Helper Setu

2025 December 28

BASH SCRIPTING

Aziz Bohra

Bash (Bourne Again Shell) is a command-line language used to interact with the operating system.

Bash scripting means:

writing a file that contains multiple terminal commands and running them together automatically.

Instead of typing commands one by one, we automate tasks using a script.

Why Bash

- **Server Management**

Bash is used to start, stop, and restart backend services.

- **Deployment Automation**

Bash automates code pull, build, and deploy processes.

- **CI/CD Pipelines**

CI/CD tools execute Bash commands to test and deploy code.

- **Environment Configuration**

Bash sets environment variables required by backend apps.

- **Log Monitoring**

Bash helps read and monitor logs using commands like tail.

- **Debugging Production Issues**

Bash is used to debug crashes and errors directly on servers.

- **Automation Tasks**

Bash automates repetitive backend tasks such as backups and cleanup.

- **Cron Jobs**

Bash scripts are scheduled using cron for periodic backend jobs.

- **Docker & Containers**

Bash scripts run as Docker entrypoints to prepare containers.

- **Cloud & Infrastructure**

Bash is the primary tool to manage cloud servers without GUI.

Basic

Core Commands

```
pwd          # where am I
ls           # list files
ls -la       # detailed list
cd folder    # move
cd ..        # go back
mkdir app
rm file
rm -rf folder
cp a b
mv a b
```

File viewing

```
cat file
less file
head file
tail -f logfile.log
```

Permissions

```
ls -l
chmod +x script.sh
```

Basic

COPY

Copies file a → creates a new file b

```
cp a b
```

- a remains unchanged
- b is a duplicate of a

Copy folder

```
cp -r src backup_src
```

Basic

Rename/Move

Rename File

```
mv app_old.js app.js
```

Move File

```
mv app.js scripts/
```

Basic

Less Read File Safely

Opens file in a scroll able viewer

Controls

- ↑ ↓ → scroll
- Space → page down
- /word → search
- q → quit

Basic

head

Show first 10 line

```
head file
```

head custom line

Show first 20 line

```
head -n 20 file
```

Basic

tail

Show last 10 line

```
tail file
```

custom line

Show last 50 line

```
tail -n 50 file
```

Live Logs

```
tail -f logfile.log
```

Basic

Terminal Sysmbols

PIPE (Connect output → input) |

Take output of left command → send it as input to right command

```
ls | wc -l
```

Redirect output (overwrite) >

Send output to a file (overwrite if exists)

```
echo "Hello" > hello.txt
```

Basic

Terminal Sysmbols

Redirect output (append) >>

Send output to file (append) but append the output to a file

```
echo "Log entry" >> app.log
```

Run Process in Backround &

Run command without blocking terminal

```
command &
```

Redirect input <

Take input from a file instead of keyboard

```
wc -l < users.txt
```

Comments

Comments start with a # and Bash ignores them. They explain what your code does, making it easier to understand.

```
# This script prints a greeting message
echo "Hello, World!"
```

Command Execution Order

Commands are run (or executed) in sequence from top to bottom

```
echo "First command"
echo "Second command"
```

Semicolons

Semicolons ; can be used to separate multiple commands on the same line, which is useful for writing concise scripts.

```
echo "This is a test"; echo "This is another test"
```

Creating a file

```
touch demo.sh
```

Changing mode to execution

```
chmod +x demo.sh
```

Basic syntax

```
#!/bin/bash
# This script prints a greeting message
echo "Hello, World!"
```

Input at Script Execution (Command-line Argument)

- User provides input while running the script
- Accessed using $1, $2, $3, ...

```
#!/bin/bash

name=$1
echo "Hello $name"
```

Input While Program is Running (read)

- Script asks user for input interactively
- Uses read command

```
#!/bin/bash

read -p "Enter your name: " name
echo "Hello $name"
```

Variables

Variables are declared by simply assigning a value to a name. There should be no spaces around the equal sign:

- variable_name=value
- To access the value of a variable, prefix it with a dollar sign: $variable_name

```
name="John Doe"
echo "Hello, $name!"
number=42
echo "The number is $number"
```

Data Types

String

Strings are sequences of characters used to store text.

```
#concate variable
name="John"
surname="Deo"
fullname=$name" "$surname
echo $fullname
```

Number

Numbers in Bash can be used for arithmetic operations. Bash supports integer arithmetic natively, such as addition, subtraction, multiplication, and division.

```
# Number example
num1=5
num2=10
sum=$((num1 + num2))
difference=$((num2 - num1))
product=$((num1 * num2))
quotient=$((num2 / num1))
echo "Sum: $sum, Difference: $differe
```

Arrays

Arrays are used to store multiple values in a single variable. Each element in an array is accessed using an index. You can iterate over arrays and modify elements.

```
#Array

fruits=("apple" "mango" "banana")
echo ${fruits[0]}
echo ${fruits[1]}
echo ${fruits[2]}

echo "All element is array"

echo ${fruits[@]}

echo "Length of array"

echo ${#fruits[@]}
```

Associative Arrays (Object) (key:value)

Associative arrays allow you to use named keys to access values. They are similar to dictionaries in other programming languages. You can add or remove keys and values.

```
declare -A colors

colors[apple]="red"
colors[mango]="yellow"
colors[grape]="green"

echo ${colors[apple]}
echo ${colors[mango]}

#deleting field from associative array

unset colors[graps]

echo $colors
```

Operators

Comparison Operators

- -eq: Equal to
- -ne: Not equal to
- -lt: Less than
- -le: Less than or equal to
- -gt: Greater than
- -ge: Greater than or equal to

#0 means true
#1 means false
$? means: exit status of the last executed command

```
echo "Comparison operators []"
[ 10 -eq 20 ]; echo $?
[ 10 -ne 20 ]; echo $?
[ 10 -lt 20 ]; echo $?
[ 5  -le 30 ]; echo $?
[ 20 -gt 12 ]; echo $?
[ 30 -ge 25 ]; echo $?

echo "Comparison operators in usin (())"

((10>4)); echo $?
((10==4)); echo $?
((10!=4)); echo $?

((10>=4)); echo $?
((10<=4)); echo $?
((10<4)); echo $?
```

Operators

Logical Operators

&&: Logical AND
||: Logical OR
!: Logical NOT

#0 means true
#1 means false
$? means: exit status of the last executed command

```
echo "Logical operators"
#square bracket
[[ 10 -eq 10 && 30 -gt 20 ]]
echo $?

#parentheses

((10 == 10 && 30 > 20))
echo $?

((10 == 20 || 30>50))
echo $?

(( !(10 > 20) ))
echo $?
```

Other Imp Operators

Shell Options (Global Script Behavior)

```
set -<flag>
```

Flag	Meaning
`-e`	Exit on error
`-u`	Error on undefined vars
`-x`	Print commands (debug)
`-o pipefail`	Fail on pipeline errors

String operator

Operator	Meaning
`-z`	string is empty
`-n`	string is not empty

Other Imp Operators

File Operators

Operator	Meaning
`-f`	file exists
`-d`	directory exists
`-x`	executable
`-r`	readable
`-w`	writable

```bash
#!/bin/bash

file="demo.txt"
dir="logs"

[ -f "$file" ] && echo "-f : file exists"        # checks if file exists
[ -d "$dir" ]  && echo "-d : directory exists"   # checks if directory exists
[ -r "$file" ] && echo "-r : file is readable"   # checks read permission
[ -w "$file" ] && echo "-w : file is writable"   # checks write permission
[ -x "$file" ] && echo "-x : file is executable" # checks execute permission
```

Control Flow

If-Else

Using parentheses

```
#!/bin/bash

b=20
c=15

if (( b > c )); then
    echo "b is bigger"
else
    echo "c is bigger"
fi
```

```
echo "if flow"
a=5
if [ $a -gt 2 ]; then
        echo "a is 5"
fi


echo "if else flow"
b=10
c=20

if [ $b -gt $c  ]; then
        echo "b is bigger"
else
        echo "c is bigger"
fi


echo "if ele-if control flow"

d=15

if [ $d -eq 10 ]; then
        echo "d is 10"
elif [ $d -lt 10  ]; then
        echo "d is less than 10"
else
        echo "d is bigger than 10"
fi
```

For loop

For loops allow you to iterate over a list of items or a range of numbers. They are useful for repeating tasks a specific number of times.
The for keyword is followed by a variable name, a range of values, and a do keyword, which marks the start of the loop block.

```
for i in {1..5}; do
        echo "iterate $i"
done

echo "for loop with array"

fruits=("apple" "mango" "banana")

for((i=0; i<${#fruits[@]}; i++ )); do
        echo "${fruits[i]}"
done
```

While Loop

While loops execute a block of code as long as a specified condition is true.
They are useful for tasks that need to repeat until a certain condition changes.

**loops start with do and end with done

```
echo "while loop"

a=1

while((a <= 5)); do
        echo "$a"
        ((a++))
done

echo "while loop using square baracket"

while [ $a -le 10  ]; do
        echo "$a"
        ((a++))

done
```

Loops

Until Loop

Until loops are similar to while loops, but they execute until a specified condition becomes true.

a=10

```
echo "until loop"
until [ $a -eq 15 ]; do
        echo "$a"
        ((a++))
done
```

```
# Until loop example
count=1
until [ $count -gt 5 ]; do
  echo "Count is $count"
  ((count++))
done
```

Loops

Break and Continue

Break and continue statements are used to control loop execution. break exits the loop, while continue skips to the next iteration.

These statements are typically used inside conditional blocks to alter the flow of the loop.

```
echo "break and continue in loop"
while [ $a -le 25  ]; do
        if [ $a -eq 20  ]; then
                ((a++))

                continue
        fi
        echo "$a"
        if [ $a -eq 22  ]; then
                ((a++))

                break
        fi
        ((a++))
done
```

```
for i in {1..5}; do
  if [ $i -eq 3 ]; then
    continue
  fi
  echo "Number $i"
  if [ $i -eq 4 ]; then
    break
  fi
done
```

Nested Loops

Nested loops allow you to place one loop inside another, enabling more complex iteration patterns. Each loop must be closed with its own done.

```
# Nested loops example
for i in {1..3}; do
  for j in {1..2}; do
    echo "Outer loop $i, Inner loop $j"
  done
done
```

Loops

Function

Defining Functions:

To define a function in Bash, use the following syntax. The function name is followed by parentheses, and the function body is enclosed in curly braces.

Calling Functions:

In Bash, execute (or call) a function by using its name.

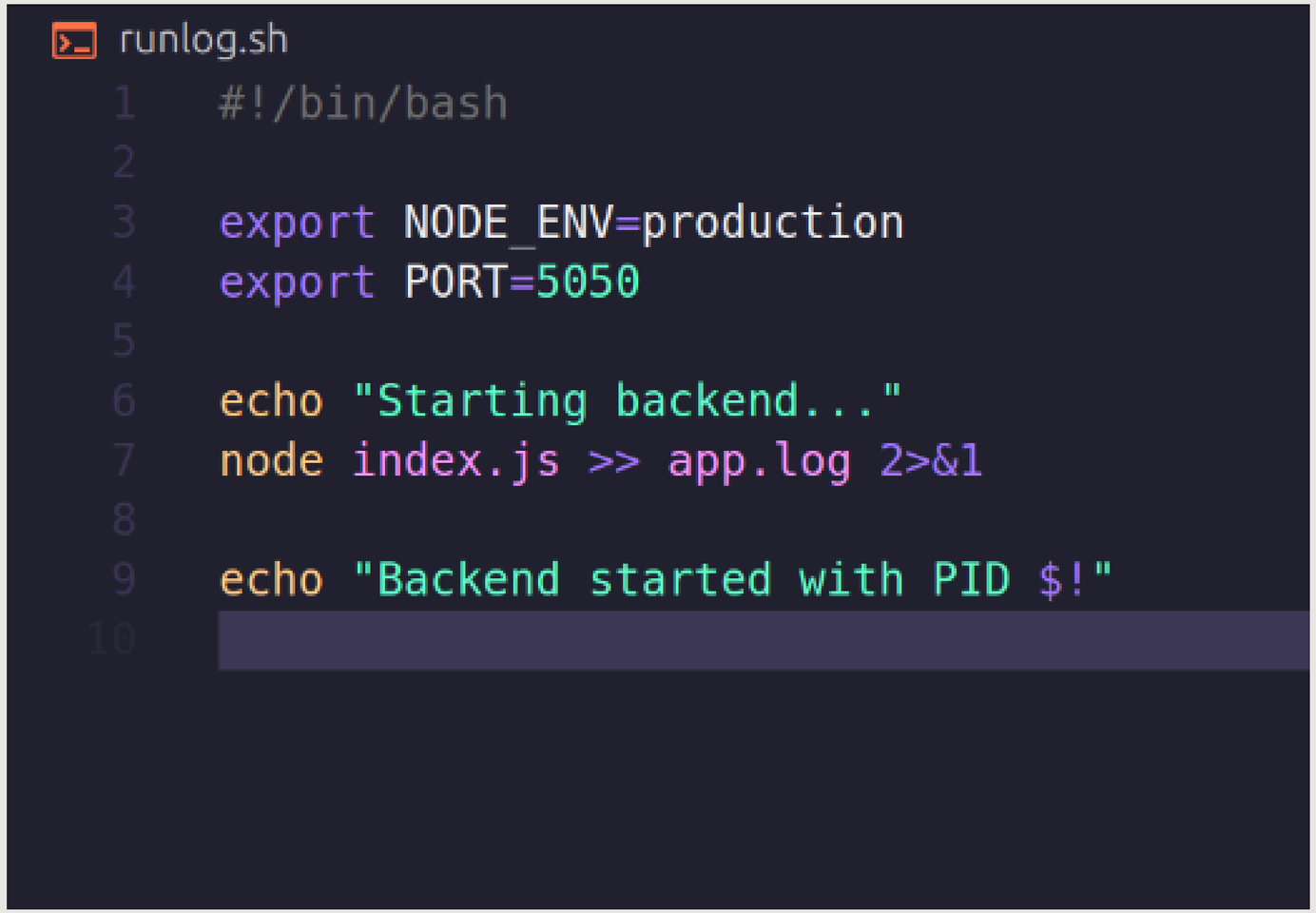
Advanced Function Features

Functions can accept arguments, return values, and use local variables. Here's an example of a function that takes an argument and uses a local variable

```
greet () {
        echo "hello world"
}


greet


sum () {
        local sum=$(($1 + $2))
        echo $sum
}

sum 1 2

result=$(sum 1 2)

echo "the sum is $result"
```

BASIC BASH SCRIPT FOR BACKEND

Starting a back-end and logs



Seeding data

seed.sh

```bash
#!/bin/bash

BASE_URL=http://localhost:5050

animes=("One Piece" "Bleach" "Attack on Titan")

for name in "${animes[@]}"; do
  curl -s -X POST $BASE_URL \
    -H "Content-Type: application/json" \
    -d "{\"name\":\"$name\",\"rating\":8}" | jq
done

echo "Data seeded"
```

Getting all data

getAllData.sh

```
1   #!/bin/bash
2
3   BASE_URL=http://localhost:5050
4
5   echo "Fetching all anime..."
6   curl -s $BASE_URL | jq
7
```

Post data

```bash
#!/bin/bash

BASE_URL=http://localhost:5050

curl -X POST $BASE_URL \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Naruto",
    "mangaWriter": "Masashi Kishimoto",
    "studio": "Pierrot",
    "rating": 9
  }'
```

Update Data

update.sh

```
#!/bin/bash

BASE_URL=http://localhost:5050
ID=$1

curl -X PUT $BASE_URL/$ID \
  -H "Content-Type: application/json" \
  -d '{"rating":10}'

```

Delete Data

delete.sh

```
#!/bin/bash

BASE_URL=http://localhost:5050
ID=$1

result=curl -X DELETE $BASE_URL/$ID | jq
echo "Deleted anime with id $ID"
echo "$result"
```

Git Merge Script

gitMerge

```
# branch name
devBranch=$1

if [ -z "$devBranch" ]; then
  echo "Error: Branch name not provided"
  exit 1
fi

echo "Switching to main branch..."
git switch main || exit 1

echo "Pulling latest changes from main..."
git pull || exit 1

echo "Switching back to $devBranch..."
git switch "$devBranch" || exit 1

echo "Merging main into $devBranch..."
git merge main

echo "Merge completed for branch: $devBranch"
```

THANK YOU

For Listening

GitHub Repo for all programs used in presentation

Aziz Bohra