A Comprehensive Guide to Bash Scripting for DevOps Engineers

- **Bash (Bourne Again Shell**) is one of the most essential scripting languages for DevOps engineers. It is used for automating system administration tasks, managing cloud infrastructure, creating CI/CD pipelines, and handling configuration management.

1. What is Bash Scripting?

- Bash scripting is a way of automating command-line tasks in Unix-based operating systems like Linux and macOS. A Bash script is simply a text file containing a series of commands that are executed sequentially by the shell.

Key Features of Bash Scripting

- Automates repetitive tasks like backups, updates, and deployments.
- Executes multiple commands in a sequence, reducing manual effort.
- Handles file operations (create, delete, move, modify files).
- Can be used in CI/CD pipelines and cloud automation.

2. Why Should DevOps Engineers Learn Bash Scripting?

- Bash scripting is a fundamental skill for DevOps engineers for the following reasons:

Server Automation & Configuration Management

- Automating software installation and configuration on Linux servers.

Example: Updating system packages using Bash scripts.

CI/CD Pipeline Integration

- Writing Bash scripts to automate code building, testing, and deployment in Jenkins, GitHub Actions, and GitLab CI/CD. Example: A script that triggers deployment when code is pushed to GitHub.

Cloud Infrastructure Automation

- Automating AWS, Azure, and GCP services using CLI tools inside Bash scripts.

  Example: A script that provisions AWS EC2 instances using AWS CLI.

Log Monitoring & System Alerts

- Writing Bash scripts to monitor system logs and send alerts when issues arise.

  Example: A script that scans logs for error messages and emails the DevOps team.

Containerization & Kubernetes Management

- Automating Docker container builds and Kubernetes deployments using Bash.

  Example: A script that restarts a Kubernetes pod if it crashes.

3. Bash Scripting Basics

- Before diving into advanced scripting, let's cover the fundamentals.

1. Creating and Running a Bash Script

To create a Bash script, follow these steps:

Step 1: Create a Script File

nano myscript.sh

or

vim myscript.sh

Step 2: Add the Shebang (Interpreter Directive)

```
#!/bin/bash

echo "Hello, DevOps Engineer!"
```

Step 3: Save and Exit

Press Ctrl + X, then Y, and hit Enter.

Step 4: Make the Script Executable

chmod +x myscript.sh

Step 5: Run the Script

./myscript.sh

Output:

Hello, DevOps Engineer!

4. Bash Scripting Fundamentals

1. Variables in Bash

```
#!/bin/bash
name="DevOps Engineer"
echo "Welcome, $name!"
```

Output:

Welcome, DevOps Engineer!

2. User Input Handling

```
#!/bin/bash
echo "Enter your name:"
read user_name
echo "Hello, $user_name!"
```

3. Conditional Statements (if-else)

```
#!/bin/bash
echo "Enter a number:"
read num
if [ $num -gt 10 ]; then
```

```
    echo "The number is greater than 10."
else
    echo "The number is 10 or less."
fi
```

4. Loops in Bash

For Loop

```
#!/bin/bash
for i in {1..5}; do
    echo "Iteration $i"
done
```

While Loop

```
#!/bin/bash
count=1
while [ $count -le 5 ]; do
    echo "Count: $count"
    ((count++))
done
```

5. Advanced Bash Scripting for DevOps

1. Automating System Updates

```
#!/bin/bash
echo "Updating system packages..."
sudo apt update && sudo apt upgrade -y
echo "System update complete!"
```

2. Backup and Restore Files

```
#!/bin/bash
backup_dir="/backup"
mkdir -p $backup_dir
cp -r /var/log/* $backup_dir
echo "Logs have been backed up to $backup_dir"
```

3. Monitoring Disk Usage and Sending Alerts

```
#!/bin/bash
threshold=80
usage=$(df -h / | grep '/' | awk '{print $5}' | sed 's/%//g')

if [ $usage -gt $threshold ]; then
   echo "Warning: Disk usage exceeded $threshold% ($usage%)"
   # Send an email alert (requires mailutils installed)
   echo "Disk usage warning: $usage%" | mail -s "Disk Alert" admin@example.com
fi
```

4. Automating Docker Container Deployment

```
#!/bin/bash
echo "Pulling the latest Docker image..."
docker pull nginx:latest
echo "Starting a new container..."
docker run -d -p 80:80 --name mynginx nginx:latest
```

5. Kubernetes Pod Health Check

```
#!/bin/bash
pod_status=$(kubectl get pods my-app -o jsonpath='{.status.phase}')
if [ "$pod_status" != "Running" ]; then
   echo "Pod my-app is not running. Restarting..."
   kubectl delete pod my-app
fi
```

6. Best Practices for Bash Scripting in DevOps

- **Use Shebang (#!/bin/bash)** – Always define the shell interpreter.
- **Use Comments (#)** – Make scripts understandable.
- **Use Meaningful Variable Names** – Improve readability.
- **Test Before Deployment** – Run in a safe environment first.
- **Log Outputs (>> logfile.log)** – Keep a record of script execution.
- **Error Handling (set -e)** – Stop script on errors.
- **Security Measures** – Avoid storing sensitive credentials in scripts; use environment variables instead.

7. Conclusion

Bash scripting is an essential skill for any DevOps engineer. It enables automation of system administration, CI/CD pipelines, cloud management, and monitoring tasks.

Learning Bash scripting will significantly enhance your efficiency in handling Linux-based DevOps workflows.

Scripts are used to automate the daily tasks, simplify repetitive tasks, and perform system administration tasks.

Types of scripts:

1. Bash
2. Zsh - Z Shell
3. FISH - Friendly Interactive Shell
4. Ksh - Korn Shell
5. Csh - C Shell
6. Tcsh - TENEX C SHell
7. PowerShell

Bash scripts can be used for various purposes,

such as executing a shell command, running multiple commands together, customizing administrative tasks, performing task automation etc.

So knowledge of bash programming basics is important for every Linux user.

cat $SHELL - displays the current shell type you are working on.

cat /etc/shells - displays the available shells of that machine.

which bash

echo $SHELL

ps -p $$

cat /etc/passwd | grep ec2-user

To switch from one shell to an other shell : sh

==========

==========

#! /bin/bash (#! = shebang)

echo "Hello DevOps" > file.txt

The first line of a shell script is always a shebang usually the bash #!/bin/bash which specifies the interpreter to execute the script. When the script is executed the kernel reads the shebang line and uses that interpreter to execute that script.

```
================================================================================
================================================================================
#!/bin/bash
mydata="Hello, world!"
echo $mydata
```

```
================================================================================
================================================================================
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
================================================================================
================================================================================
```

```
echo "Enter Username: "
read username
echo $username

# displays the prompt message
# -p stand for prompt
# reads input from the user and puts it in the newusername variable
read -p "Enter the new username: " newusername
echo $newusername
```

```
# reads input from the user & hides the text from echoing in the terminal
# -s stands for silent
read -sp "Enter Password: " password
echo ""
echo $password
```

if you don't wish to specify the variable name for the read we can use $REPLY to echo the value

```
#!/bin/bash

echo "Enter the username: "
read
echo "Read without variable name assignment: "$REPLY
```

==========

Argument Passing

We can pass arguments that can be used inside the scripts when it is executed.

Those arguments can be accessed by the script using special variables like $1 $2 $3 etc.

```
# gives the filename of the script itself
echo "FileName Argument: "$0 # argument_passing.sh

# gives the first argument passed
echo "First Argument: "$1 # Vish

# gives the second argument passed
```

```
echo "Second Argument: "$2 # DevOps Engineer


# displays all arguments passed

echo "All Arguments: "$@ # Vish DevOps Engineer


# displays number of arguments passed

echo "No of Arguments: "$# # 2

echo "Third Argument: "$3 # 3

echo "Fourth Argument: "$4 # 4
```

==========

==========

Arithmetic Operations

In shell scripting, to perform arithmetic operations we need to use double parenthesis (( )) which is used for arithmetic expansion on integers.

The double parenthesis (( )) is also called a unary operator.

```
#!/bin/bash


n1=10

n2=5


echo "Sum of two numbers: "$(($n1+$n2)) # Addition

echo "Sub of two numbers: "$(($n1-$n2)) # Substraction

echo "Mul of two numbers: "$(($n1*$n2)) # Mulitplication

echo "Div of two numbers: "$(($n1/$n2)) # Division

echo "Modulus of two numbers: "$(($n1%$n2)) # Modulus
```

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

Conditions:

[[-z STRING]] - Empty string

```
#!/bin/bash

my_string=""

if [[ -z $my_string ]]; then
echo "The string is empty."
else
  echo "The string is not empty."
fi
```

[[-n STRING]] - Not empty string

```
#!/bin/bash

my_string="Hello, World!"

if [[ -n $my_string ]]; then
  echo "The string is not empty."
else
  echo "The string is empty."
```

```
fi

[[ STRING == STRING ]] - Equal
#!/bin/bash

string1="apple"
string2="apple"

if [[ $string1 == $string2 ]]; then
  echo "Strings are equal."
else
  echo "Strings are not equal."
fi

[[ STRING != STRING ]] - Not equal
#!/bin/bash

#!/bin/bash

string1="apple"
string2="orange"

if [[ $string1 != $string2 ]]; then
echo "Strings are not equal."
else
  echo "Strings are equal."
fi
```

[[NUM -eq NUM]] - Equal

```
#!/bin/bash

num1=5
num2=5

if [[ $num1 -eq $num2 ]]; then
echo "Numbers are equal."
else
  echo "Numbers are not equal."
fi
```

[[NUM -ne NUM]] - Not equal

```
#!/bin/bash

num1=5
num2=10

if [[ $num1 -ne $num2 ]]; then
echo "Numbers are not equal."
else
  echo "Numbers are equal."
fi
```

[[NUM -lt NUM]] - Less than

```
#!/bin/bash

num1=5
num2=10

if [[ $num1 -lt $num2 ]]; then
  echo "Number 1 is less than Number 2."
else
  echo "Number 1 is not less than Number 2."
fi
```

[[NUM -le NUM]] - Less than or equal

```
#!/bin/bash

num1=5
num2=5

if [[ $num1 -le $num2 ]]; then
  echo "Number 1 is less than or equal to Number 2."
else
  echo "Number 1 is greater than Number 2."
fi
```

[[NUM -gt NUM]] - Greater than

```
#!/bin/bash

num1=10
num2=5

if [[ $num1 -gt $num2 ]]; then
  echo "Number 1 is greater than Number 2."
else
  echo "Number 1 is not greater than Number 2."
fi
```

[[NUM -ge NUM]] - Greater than or equal

```
#!/bin/bash

num1=5
num2=5

if [[ $num1 -ge $num2 ]]; then
  echo "Number 1 is greater than or equal to Number 2."
else
  echo "Number 1 is less than Number 2."
fi
```

[[X && Y]] - And

```
#!/bin/bash

!/bin/bash

echo "Enter username"
read username
echo "Enter password"
read password

if [[ ( $username == "raham" && $password == "1234" ) ]]; then
echo "valid user"
else
echo "invalid user"
fi
```

[[X || Y]] - Or

```
!/bin/bash

echo "Enter username"
read username
echo "Enter password"
read password

if [[ ( $username == "raham" || $password == "1234" ) ]]; then
```

```
echo "valid user"
else
echo "invalid user"
fi
```

==========

[[-e FILE]] - Exists

```
#!/bin/bash

file_path="/path/to/file.txt"

if [[ -e $file_path ]]; then
  echo "File exists."
else
  echo "File does not exist."
fi
```

[[-r FILE]] - Readable

```
#!/bin/bash

file_path="/path/to/file.txt"

if [[ -r $file_path ]]; then
echo  "File  is  readable."
else
  echo "File is not readable."
fi
```

[[-d FILE]] - Directory

```
#!/bin/bash

directory_path="/path/to/directory"

if [[ -d $directory_path ]]; then
echo "Path is a directory."
else
  echo "Path is not a directory."
fi
```

[[-w FILE]] - Writable file

```
#!/bin/bash

file_path="/path/to/file.txt"

if [[ -w $file_path ]]; then
echo "File is writable."
else
  echo "File is not writable."
fi
```

[[-s FILE]] - File size is > 0 bytes

```
#!/bin/bash

file_path="/path/to/file.txt"

if [[ -s $file_path ]]; then
  echo "File size is greater than 0 bytes."
else
  echo "File size is 0 bytes or the file does not exist."
fi
```

[[-f FILE]] - File

```
#!/bin/bash

file_path="/path/to/file.txt"

if [[ -f $file_path ]]; then
echo "Path is a regular file."
else
  echo "Path is not a regular file."
fi
```

[[-x FILE]] - Executable file

```
#!/bin/bash

file_path="/path/to/executable"

if [[ -x $file_path ]]; then
echo "File is executable."
else
  echo "File is not executable."
fi
```

==========

Loops:

if else loop is a conditional statement that allows executing different commands based on the condition true/false.

Here square brackets [[ ]] are used to evaluate a condition.

```
#!/bin/bash

# -e stands for exists
if [[ -e ./ifelse.sh ]]
then
  echo "File exists"
else
  echo "File does not exist"
fi
```

elif

elif is a combination of both else and if. It is used to create multiple conditional statements and it must be always used in conjunction with if else statement

```
#!/bin/bash

echo "Enter your lucky number"
read n

if [ $n -eq 101 ];
then
echo "You got 1st prize"
elif [ $n -eq 510 ];
then
echo "You got 2nd prize"
elif [ $n -eq 999 ];
then
echo "You got 3rd prize"

else
echo "Sorry, try for the next time"
```

==========

Srujana – Cloud & Dev-Ops Explorer

for:

The for loop is used to iterate over a sequence of values and below is the syntax

```
#!/bin/bash

for i in {1..10}
do
  echo "Val: $i"
done
```

==========

while

The while loop is used to execute a set of commands repeatedly as long as a certain condition is true.

The loop continues until the condition is false.

```
#!/bin/bash

count=0

while [ $count -lt 5 ]
do
  echo $count
count=$(($count+1))
done
```

==========

until

The until loop in shell scripting is used to execute a block of code repeatedly until a certain condition is met.

```
#!/bin/bash

count=1

until [ $count -gt 5 ]
do
   echo $count
   count=$(($count+1))
done
```

==========

Arrays

An array is a variable that can hold multiple values under a single name

${arrayVarName[@]} - displays all the values of the array.

${#arrayVarName[@]} - displays the lenght of the array.

${arrayVarName[0]} - displays the first element of the array

${arrayVarName[-1]} - displays the last element of the array

unset arrayVarName[2] - deletes the 2 element

```
#!/bin/bash

# Declare an array of fruits
fruits=("apple" "banana" "orange" "guava")

# Print the entire array
echo "All fruits using @ symbol: ${fruits[@]}"
echo "All fruits using * symbol: ${fruits[*]}"

# Print the third element of the array
echo "Third fruit: ${fruits[2]}"

# Print the length of the array
echo "Number of fruits: ${#fruits[@]}"
```

==========

Break Statement

break is a keyword. It is a control statement that is used to exit out of a loop (for, while, or until) when a certain condition is met.

It means that the control of the program is transferred outside the loop and resumes with the next set of lines in the script.

```
#!/bin/bash

count=1

while true
do
  echo "Count is $count"
```

```
 count=$(($count+1))
 if [ $count -gt 5 ]; then
  echo "Break statement reached"
  break
 fi
done
```

==========

Continue statement

continue is a keyword that is used inside loops (such as for, while, and until) to skip the current iteration of the loop and move on to the next iteration.

It means that when the continue keyword is encountered while executing a loop the next set of lines in that loop will not be executed and moves to the next iteration.

```
#!/bin/bash

for i in {1..10}
do
 if [ $i -eq 5 ]
 then
 continue
 fi
 echo $i
done
```

==========

Functions

Functions are a block of code which can be used again and again for doing a specific task thus providing code reusability.

Normal Function:

```
#!/bin/bash


sum(){
    echo "The numbers are: $n1 $n2"
    sum_val=$(($n1+$n2))
    echo "Sum: $sum_val"
}

n1=$1
n2=$2
sum
```

Function with return values

To access the return value of the function we need to use $? to access that value

```
#!/bin/bash


sum(){
    echo "The numbers are: $n1 $n2"
    sum_val=$(($n1+$n2))
    echo "Sum: $sum_val"
    return $sum_val
}

n1=$1
```

```
n2=$2

sum


echo "Retuned value from function is $?"
```

==========

==========

Variables in Functions

The variable is a placeholder for saving a value which can be later accessed using that name. There are two types of variables

Global - Variable defined outside a function which can be accessed throughout the script

Local - Variable defined inside a function and can be accessed only within it

```
#!/bin/bash


# x & y are global variables

x=10

y=20


sum(){

    sum=$(($x+$y))

    echo "Global Variable Addition: $sum"

}


sum
```

```
sub(){
        # a & b are local variables
        local a=20
        local b=10
        local sub=$(($a-$b))
        echo "Local Variable Substraction: $sub"
}

sub
```

==========

```
if [ $(whoami) = 'root' ]; then
                echo "You are root"
        else
                        echo "You are not root"
fi
```

==========

```
#!/bin/bash
valid=true
count=1
while [ $valid ]
do
echo $count
if [ $count -eq 5 ];
```

```
then
break
fi
((count++))
done
```

```
#!/bin/bash
echo "Enter Your Name"
read name
echo "Welcome $name"
```

```
#!/bin/bash
n=10
if [ $n -lt 10 ];
then
echo "It is a one digit number"
else
echo "It is a two digit number"
fi
```

```
#!/bin/bash

echo "Enter your lucky number"
read n

if [ $n -eq 101 ];
```

```
then
echo "You got 1st prize"
elif [ $n -eq 510 ];
then
echo "You got 2nd prize"
elif [ $n -eq 999 ];
then
echo "You got 3rd prize"

else
echo "Sorry, try for the next time"
fi
```

==========

```
Name_of_dir=$1
start_no=$2
end_no=$3

eval mkdir $Name_of_dir{$start_no..$end_no}
```

Command : ./one.sh mustafa 1 10

This command will create 1 to 10 folders

==========

WRITE A SCRIPT TO KNOW WHICH USER WE CURRENTLY LOGGED INT

```
if [ $(whoami) = 'root' ]; then
        echo "You are root"
    else
        echo "You are not root"
fi
```

==========

SCRIPT TO CREATE 10 FOLDERS:

```
Name_of_dir=$1
start_no=$2
end_no=$3

eval mkdir $Name_of_dir{$start_no..$end_no}
```

Command : ./one.sh mustafa 1 10

This command will create 1 to 10 folders

==========

SCRIPT TO TAKE BACKUP LOGS:

```
#!/bin/bash
src=/var/log/httpd/access_log
dest=mybackup
time=$(date +"%Y-%m-%d-%H-%M-%S")
backupfile=$dest/$time.tgz
```

```
#Taking Backup
echo "Taking backup on $time"
tar zcvf $backupfile --absolute-names $src

if [ ${?} -eq 0 ]
then
    echo "Backup Complete"
else
    exit 1
fi
```

The z option compresses the backup using gzip,

the c option creates a new archive,

the v option enables verbose mode to display the progress,

and the f option specifies the output file.

if [${?} -eq 0]: This line checks the exit status of the tar command using the special variable $?. If the exit status is 0, it means the backup was successful.

${?} is a special variable that represents that most recently executed command.

==========

CREATE A SCRIPT THAT USER EXIST OR NOT, IF NOT LET'S CREATE

```
echo "Enter username:"
read username

# Check if user exists
id $username >/dev/null 2>&1
if [ $? -eq 0 ]; then
  echo "User exists"
else
  echo "User does not exist"
  read -p "Do you want to create the user? (y/n) " create_user
  if [ $create_user == "y" ]; then
    read -s -p "Enter password: " password
    echo
    useradd -m -p $(openssl passwd -1 $password) $username
    echo "User created"
  else
  exit 0
  fi
fi
```

==========

```
#!/bin/bash
echo "Total arguments : $#"
echo "1st Argument = $1"
echo "2nd argument = $2"

./filename Redhat ubuntu fedora centos
```

==========

```
#!/bin/bash

string1="Linux"
string2="Hint"
echo "$string1$string2"
```

==========

```
#!/bin/bash
echo "Enter directory name"
read newdir
mkdir $newdir
```

==========

```
#!/bin/bash
echo "Enter filename"
read newfile
touch $newfile
```

==========

```
#!/bin/bash
echo "Enter directory name"
read ndir
```

```
if [ -d "$ndir" ]
then
echo "Directory exist"
else
`mkdir $ndir`
echo "Directory created"
fi
```

==========

```
#!/bin/bash
echo "Enter filename to remove"
read fn
rm -i $fn
```

==========

To check HTTPD IS RUNNIG OR NOT?

```
#!/bin/bash
check_service() {
   if systemctl status $1 | grep "active (running)"; then
      return 0
   else
      return 1
   fi
}

# Call the check_service function with argument "apache2"
if check_service httpd; then
   echo "HTTPD is running"
```

```
else
   echo "HTTPD is not running"
fi
```

==========

To check the disk usage of HOME directory:

```
#!/bin/bash

get_disk_usage() {
   directory=$1
   # Calculate disk usage of specified directory
   disk_usage=$(du -s $directory | awk '{print $1}')
   echo $disk_usage
}

# Call the function and store the result in a variable
usage=$(get_disk_usage $HOME)

# Display the result
echo "The disk usage of the home directory is: $usage bytes."
```

==========

To check the disk usage of /home directory:

```
#!/bin/bash

get_disk_usage() {
   directory=$1
   # Calculate disk usage of specified directory
   disk_usage=$(du -s $directory | awk '{print $1}')
   echo $disk_usage
}

# Call the function and store the result in a variable
usage=$(get_disk_usage /home)

# Display the result
echo "The disk usage is: $usage"
```

==========

TO CHECK TOOLS ARE INSTALLED OR NOT

```
#!/bin/bash
check_package() {
   local PACKAGE_NAME="$1"
   if ! command -v "${PACKAGE_NAME}" > /dev/null 2>&1
   then
        printf "${PACKAGE_NAME} is not installed.\n"
   else
```

```
        printf "${PACKAGE_NAME} is already installed.\n"
    fi
}
check_package "vim"
check_package "git"
```

==========

INSTALL PACKAGES USING FUNCTIONS:

```
#!/bin/bash
install_package() {
  local PACKAGE_NAME="$1"
  yum install "${PACKAGE_NAME}" -y
}
install_package "vim"
install_package "git"
```

==========

TO CHECK MULTIPLE SERVICES ARE RUNNING OR NOT

```
#!/bin/bash

check_services() {
    for service in "$@"; do
        if systemctl status "$service" | grep "active (running)"; then
```

```
        echo "$service is running"
    else
        echo "$service is not running"
    fi
  done
}
# adding service name apache2 mysql to check
check_services apache2 mysql
```

==========

SCRIPT TO GET OLDER FILES THAN 60 DAYS

```
#!/bin/bash

find path -mtime +60 -exec rm -f {} \;
```

1. CREATE A FOLDER (mkdir folder1)
2. CREATE SOME FILES IN SIDE THE FOLDER WITH OLDER TIME STAMP (touch -d "Fri, 21 Aug 2023 11:14:00" tcs infosys infotech)
3. WRITE A SCRIPT TO FIND FILES : (find folder1 -mtime +60 -exec ls -a {} \;)
4. MODIFY THE SCRIPT TO DELETE A FILES : (find myfiles -mtime +60 -exec rm -f {} \;)

==========

==========

SCRIPT TO GET MEM INFO:

```
echo "Total Memory: $(free -m | awk '/Mem/ {print $2 " MB"}')"
echo " Used Memory: $(free -m | awk '/Mem/ {print $3 " MB"}')"
echo "Free Memory: $(free -m | awk '/Mem/ {print $4 " MB"}')"
echo "Avail Memory: $(free -m | awk '/Mem/ {print $7 " MB"}')"
```

NOTE:

free -m gives full info about memory

Awk : this is used to divide the data in the format of rows and column's which are present in a file.

/Mem : matches the lines which contains memory info

$2 : prints the second column of the output of (free -m) command

$3 : prints the third column of the output of (free -m) command

==========

SCRIPT TO GET EBS INFO:

```
echo "EBS Volume Usage: $(df -h | grep -E '^/' | awk '{print $4 " free out of " $2}')"
```

NOTE:

-E '^/' : that ends with

awk : used to split the output on rows and columns

$4 : used mem

$2 : total mem