Bash+Linux

Bash

Bash (ScriptName).sh == ./Script ===>Execute Bash File File

Note "File Extension is not Necessarily & don't Use a name that is used for command"

#(sharp) + ! (Bang)==>shebang

#! /usr/bin/bash ===> Same As Doctype in html it tills the system that this is bash

#! /bin/bash

#! /usr/bin/env bash ====>looks into our environment into our Path variable, find out where our bash executable is located and execute it

For Example if we do :

Touch text.txt

Inside the file we write #! /usr/bin/env nano ⇒it will open text Editor

Conclusion ===>#! Is not only for bash it is a universal

Shell Expansion

⇒ Before Shell Execute The command Rewrite and Parses The Command

Filename Expansion:
EX: ls *
Echo * ===> Same output as ls *

Conclusion :
In this Case The * is being "expanded"===>content of the current directory And passes those items as parameters to the command

***.txt ===>1 or more Character That have .txt**

?.txt=====>1 character

Echo /home/Username/Documents/*.txt ====> Return all the files that gave those .txt extension

~ ==>home Folder

Ls ~ =====>listing Home folder

Ls ~ ⇒ls "${HOME}" "~=${HOME} "

Home=/

Ls ~ ====> shows The root Folder

Echo ~ ===>/

~+ ===> Current Working Directory "PWD"

—----------

Variable Expansion

Allow us to access variable with those Methods:
=>echo $HOME
=>echo ${HOME}
=>echo "$HOME"
=>echo "${HOME}"

No quotes ➔ Every Expansion are enabled
=> " " = String & Variable ||Most Expansion is disabled such as ~ , filename Expansion(*),word Splitting however some expansion are enabled such as variable Expansion , Parameter Expansion

' ' ➔ Only String || Disabled All expansion

$ ➔ we want to Open one of the few expansion

Best Practice =➔echo "${HOME}"
Why so we can add something after it for EX:
➔ echo "This is my home Directory ${HOME}"

Note: echo doesn't knows what is printing because bash assign the value Before echo is even invoked or executed

To Assign a Value to a variable
Hamada = 5
echo $"{Hamada}"
—----------

Shell Parameter Expansion

it works with existing Variables:
Quart The length of String : #

"${#HOME}"

To cut out a substring: :Start:Length

“${HOME:Start:Length}”
➔ “${HOME:40:10}”

To Change a pattern : // Patttern/Replacment
“${HOME//Patttern/Replacment}”
➔ “${HOME//home/users}”
—----------

Word Splitting

Bash perform a Word Splitting in our input:

It happens after our command has been rewritten
By our expansion

EX: Touch file.txt file2.txt
The command will be splited into 3 words
➔Touch
➔File.txt
➔File2.txt
➔etc

Word Splitting Will occur at any Character That is listed in the Variable IFS
IFS is the Spaces in short term it is every space ,every tab, every Newline
Note:{
Note :to check or view IFS use ➔echo “{IFS}”| hexdump
1. **IFS** (Internal Field Separator)
Determines how the shell splits input (like variables, strings, or command outputs) into separate fields.
2. **FS** (Field Separator)
Used in tools like `awk` to specify the delimiter for separating fields in the input file or stream.
3. **OFS** (Output Field Separator)
Determines how the shell splits input (like variables, strings, or command outputs) into separate fields.
}
“That is why” if we used this command Touch file.txt file2.txt

With those spaces it still works cuz IFS isn’t getting rewritten by the shell to get

Ignored by Word Spliting cuz it detect [A-Z-a-z-0-9-etc]

Touch “A File3.txt”
Touch ‘A File3.txt’

Will work cuz the “ ” disable the IFS between the quotes

Brace Expansion

To Expand String of characters (Brace) in Bash 4

Echo data.{csv,txt} ➔ data.csv data.txt

{start..end} generate strings

Echo {A..Z} print ABCD→Z

Touch file{1..10}.txt create files from 1 to 10v

—----------

Command Substitution

Execute a command and use the output as Replacement $(...)
EX: echo "$(cat file.txt)"
"" is used to disable Word Splitting
$enable variable Execution
backticks is also a command substitution
Ex
"`ls`"
Command Substitution==?"$(ls ~/Documents)"

Process Substitution

It allows us to use the input or the output of a process as temporary file
By using <(command)

EX: echo <(ls)

We can also combine it with a redirect

wc -l < <(ls) it is same as using pipe line ls | wc -l but more simple in bigger project

It allows us to use the input of a process as temporary file

>(Command)

—----------

BashScript

#===? Comments
MultiLine Comments

: << 'My_comment'
Dasdasdasd

Sdasdasd
Asdasdasd
My_comment

: ➔ Empty Command (Command That Executed But Returns nothing)
<< ➔Redirect

Conclusion: So multiline comment overhead our application Because it still executes

1) In Bash everything is a string
2) in variable Declaration most expansion is disabled {name,shell,etc}
3) We can't use symbols or space after =
4)we can use the declare [option] var=value to declare a variable
5)Shouldn't be more than 100 line in the file

Exporting a variable
Export Name='hmada'
It is exported to the env so if we started a new bash it is there
If we assigned a value to is like this without export
NAME='HMADA'

Bash

Echo $NAME

There is no output because it isn't exported as environmental variable
And variable doesn't get inherited through bash unlike environmental variable

Declare and intger
declare -i num=10
declare -x Environmental Variable
declare -r read only

and can be combined
declare -xri num=20

unset is used to remove Variable

reading user input : if the variable exist it will be used if not will be created
read -p "Text" -d "." Variable
-d => delimiter
-p ➔ print Before Entering The value
Ex:

Read -p “Enter Your Name ” name
Read -p “Enter Your Name ” Fname lname
It get 2 variable fist value is the one before Space 2nd one after the space
By word splitting (IFS) Detection
Executing files from another Script
./Script_name
Bash Script_Name

And this script will be executed in the subshell

Only the Environmental variable can be inherited from parent to child

To share the variable we can use Source

Source ./script_name
. ./script_name

Why because it is executed in the main shell

Best Practice by google: https://google.github.io/styleguide/shellguide.html

1) best Practice to use other script it is preferable to use source ./script_name.sh but that doesn’t include the main script
2) start file with #! /bin/bash
3) should be more than 100 lines long
4) ; separate multiple statements

—----------

Numeric Variables & Arithmetic Operations - Work with Numerical Data

Declare -I variable

((result =1+2)) to calculate
To output with assigning it to a variable “$((1+2))”

[^0-9] anything that is not a digit

To Calculate Float we use➔ bc : Basic Calculator
It takes string and returns it after calculating

Echo “12.3 +14.7 ” | bc

Scale can be used to get fixed decimal of number

Echo " scale=4; 15 / 13 " | bc

—----------

Exit Code in Bash

If a program exists it provides an exit code which indicates the success or failure of its execution

Exit code 0 is considered true and any non-zero value is considered false
To access exit code of the last command we can use $?

—----------

Chaining Commands

[command] && [command]
[command] ; [command]

; means ➔it always execute the next command even if command one failed
&&➔ it execute the next command if executed successfully (exit code =0)

Testing Values [[condition]] it will set exit code ➔
0 : if the condition has been fulfilled
1 : If the condition has not been fulfilled

Note: in this case we have to not forget to put space after and before the condition and name expansion path expansion doesn't work inside [[...]]
Etc

—----------

If Statement

```
If command ; then

        #code to be executed if the exit code is 0
else
        #code to be executed if the exit code is 1
Fi
```

EX:

```
If  ls ; then

        #code to be executed  successfully
else
         #code to has failed to execute
fi
```

EX:

```
Read -p “Enter your name ” name

If  [[ $name ==”max” ]] ; then

        echo “hello ${name}”
fi
```

Else IF

EX:

```
If  ping akasdasdasd.com; then

        echo ‘akasdasdasd.com has been pinged’

elif  ping google.com;then

         echo “ ping google.com”
else
        echo “no server could be pinged”
fi
```

testing strings

[[“${filename}” != ‘ ’]] checks if the value of file != o length
Preferred version
[[-n “${filename}”]] checks if the value of file != o length

-n checks if the value != 0

[["${filename}" ! == '']] checks if the value of file = 0 length
Preferred version

[[-z "${filename}"]]
-z checks if the value of file name is not 0 (empty string)

pattern matching
[["file.txt" ==*.txt]] wont work

[[*.txt ==“file.txt"]] wont work
why because it takes first The pattern then the patter we want to test with
? ➔ matchs one charcters and it works inside [[ ]]
[ ]➔ matches the value from the brackets
for more pattern matching
check➔https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

[["file.txt" =~ \.txt$]]
=~ enables the Regular expression

[[-e "${filename }"]] if the files exists
-e ➔exits
-f ➔regular file
-d➔ regular directory
-r ➔ readable
-w ➔ writable
-x ➔ executable

Numeric Test
-lt ➔ Lower Than
-le ➔ lower Than or Equal
-gt ➔ Greater Than
-ge ➔ Greater Than or Equal
-eq ➔ Equal
numeric operation must be done in ((num_file > 15))
or using the numeric test [["${num}" -gt 15]]

negate condtion **[[! -e "${filename }"]]**
&& works like any programming language as And
|| works like any programming language as OR

[["${num}" -gt 15 && "${num}" -lt 20]]

or we can use it like this only for (&&)
why because as we said before && execute the second command in bash if the 2nd one executed without problem

[["${num}" -gt 15]]&& [["${num}" -lt 20]]

Case Statement:

it is like a switch case and ;;➔means Break

read -p "enter the value" value

case "${value}" in

condtion1)
echo #condition 1
;;
condtion2)
echo #condtion 2
;;
esac

pattern works in case

While loop

while command;do
 # code to be executed while the command is true
done

while [[condition]];do
 # code to be executed while the condition is true
done

while ((i < 10));do

```
# code to be executed while the condition is true
echo “${i}”
((i=i+1))
((i+=1))
done
read -r  myline
read and assign the value to myline
```

—----------

for loop

for variable in elements; do

```
#code to execute

done

for name in  “lauren”  “olivar” ;do
        echo “${name}”
done

for number in 1 2 3  4 5 6 7 8 9 10 11 12 ; do
        echo ”${number}”
done
```

note:break , continue works inside for loop

instead of writing number for example we can use
{start..end} {start..end..step}
{1..12..2} or string {a..z} {A..Z}

we can use brace expansion to generate additional element
{.csv,.txt,.docs};

note: brace expansion doesn’t check if the file exist

EX

```
for file in name{.csv,.txt,.docs};do
        echo "${file}"
done

for file in *.txt;do
        echo "${file}"
done
```

out echo all file name ends with .txt

seq start end
generate from the start to end
seq 1 10
from 1 to 10
same as echo {1..10}
but can be used in forloop
Ex

```
read -p "enter start and end " start end
for file in $(seq $start $end );do
        echo "${file}"
done
```

—----------

Selection + Select Case

normal Select

```
select product in "orange" "apple" "mange";do
echo  "$REPLY" : "${product}"
break
done
```

Select + Case

```
select option in "uptime" "user" "free diskspace" "quit";do

  case "option" in
        uptime) uptime ;;
        user) echo "${USER}";;
        "free diskspace") df;;
        quit) break;;
```

```
        *)echo “Option not found”
  esac
done
```

Read Option

`-a` Read the input into an indexed array instead of a single variable.

`-d` Specify a delimiter character other than a newline to terminate the input.

`-p` Specify a prompt string to be displayed before reading the input.

`-r` Do not treat backslashes as escape characters.

`-s` Do not echo the input to the terminal.

`-t` Specify a timeout in seconds to wait for input.
If no input is received within the timeout, the command will exit.

`-u` Read input from the specified file descriptor
(e.g. `-u 3` reads from file descriptor 3).

```
shopt -s extglob # enable linux shell to extended patterns
matching in bash when we enable extglob you can use advenced
pattern matching
```

```
#declare -a arr=(1 9 99 8 55)

#echo ${arr[@]} # to print all array element
#echo ${arr[0]} # print first element
#echo ${arr[@]: 1} # skip first element in array

read -p "enter array size: " size

for ((i=0;i<$size;i++))
do
read -p "please enter element $i: " arr[$i]
done
```

```
declare -i sum=0
for num in ${arr[@]}
do
sum+=$num
done
echo "sum of the array is equal "$sum
((avg=$sum/$size))
echo "the averge is "$avg
echo ${arr[@]}


function hello(){
      echo "in hello fun"
}
hello

hello2(){
      echo "in hello2 fun"
}
hello2

hello(){
      echo "in hello2 fun"
      return 6
}
hello
returnVal=$?
echo the return is $returnVal
'
hello(){
      echo first arg: $1
      echo second arg: $2
      echo all arg: $@
      echo number of arg: $#
      ((sum=$1+$2))
      return $sum
}
hello $1 $2
echo summtion is $?
```

AWK

awk '

BEGIN{

FS=":"

OFS=":"

sum=0

}

{

sum=sum+$3

}

END{

print sum

}

We can assign the output of AWK to VAR

$Variable_Name=$(awk' BEGIN{} {} END{}' FileName)

We assign The value as array

$Variable_Name=(($(awk' BEGIN{} {} END{}' FileName))----------
---

Array

#echo ${arr[@]} # to print all array element

#echo ${arr[0]} # print first element

#echo ${arr[@]: 1} # skip first element in array

#echo ${#arr[@]} print array Size

Redirection &Pipe LInes

`Pipe Line :command1 | command2 | command3 | ... | commandN`

will execute the commands from left to right. It will start with `command1`, and then the output will be input to `command2`. Outputs of `command2` will then be used as inputs of `command3` and so on. The good thing about piping is that you can chain as many commands as you'd like.

```
ls -l | wc -l
```

1. `>` directs the output of a command to a given file.output Redirection
2. `<` directs the contents of a given file to a command. Input Redirection
3. `>>` directs the output of a command to a given file. Appends the output if the file exists and has content.
4. `2>` directs error messages from a command to a given file.
5. `2>>` directs an error message from a command to a given file. Appends the error message if the file exists and has content.
6. `&>` directs standard output and error to a given file.
7. `&>>` directs standard output and error to a given file. Appends to the file if it exists and has contents.

8. (Here String)<<< It takes the string to the right of <<< and feeds it as standard input

```
command <<< "input string"
```

```
and can be combined with other Commands
EX
```

```
lines=($( awk '{print $0}' ~DBMS-Bash-
Project/DataBase/DB1/TB3.meta_data))
```

```
for i in "${!lines[@]}";do
```

```
#Change The field Seperator To + Accept The Value As Array:
Instead of the Default Note: Default IFS That is " "
read -r -a columnContent <<< "${lines[i]}"
```

```
Passing Array Content to The here string and
```

echo "${columnContent[@]}"

MapFIle

```
mapfile [options] array_name
```

`-t`	Removes the trailing newline character from each line before storing it in the array. (Default)
`-n count`	Reads a maximum of `count` lines into the array.
`-O index`	Starts storing lines in the array at the specified index (instead of starting from 0).
`-s count`	Skips the first `count` lines of input.
`-C callback`	Calls a specified function (`callback`) for each line of input.
`-c quantum`	Specifies how many lines to read before invoking the callback.

```
Bash built-in command used to read lines from a file or
standard input into an array
```

```
It simplifies working with files and ensures each line becomes
an element in the array without splitting by spaces or tab
```

Patterns

Wildcards are simple patterns used for matching filenames or strings in Bash.

Pattern	Matches
`*`	Zero or more characters (e.g., `file*` matches `file1`, `file.txt`).
`?`	Exactly one character (e.g., `file?` matches `file1`, but not `file12`).
`[abc]`	Any one character in the set (e.g., `file[12]` matches `file1` or `file2`).
`[a-z]`	Any character in the range (e.g., `[a-d]` matches `a`, `b`, `c`, `d`).
`[^abc]`	Any character **not** in the set (e.g., `[^a-c]` excludes `a`, `b`, and `c`).

for file in *.txt; do echo "Found: $file" done

---

shopt -s extglob

Extended globbing allows more complex pattern matching. Enable it with:

*(pattern): match zero or more occurrence

?(pattern): match zero or one occurrence

+(pattern): match one or more occurrence

@(pattern): match exactly one occurrence

!(pattern): match anything except this pattern

shopt -s extglob for file in *@(txt|csv|log); do echo "Matched file: $file" done

---

3. Pattern Matching in Variables

Removing Prefix or Suffix

Syntax	Description
`${var#pattern}`	Remove the shortest match of `pattern` from the start.
`${var##pattern}`	Remove the longest match of `pattern` from the start.
`${var%pattern}`	Remove the shortest match of `pattern` from the end.
`${var%%pattern}`	Remove the longest match of `pattern` from the end.

FILENAME="EXAMPLE.TAR.GZ"

echo "${filename%.gz}" # Output: example.tar

echo "${filename%%.*}" # Output: example

---

5. [[WITH PATTERN MATCHING

In `[[ ... ]]`, patterns are used for string comparison.

Operator	Description
==	Matches a pattern (supports wildcards).
!=	Does not match a pattern.

name="JohnDoe" if [[$name == J*]]; then echo "Name starts with J" fi

---

6. REGULAR EXPRESSIONS IN BASH

With `[[ ... =~ ... ]]`, you can use extended regular expressions for matching.

Operator	Description
=~	Matches a regular expression.

-gt greater than

-ge greater than or equal

-le less than or equal

-lt less than

-eq equal

-ne not equal

#"current process: " $$

#"name of file: " $0

#"number of args: " $#

#"print all args: " $*

#"print all args: " $@

#"first args: " $1

#"second args: " $2

Declaring Variables with declare

The `declare` command in Bash allows you to set attributes for variables. Here are some common options:

- `-a`: Declare an array.
- `-i`: Declare an integer variable.
- `-r`: Declare a read-only variable.
- `-x`: Export the variable to the environment.

Sed

sed [OPTIONS] 'script' file

Common Options

Option	Description
`-e`	Allows multiple editing commands in a single `sed` invocation.
`-f`	Reads editing commands from a file.
`-n`	Suppresses automatic printing of lines (used with `p` command to print).
`-i`	Edits files **in-place** (modifies the file directly).
`-r` or `-E`	Enables extended regular expressions.
`-l NUM`	Sets the line-wrap length for `l` command output.
`--help`	Displays a help message with usage information.
`--version`	Displays the version of `sed`.

Basic Commands (Script Syntax)

Command	Description

Command	Description
`s/pattern/replacement/`	Substitutes a pattern with a replacement (search and replace).
`p`	Prints the current pattern space (requires `-n` to suppress auto-printing).
`d`	Deletes the current line or matching lines.
`a\ text`	Appends text after the current line.
`i\ text`	Inserts text before the current line.
`c\ text`	Changes the content of matching lines to the given text.
`q`	Exits immediately after processing the specified line(s).
`r file`	Reads a file and inserts its content after the current line.
`w file`	Writes the current pattern space to a file.
`y/source/dest/`	Translates (replaces) characters in `source` with corresponding `dest`.

Regular Expressions in sed

- **Anchors**:
  - `^`: Matches the start of a line.
  - `$`: Matches the end of a line.
- **Character Classes**:
  - `[abc]`: Matches any one character (a, b, or c).
  - `[a-z]`: Matches any lowercase letter.
  - `[^abc]`: Matches anything except a, b, or c.
- **Quantifiers**:
  - `*`: Zero or more occurrences.
  - `+`: One or more occurrences (requires `-E` or `-r`).
  - `?`: Zero or one occurrence.
  - `{n,m}`: Between `n` and `m` occurrences.

1. **Search and Replace**

sed 's/old/new/' file.txt # Replace the first occurrence of "old" with "new" on each line

sed 's/old/new/g' file.txt # Replace all occurrences of "old" with "new" globally

sed -i 's/old/new/' file.txt # Edit the file in-place (modifies the file)

2. **Delete Lines**

sed '2d' file.txt # Deletes the 2nd line

sed '/pattern/d' file.txt # Deletes lines matching "pattern"

sed '1,3d' file.txt # Deletes lines from 1 to 3

3. **Print Specific Lines**

sed -n 'p' file.txt # Prints all lines (same as `cat`)

sed -n '2p' file.txt # Prints the 2nd line only

sed -n '/pattern/p' file.txt # Prints lines matching "pattern"

4. **Insert, Append, or Change Text**

sed '2i\Inserted text' file.txt # Inserts text before the 2nd line

sed '2a\Appended text' file.txt # Appends text after the 2nd line

sed '2c\Changed text' file.txt # Replaces the 2nd line with "Changed text"

5. **Read or Write to a File**

sed '2r additional.txt' file.txt # Reads and inserts content of "additional.txt" after the 2nd line

sed '/pattern/w output.txt' file.txt # Writes lines matching "pattern" to "output.txt"

sed 'y/abc/123/' file.txt # Replaces 'a' with '1', 'b' with '2', and 'c' with '3'

8. **Using Multiple Commands**

sed -e 's/foo/bar/' -e 's/hello/world/' file.txt

7. **Quit Early**

sed '3q' file.txt # Stops processing after the 3rd line