SHELL SCRIPTING :-

Linux architecture :-

Kernel :-

→ a computer program that is the core of a OS, with complete control over everything in the system.

It manages the following resources of Linux System :

→ File management
→ Process management
→ I/O management
→ Memory management
→ Device management etc.

* also manages hardware.

Shell :-

→ Shell is basically the program that lets you talk to Kernel/Linux using text commands.

shell file → /bin/sh
↓
Bourne again shell (Bash) → /bin/bash → most common

more example of shells → csh, ksh, fish, zsh

→ shell accepts human readable commands from users and converts them into something which kernel can understand.

Shell Script : a file containing a series of commands for shell to execute.

Commands :-

Vim text editor :-

Vim hello.txt

now press i to get into insert mode & we can now insert inside the file.

after writing, if you want to get out of insert mode press esc.

then : wq → write & quit saves the changes in the file.

: qa → quit without saving.
↳ quit all

: qa → closes all open files in vim & exits the editor.
→ if there are unsaved changes, Vim will not quit and will show a warning.

if you want to force quit, you can use

: qa!

1) cat → display file content

If we want to create a shell script file & probably use vim to write

→ vim myscript.sh
↳ extension

Shebang → it is the very first line in a script, that starts with

#! /bin/bash
↳ path to bash interpreter

Shebang

Commands :-

Vim text editor :-

Vim hello.txt

now press i to get into insert mode & we can now insert inside the file.

after writing, if you want to get out of insert mode press esc.

then : wq → write & quit saves the changes in the file.

: qa → quit without saving.
↳ quit all

: qa → closes all open files in vim & exits the editor.
→ if there are unsaved changes, Vim will not quit and will show a warning.

if you want to force quit, you can use

: qa!

1) cat → display file content

If we want to create a shell script file & probably use vim to write

→ vim myscript.sh
↳ extension

Shebang → it is the very first line in a script, that starts with

#! /bin/bash
↳ path to bash interpreter
↳ shebang

2) which bash → when writing shebang #! , we need the correct path to the interpreter.

so instead of guessing, you run

which bash → It shows the full path of the bash executable being used.

steps to write a shell script.

vim shellscript.sh

#!/bin/bash

echo "Hello all"

then press esc & :wq to exit and save changes.

But this script is not executable

if we type

3) ls -l → detailed list of files with permissions, ownership, size and timestamp.

To make file executable

4) chmod → change mode
→ used to change permissions of a file or directory in linux

(owner) user group others

-rw-rw-r--

if this is in starting it means it is a file

r → read
w → write
x = execute

chmod u+x shellscript.sh → add execute permission for the owner.

chmod g-w " → remove write permission for the group.

chmod a+r " → give read permission to everyone.

So to make file executable, type

chmod u+x shellscript.sh or

chmod 754 shellscript.sh

when we made .sh file executable, it will change its color

And now to run that executable script

type ./ shellscript.sh

↳ looks in this directory.

We have to go to that directory where our script file is & then ./ filname

How to write comments in a script

→ Use # to write single line comment.

and to write multi line comment

<< comment
Anything
written
here will not execute
comment

} this is multiline comment

Best practices to write script:-

Add short description at the top of script

```
#!/bin/bash
# Script Name : script.sh
# Author : Your Name
# Date : Sept 2025
# Purpose : Simple script to greet user.
```

Creating variables :-

```
name = "Alok"
age = 25
```

name = " Alok " → X
↳ we don't have to give space while writing variables.

To print this

```
echo " my name is $name & age is $age"
```

to print date
use $(date)

Taking input from user :-

```
echo " Enter your city :"
read city
echo " You live in $city"
```

Proper Script showing variables:

```
#!/bin/bash
# script name: variable.sh
# Author name: Alok
# Date : Sep 2025
# Purpose : script showing variables

name = "Alok"                    # String variable
age = 25                         # Number variable
today = $(date)                  # store command output
HOSTNAME = $(hostname)           # store system hostname

echo " Hello, $name!"
echo " You are $age years old."
echo " Today is $today."
echo " This system's hostname is $HOSTNAME."
```

Arguments :-

Accessing arguments :

$0 → the script name itself.
$1 → first argument
$2 → second argument ---- & so on
$# → no. of arguments passed.
$@ → all arguments as separate words
$* → all arguments as a single string.

Constants (read-only variables)

```
pi = 3.14
readonly pi
echo "Value of pi is $pi"
```

using all arguments

```
echo " All arguments using \$@ : $@"
echo " All arguments using \$* : $*"
```

Lets say arguments are

```
./script.sh Alok Devops 2025
```

Output : All arguments using $@ : Alok Devops 2025
" " " $* : " " "

$@ → treats each arguments as separate, good for loops
$* → treats all arguments as a single string.

Conditional Statement:

```
#!/
read -p " Enter a movie : " movie
read -p " Enter another movie : " another_movie
if [[ $movie == " Avengers " ]]
then
    echo " Marvel Universe "
elif [[ $another_movie == " Superman " ]]
then
    echo " DC universe "
else
    echo " Another universe "
fi
```

↳ for closing.

NOTE

If I want to give multiple values inside a condition then

Ist method

```
if [[ $movie == " Avengers " || $movie == " Ironman "
    || $movie == " Thor " ]]
```

2nd method

```
if [[ " Avengers Ironman Thor " =~ $name ]]
```

|| → OR

=~ → regex (pattern matching)

Loops : → Later

```
for (( num=1 ; num<=5 ; num++ ))
do
        mkdir "demo $num"
done
```

---

man command

man mkdir → this will give every details about mkdir command.
We can use any command with man command_name

history → it is a command which shows all the commands we have used in currently.

Diff. b/w touch & vim command
touch → only creates files.
vim file_name → this will create a file as well as opens it.
this command is used for automation.

pwd (shows present working directory)

nproc → command used to display no. of processing units available to current process or system.

free → used to check the amount of free and used memory on your system.

top → a dynamic real-time utility that provides a detailed overview of system performance.

df → shows the amount of free space that is left on file system.

df → for disk space utilization

free → for displaying system memory utilization including both physical RAM and swap memory.

swap memory → area on your hard disk (or SSD) used as virtual memory to extend systems Physical ram.

when your RAM fills up, the OS moves less used data from RAM to slower swap space, freeing up RAM for active processes.

RAM → temporary, short term memory in computer where CPU stores data and program instructions that it needs for immediate use.

#!/bin/bash

↓ shebang

↳ path to bash shell

→ it is needed because if we do not provide this path, your script might run with the default shell of the system like sh, dash, zsh.

→ this can cause errors if your script uses Bash-specific features.

#!/bin/bash ensures that your script always runs with **Bash**.

set -x → enables debug/trace mode.

when it's enable every command in script is printed to the terminal before it is executed.

ps -ef :- used to see all the processes currently running on your system.

ps → process status
-e → shows processes for all users.
-f → show processes in full format

It shows:

UID → user who started process
PID → Process ID
PPID → Parent Process ID
C → CPU usage
STIME → start time of the process
TTY → Terminal associated with the process (? if none)
TIME → Total CPU time used
CMD → Command used to start the process.

ps-ef

→ snapshot of all processes at the moment you run the command.
→ output doesn't update automatically.
→ shows process detail in a list format.
→ often used with grep to search for a specific process.

top

→ real time view of processes.
→ updates every few seconds until you quit
→ we can press keys to manage processes.
→ monitors systems performance live.

ps -ef | grep nginx
↳ it will filter & display only the processes that have nginx in their command line or name.

ps -ef | grep amazon

↳ this pipe sends the output of ps -ef as input to next command.

↳ first command

Imp qu:

date | echo "Today is"

output: ~~Today~~ Today is

this is because

First lets understand stdin, stdout, stderr.

stdin → where a program get input.
stdout → where program ~~gets~~ sends output.
stderr → where program sends error messages.

→ stdin takes input from keyboard

→ stdout is think it like the normal output of a program.

eg: echo "Hello"
Hello is printed on screen → that is stdout

echo "Hello" > output.txt
This command will redirects the output ~~out~~ of echo into output.txt file.

→ stderr: think of it as a error message a program shows.

eg. ls nofile we will get an error, that message comes through stderr.

stdin is the default channel through which program receives input.

cat < myfile.txt

in this it will print the content of myfile.txt.
means stdin is taken from myfile.txt.

So when we write

date | echo "Today is"

date runs and produces output like 17 Sept 2025
The pipe sends the output to the input of next command.
But echo doesn't read input from the pipe —
it only prints what we give it to echo.

because

date prints the output to stdout.
the | (pipe) takes that stdout and send it as in stdin to next command.
but here echo doesn't read from stdin at all.
its job is to just take whatever arguments you passed, prints them to std out.

echo never read stdin because it was never designed for that purpose.

But if echo "Hello" | cat

output: Cat.

There are some commands that takes stdin as their input

cat → prints whatever it gets from stdin
grep → searches text from stdin.
sort → sorts text from stdin.
wc → counts lines | words | chars from stdin.

date also doesn't read from stdin.

powerful commands of Linux | Bash :-

1) **grep** (search text)

Find text patterns in files or input.

e.g. grep "error" logfile.txt

If we want to search for multiple patterns (text)

- grep -e "pattern1" -e "pattern2" filename.txt

grep -e "error" -e "fail" logfile.txt

This will find lines containing error or fail.

OR

grep "pattern1\|pattern" filename.txt

↳ this works as or operator.

For case insensitive :-

grep -i "error\|fail" logfile.txt

Invert match (show lines not matching)

grep -v -e "error" -e "fail" logfile.txt

2) **awk** :

→ awk is a text processing tool.
→ it works on files line by line.

By default it splits each line into fields (columns), separated by spaces (or other delimiter)

$0 → whole line
$1 → first column
$2 → second column ---& so on.

eg:- awk '{print $0}' file.txt → this will print entire file

awk '{print $1}' file.txt → print only first column.

awk '{print $1, $3}' file.txt → print first & third column.

- ps -ef | awk '{print $1, $2}'
  ↳ prints only user & process ID of running processes.

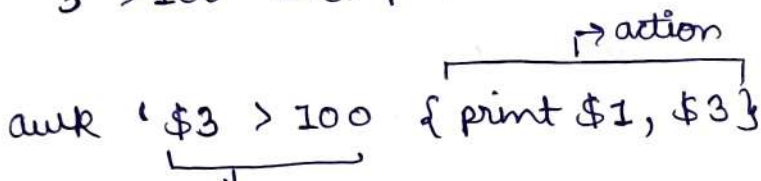
awk is useful because:
→ extracts specific columns.
→ filters lines based on conditions
→ do calculation.
→ great for log files, csv & system output.

awk → a tool to read text line by line, split into columns, & let you print or process specific parts.

awk with condition :-

```
awk '$3 > 100 {print $1, $3}' data.txt
```

→ Basically this command filters lines where column 3 > 100 and prints column 1 and 3.



3) sed: stream editor

→ It reads text (from a file or stdin), applies changes (substitute, delete, insert etc) and outputs the result.

i) substitute text (replace)

```
sed 's/old/new/' file.txt
```
↓ substitute

→ replaces first occurence of old with new in each line.

→ output goes to screen (doesn't change file)

For replacing all occurences

```
sed 's/old/new/g' file.txt
```
↳ global (replaces all occurences in each line)

ii) edit file in place :

→ sed -i 's/world/Alok/g' file.txt

-i → edit the file in place (directly modifies file.txt)

's/world/Alok/g' → substitution command

file.txt → the file being modified.

Quick tip for safety:

sed -i.bak 's/world/Linux/g' file.txt

This will create a backup file file.txt.bak before editing. and modifies permanently in original file.

iii) Delete lines :-

- sed '2d' file.txt → delete line 2.

this does not change the file unless you use -i.

sed -i '2d' file.txt → this will actually remove line 2 from file.txt

sed '2,4d" file.txt → deletes lines from 2 to line no. 4.

sed '$d' file.txt → delete the last line

iv) Insert text:

sed '3i This is a new line' file.txt

↳ inserts before line 3.

sed '3a This is after line 3' file.txt.

↳ append after line 3.

line1: Hello
line2: world
line3: linux
Line4: scripting

insert

line1: Hello
line2: world
This is a new line
line3: linux
line4: scripting.

append

line1: Hello
line2: world
line3: linux
This is after line 3.
line4: scripting.

To append multiple lines :- suppose file.txt contains

Line 1
Line 2
Line 3
Line 4

sed '3a This is line A\
This is line B\
This is line C' file.txt

Output

Line 1
Line 2
Line 3
This is line A
" " " B
" " " C
Line4

- sed '2d ; 3i Insert before line 3 ; 4a append after line 4' file.txt

→ deletes line 2
→ inserts text before line 3
→ appends text after line 4

-

sed -n '/error/p' logfile.txt
↓ ↳ prints the line (if pattern matches)
→ normally sed prints all lines by default.
→ but with -n, it only prints what you explicitly tell it to.

/error/ → this is a pattern
→ it matches any line containing the word error.

- set -e → exits the script immediately if any command fails.
→ earlier it, a script may continue running even if an earlier command fails — which can cause big issues in automation.

In devops scripts, you often combine:

set -euo pipefail → -e → exit on error
-u → treat unset variables as errors
-o pipefail → if any command in a pipeline fails, the whole pipeline fails.

curl :- stands for client URL
- → used to transfer data to/ from a server using URLs.
- → supports HTTP, HTTPS, FTP & more.

→ a command that fetches data from internet.

- curl -o filename https://-----.com → saves response into a file instead of printing to terminal.
- curl -O https://---- → saves with original filename.
- curl -I https://--- → show headers only.

for APIs:

1) By default curl uses GET

2) **POST request (send data):**

```
curl -X POST https://api.example.com/users \
     -H "Content-Type: application/json" \
     -d '{"name": "Alok", "age": 23}'
```

Here
- -X POST → specify request type
- -H → set header (here will tell the server, we're sending JSON)
- -d → send data.

3) **PUT request (update data) :-**

```
curl -X PUT
```

4) **Delete (remove data) :-**

```
curl -X DELETE
```

- **wget** → stands for Web GET
  → it's a command line utility to download files from the web (HTTP, HTTPS, FTP)

Eg. wget https://example.com/file.zip
this will saves the file in current directory.

wget -O filename ---
↳ save with the custom file name that we give.

wget -c → continue download if it was interrupted.

wget -r https://google.com
↳ recursive download.
this can mirror websites locally.

wget -b → runs in background & logs progress in wget-log

- **find** → searches for files and directories in a directory hierarchy.
  → it can filter by name, type, size, date, permission, owner etc.

i) **find by name:-**
find /home/alok -name "script.sh"
-iname → case-insesitive

ii) **find by type:-**
find /var/log -type d
-type f → files only
-type d → directories only.

iii) finds by size :-

find / -size +100m

→ files larger than 100MB

+ → greater than, - less than

& many more.

sudo su → becoming super user

su → switch user.

Foor loop

```
for i in {1..100}
do
    echo $i
done
```

} this will print no. from 1 to 100

we can also specify steps :

```
for i in {1..10..2}
do
    echo $i
done
```

→ 1, 3, 5, 7, 9

C style for loop :-

```
for ((i=1; i<=5; i++))
do
    echo $i
done
```

Loop through files in a directory :-

```
for file in *.sh
do
    echo "script found: $file"
done.
```

Loop through script arguments :-

```
for arg in $@
do
    echo $arg
done
```

} while running script

```
./script.sh Alok Devops 2025
```

these are arguments.

Small script

```
#!/bin/bash
# Loop through log files and delete if older than
    7 days.

for logfile in /var/log/*.log
do
    echo "Checking $logfile ...."
    if [ $(find "$logfile" -mtime +7 2>/dev/null) ]
    then
        echo "Deleting old log: $logfile"
        rm -f "$logfile"
    fi
done
```

-mtime +7 → modified more than 7 days ago

2>/dev/null → hides error messages (in case file doesn't exist)

$(---) → command substitution.

How to open a file in read mode:

vim → test.sh

↓

read.

trap → lets you catch signals or errors and run custom commands when they happen.

why to use:-

→ to cleanup temporary files before the script exits.
→ to stop background processes if the script is killed.
* → to handle Ctrl+C (SIGINT) gracefully.
→ to debug error in scripts

```
trap 'commands' SIGNALS
```

'commands' ↓ what to run when signal is received

SIGNALS ↓ which signal to catch (SIGINT, SIGTERM, EXIT)

Write a shell script to print numbers that are divisible by 3 & 5 and not 15.

range of no.- 1 to 100

```
for i in {1..100}; do
if ['expr $i % 3' == 0] || ['expr $i % 5' == 0]
    && ['expr $i % 15' != 0];
then
    echo $i
fi;
done
```

Script to print no. of s in mississippi

```
x = "mississippi"
grep -o "s" <<<"$x" | wc -l
```

-o ↓ only

crontab :

cron : time-based job scheduler in Linux/Unix

crontab : cron table, where you define jobs to run at scheduled times.

It allows you to automate tasks (backups, log cleanup, monitoring, updates etc)

crontab -l → view crontab
crontab -e → edit your crontab
crontab -r → remove all jobs.

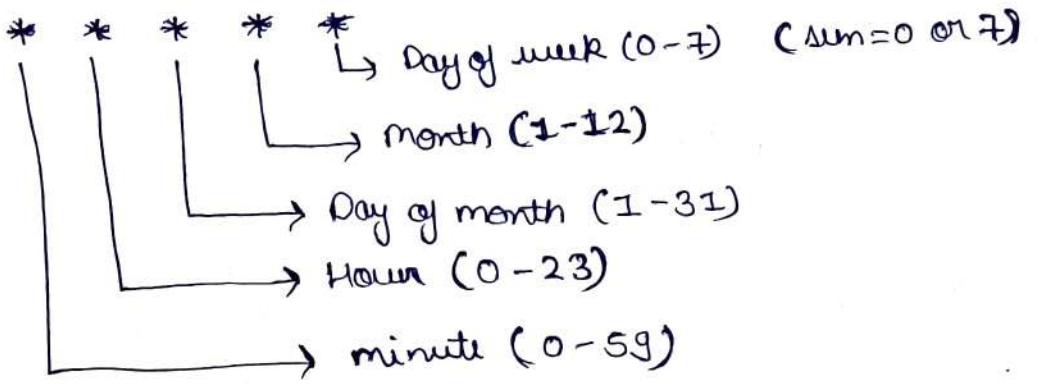
→ crontab is your linux scheduler — it lets you run command / scripts automatically at fixed times & dates or interval.

why Devops engineer use crontab :-

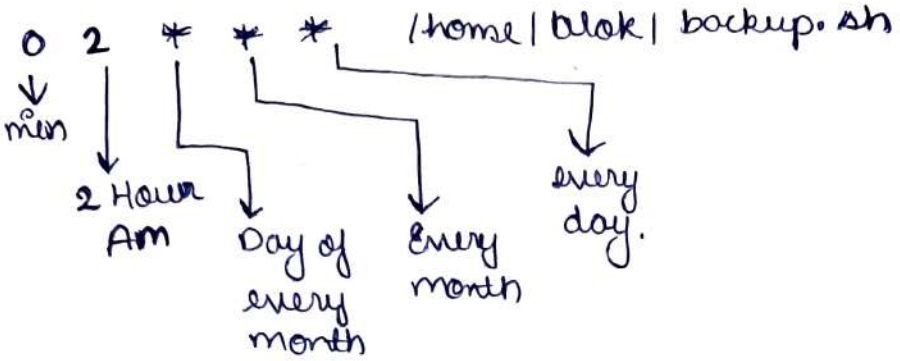
→ automate log cleanup
→ schedule backups.
→ run monitoring / health check scripts
→ trigger deployment jobs.

How to write cron jobs :-

type crontab -e → this open crontab editor for user.



Run a script every day at 2AM



Run /home/alok/backup.sh ~~at~~ every day at 2:AM

Run every monday at 5 PM

0 17 * * 1 /home/alok/report.sh

Run every 15 min:

*/15 * * * * /home/alok/check_status.sh

0 2 * * * /home/alok/backup.sh >> /home/alok/backup.log 2>&1

runs the script everday at 2AM

the script being executed.

>> /home/alok/backup.log → append stdout (output) of script to /home/alok/backup.log.

2>&1 → redirects stderr (error messages) to the same place as stdout.

So both normal output & error go into backup.log

>> → append

> → overwrite fill each time.

Instead of strings * * * * * you can use:

@reboot → run once at startup
@daily → run everyday at midnight.
@hourly → run every hour
@weekly → run once a week
@monthly → run once a month.

eg.
```
#!/bin/bash
x=5
x="Alok"
echo "$x"
```
: The output will be Alok because firstly number 5 is assigned to variable x. then string Alok is reassigned to x
∴ the value of x is overwritten.

Is bash scripting dynamic or static?

Bash is dynamic.

why?
variables in Bash have no fixed type.

```
x=5
x="Alok"
```
} Both are valid, same variable can hold a number, then later a string.

we don't have to declare types (int, string etc) like in static languages (C, Java)

int x=5; in languages like C or Java

logrotate :-

- linux utility that helps manage log files so they don't grow endlessly and fill up your disk.

→ Rotate logs → renames the old log file and starts a fresh one.

→ compresses old logs to save space.

→ removes older logs ~~than~~ after a certain no. of days.

→ can run daily / weekly / monthly.

→ works automatically via a cron job.

e.g:- Lets say you have a growing log file

/var/log/myapp.log

If you configure logrotate, it can:

→ rename it to myapp.log.1 & create a new empty myapp.log

→ next rotation: move myapp.log.1 → myapp.log.2 & so on

→ compress older ones → myapp.log.2.gz

→ Delete logs older than, say 30 days.