down,jenkin will moves go down if I install this plugins.
  + German plugin will allow you to configure other server details, if the master goes down other serverwill act as a master. Both the server will be in a sink frequently.
  + Other server will allows have complete Jenkins backup. So jenkin will never go down.

**What is the most challenging task you done it?**

* + When I installed Jenkins for 1st time on the server, after a month server got crashed due to some hardware issue, server didn‘t come up at all. I lost the Jenkins builds were stopped from the day, we couldn‘t recover Jenkins. I had to resetup Jenkins on the other server.
  + Now I searched for the high availability plugin, I got to know about GEARMAN PLUGIN. I installed gearman plugin in Jenkins and I configured other server details in it. If the master goes down, other server will act as a master. So Jenkins will never go down now.

**How do you take Jenkins backup?**

/var/lib/Jenkins

* + We create separate git\_repo for Jenkins and can push Jenkins configuration files to git\_repo. Then we can clone it, whenever we need it. ((challenege)).

Jenkins safe restart

* + It will allow to current running jobs to complete but it will not allow new jobs to be trigger, and it will restart the Jenkins, once the current running jobs are completed.
    - http://localhost:8080/saferestart

## Jenkins security

* + How to give security permissions to access Jenkins?
  + How do you add new used in Jenkins?
    - We use matrix based security, it will allow to give required permission for a user by clicking the checkboxes.
    - Manage Jenkins🡪configure global security properties🡪click on enable security(these Jenkins own users db)🡪allow users to signup🡪in authorization section🡪matrix based security will be checked🡪you can add users and give permission security🡪then save it.
  + How do you add environment vaiables?
    - Goto manage Jenkins🡪configure system🡪global env🡪add environment variables.
      * Name:var
      * Value:1
    - We use configure system to configure smtp server allows.
  + Git diff current\_cmtid previous\_cmtid(what it modified)
  + Git show commitid (list all specified modified files in specified commits)

## CI CD Continous Integration, Deployment, Delivery

## Continous Integration

* + Is a continuous integration, integrating the development changes continuously without manual intervention.
  + As soon as development team commits the codeto a git, build job get triggered and build will get generated automatically.

CD (Continuous Deployment)

* + Is a continuous deployment, deploying the build to testing environment automatically or less manual effort.
  + Each change from development team is build and deployed to test environment automatically.

CD (Continuous Delivery)

* + Is a continuous delivery, releasing deploying tested build to a production environment as quickly as possible.

## Maven Life cycle

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

**Mvn clean package**: - It will delete previous build files and it will create package with build artifacts

**Mvn clean install**: - it will delete previous build files and it will store build artifacts in local repository

**Mvn clean deploy**: - It will delete previous build files and it will store build artifacts in remote repository

**Steps involved in building an application is ?**

build = compile + assemble + generate artifacts

compile ==> it will convert sourcecode to machine readable format

assemble <linking> ==> it will group link your source code

deliver <generate artifacts> ==> jar file / war file

## POM.xml

It is an XML file that defines the project's configuration, including the project's dependencies, build settings, and other project-specific information.

The pom.xml file is located in the root directory of the project and is used by Maven to manage the project's dependencies, plugins, and build process. It contains information such as the project's name, description, version, and dependencies.

The dependencies section of the pom.xml file lists all the external libraries or modules that the project depends on. When a project is built, Maven will download the required dependencies from a central repository or from a local repository and include them in the build process.

* Project
* Model version
* Group id
* Artifact id
* Version id

**Best practices of SCM (source code management)**

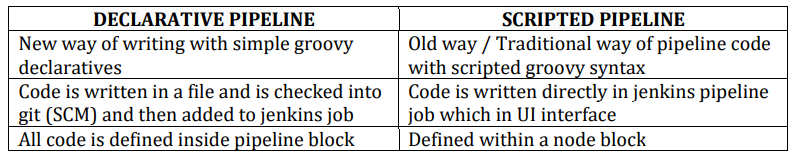
* + Keep workspace clean, mean delete temporary or unwanted files from workspace.
  + Go for branching, if it is necessary.
  + Tag the source code, whenever if it is necessary or the good code.
  + Document the tasks which you do.

## Webhooks

Webhooks are basically user defined HTTP callbacks (or small code snippets linked to a web application) which are **triggered by specific events**. Whenever that trigger event occurs in the source site, the webhook sees the event, collects the data, and sends it to the URL specified by you in the form of an HTTP request.

* Go to your stack, and click on the “Settings” icon on the left navigation panel.
* Click on Webhooks.
* Click on the + New Webhook button located at the top of the page.
* In the Create Webhook page, provide the following Webhook details: ...
* Click on the Save button.
* Jenkins URL need to be used

## Jenkins pipeline scripts and difference

* 

# Jenkins declarative script

A valid Declarative pipeline must be defined with the “pipeline” sentence and include the next required sections like AGENT, STAGES, STAGE, STEPS

**AGENT**

An agent is a directive that can run multiple builds with only one instance of Jenkins.

This feature helps to distribute the workload to different agents and execute several projects within a single Jenkins instance.

It instructs Jenkins to allocate an executor for the builds.

A single agent can be specified for an entire pipeline or specific agents can be allotted to execute each stage within a pipeline. Few of the parameters used with agents are:

Any

Runs the pipeline / stage on any available agent.

None

This parameter is applied at the root of the pipeline and it indicates that there is no global agent for the entire pipeline and each stage must specify its own agent.

Label

Executes the pipeline / stage on the labeled agent.

Docker

This parameter uses docker container as an execution environment for the pipeline or a specific stage. In the below example I’m using docker to pull an Ubuntu image. This image can now be used as an execution environment to run multiple commands

STAGES & STAGE

This block contains all the work that needs to be carried out. The work is specified in the form of stages.

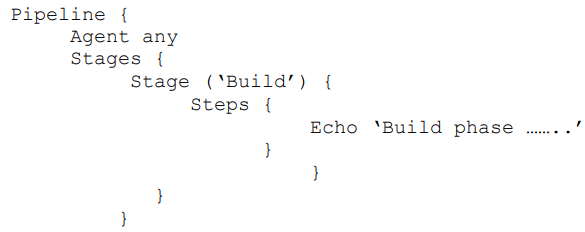
There can be more than one stage within this directive. Each stage performs a specific task.

STEPS

A series of steps can be defined within a stage block.

Steps are carried out in sequence to execute a stage.

There must be at least one step within a step’s directive.



# Shared Library

A shared library is a collection of reusable code that can be shared across multiple Jenkins pipelines.

1. Define the shared library in Jenkins by navigating to Jenkins > Manage Jenkins > Configure System and adding a new "Global Pipeline Libraries" section. Here, you specify the repository URL, version, and a name for the shared library.
2. Load the shared library in your Jenkins file by adding the @Library annotation at the top of your file:

**@Library('my-shared-library') \_**

This will load the shared library code from the repository defined in the "Global Pipeline Libraries" section.

1. Use the greeting step in your pipeline code:

pipeline {

agent any

stages {

stage('Greeting') {

steps { greeting('Jenkins')

}

}

}

}

Backup