Deloitte.

Tales of Interest

The power of BASH PROGRAMMING

Alexandra Baga

Deloitte.

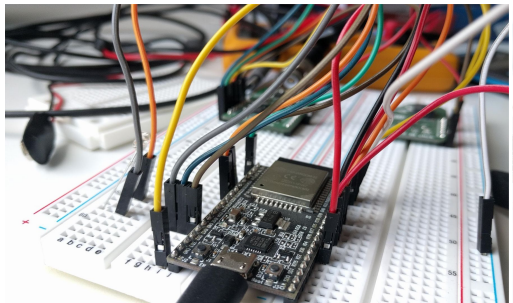
I am

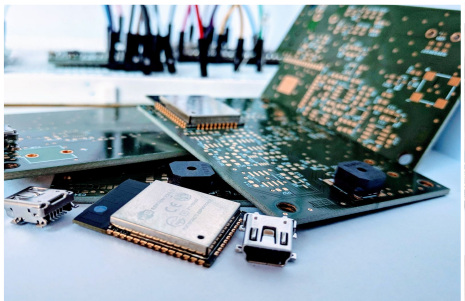
Alexandra Baga
I was born in **Odessa** , Ukraine
I live in Berlin since **2014.**
I am with Deloitte since December 2022

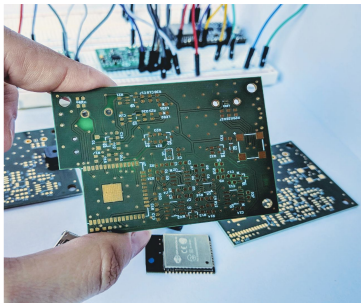
I studied **Embedded Systems** (B.Eng) and **Computer Science** (MCS).
I know electrical circuits, can work with oscilloscope and understand binary numbers.

Last years my specialisation is shifted to **backend programming, data engineering, REST APIs and cloud technologies.**

I enjoy going to the gym, jogging, spending time in nature, reading books, and playing with my son.







Introduction

What is Bash Programming and why is it Important for developers

01

02

Real life example

Small usage of Bash saved my time during my master's thesis

How to Bash

Bash Scripting Tutorial for beginners

03

04

Error handling

Creating robust and reliable scripts

Special variables

Hard to read by humans but very powerful

05

06

Cool Bash Scripts

Few examples to boost your fantasy using Bash

What is Bash Programming

Bash (Bourne Again Shell) is a **Unix shell and command-line interface** that provides a **powerful scripting language** for **automating** tasks, managing systems, and enhancing productivity.

It's widely used across **Linux** and **macOS** systems, and it is popular in daily and routine tasks by developers, sysadmins, and DevOps engineers.

Windows users must **additionally enable Subsystem for Linux** in their Win OS to be able to use Bash in Windows.

```
#!/bin/bash

# Take user input
echo "Enter your name:"
read name

# Take input from arguments
arg1=$1
arg2=$2

# Print the variables
echo "Your name is: $name"
echo "Argument 1 is: $arg1"
echo "Argument 2 is: $arg2"
```

Picture source: https://nowitsanurag.medium.com/a-step-by-step-guide-to-creating-a-bash-script-in-linux-b2bc02e892d

What is Bash Programming

A Bash **script** is a **text file** containing a series of commands written for the Bash shell.

Anything that exists in a Bash script can also be executed in the command line.

```
#!/bin/bash

# Take user input
echo "Enter your name:"
read name

# Take input from arguments
arg1=$1
arg2=$2

# Print the variables
echo "Your name is: $name"
echo "Argument 1 is: $arg1"
echo "Argument 2 is: $arg2"
```

Picture source: https://nowitsanurag.medium.com/a-step-by-step-guide-to-creating-a-bash-script-in-linux-b2bc02e892d

Why Bash is important for developers?

Bash closes the gap between manual operations and full-scale software automation.

Whether you're setting up environments, deploying applications, or parsing logs, Bash allows you to work faster and smarter.

Ok, how exactly can Bash help me?

Picture source: https://www.reddit.com/r/ProgrammerHumor/

Why Bash is important for developers?

1. Automation of Tasks: **Write scripts to schedule backups, clean logs, or deploy applications.**

Repetitive Tasks:
Bash scripts can automate repetitive tasks such as backups, file management, and system updates, saving time and reducing the risk of human error.

Scheduled Jobs:
Scripts can be scheduled using tools like cron to run at specific intervals, ensuring tasks are performed consistently without manual intervention.

Picture source: https://www.ntaskmanager.com/blog/task-automation/

Why Bash is important for developers?

2. System Administration: **Monitor processes, manage user accounts, and configure systems**

System Monitoring:
Bash scripts can be used to monitor system resources, check system health, and log data for analysis.

User Management:
Admins can automate the creation, modification, and deletion of user accounts and permissions.

Picture source: hhttps://www.softxpert.com/services/network-system-administration/

Why Bash is important for developers?

3. Data Processing: Parse logs, extract meaningful information, and format data for analysis.

Data preprocessing:
Bash can transforming data directly from the command line. Using tools like **grep**, **awk**, **sed**, and **cut**, you can parse logs, extract meaningful information, and format data for analysis efficiently.

Let you quickly process large datasets or logs without relying on heavy external tools.

Picture source: DeepAi Prompt Result

02

Real life example

Small usage of Bash saved my time during my master's thesis

Example from the real life

I used a lot Bash scripts during my Master Thesis at Fraunhofer Institute.

I developed an ML approach and needed to run the computational task hundreds of times to find the best model parameters for accurately predicting user positions in virtual reality.

Picture source: DeepAi Prompt Result

Automate creation of tuning model parameters

```
#!/bin/bash
echo "Bash version ${BASH_VERSION}..."
out=experiment_parameters_short.csv

echo "hidden_dim,batch_size,lr_adam,lr_epochs,lr_multiplicator" > $out

for hidden_dim in $(seq 8 1 10)
    do
        for batch_size in $(seq 8 1 10)
        do
            for lr_adam in $(seq 0.0001 0.0001 0.0001)
            do
                for lr_epochs in $(seq 50 10 50)
                do
                    for lr_multiplicator in $(seq 0.3 0.2 0.5)
                    do
                        echo "$((2**$hidden_dim)),$((2**$batch_size)),$lr_adam,$lr_epochs,$lr_multiplicator" >> $out
                    done
                done
            done
        done
    done
echo "$out is written"
```

Picture source: DeepAi Prompt Result

The created model parameters csv-file examples

1	hidden_dim	batch_size	lr_adam	lr_epochs	lr_multiplicator	weight_decay_adam	layers	dropout	n_epochs
2	256	128	0.0001	50	0.5	0.000000000001	1	0	500
3	256	128	0.0001	50	0.5	0.0000000000001	1	0	500
4	256	128	0.0001	50	0.5	0.00000000000001	1	0	500
5	256	128	0.0001	50	0.5	0.00000000001	1	0	500
6	256	128	0.0001	50	0.5	0.000000000001	1	0	500
7	256	128	0.0001	50	0.5	0.000000000001	1	0	500
8	256	128	0.0001	50	0.5	0.000000000002	1	0	500
9	256	128	0.0001	50	0.5	0.000000000003	1	0	500
10	256	128	0.0001	50	0.5	0.0000000000005	1	0	500
11	256	128	0.0001	50	0.5	0.0000000000008	1	0	500
12	256	128	0.0001	50	0.5	0.0000000000009	1	0	500
13	512	256	0.0001	50	0.5	0.000000000001	1	0	500
14	512	256	0.0001	600	0.5	0.000000000001	1	0	500

1	hidden_dim	batch_size	lr_adam	lr_epochs	lr_multiplicator	n_epochs
2	16	16	0.0001	50	0.5	500
3	16	32	0.0001	50	0.5	500
4	16	64	0.0001	50	0.5	500
5	16	128	0.0001	50	0.5	500
6	16	256	0.0001	50	0.5	500
7	16	512	0.0001	50	0.5	500
8	16	1024	0.0001	50	0.5	500
9	16	2048	0.0001	50	0.5	500
10	32	16	0.0001	50	0.5	500
11	32	32	0.0001	50	0.5	500
12	32	64	0.0001	50	0.5	500
13	32	128	0.0001	50	0.5	500
14	32	256	0.0001	50	0.5	500
15	32	512	0.0001	50	0.5	500

Automate setting env variables when jobs start on cluster

```
#!/bin/bash

while IFS="," read -r hidden_dim batch_size lr_adam lr_epochs lr_multiplicator weight_decay_adam n_epochs


do
  echo "$hidden_dim $batch_size $lr_adam $lr_epochs $lr_multiplicator $n_epochs"
  export HIDDEN_DIM=$hidden_dim
  export BATCH_SIZE=$batch_size
  export LR_ADAM=$lr_adam
  export LR_EPOCHS=$lr_epochs
  export LR_MULTIPLICATOR=$lr_multiplicator
  export N_EPOCHS=$n_epochs
  export WEIGHT_DECAY_ADAM=$weight_decay_adam
  export RNN_PARAMETERS=1
  nohup sbatch UserPrediction6DOF.sh &
done < <(tail -n +2 $1)
```

Schedule and run jobs on cluster

```
24  OUT_DIR=thesis_experiments
25
26  source "/etc/slurm/local_job_dir.sh"
27  mkdir -p "${LOCAL_JOB_DIR}/job_results"
28  mkdir -p "${LOCAL_JOB_DIR}/job_results/figures"
29  mkdir -p "${LOCAL_JOB_DIR}/job_results/tabular"
30  mkdir -p "${LOCAL_JOB_DIR}/job_results/tabular/distances"
31  mkdir -p "${LOCAL_JOB_DIR}/job_results/predictions"
32  mkdir -p "${LOCAL_JOB_DIR}/job_results/losses"
33  mkdir -p $SLURM_SUBMIT_DIR/$OUT_DIR
34
35  # run job and bind the output dir
36  # Launch the singularity image with --nv for nvidia support.
37  # The job writes its results to stdout which is directed to the output which starts with the job number file. Check it.
38  singularity run --nv  --bind ${LOCAL_JOB_DIR}:/mnt/output ./UserPrediction6DOF.sif
39
40  # Store Intermediate Data and Results Locally
41  # Doing this after the singularity run call ensures, that the data is copied back even when your singularity run fails.
42  cd $LOCAL_JOB_DIR
43  tar -zcvf zz_${SLURM_JOB_ID}.tar job_results
44  cp zz_${SLURM_JOB_ID}.tar $SLURM_SUBMIT_DIR/$OUT_DIR
45  rm -rf ${LOCAL_JOB_DIR}/job_results
```

Automate the parsing the archived results

The required outputs were archived in tar files. I had hundreds of them, and I needed to determine which model parameters were the best.

1. Iterates over all files in the specified directory (process_dir).
2. Extracts (-x) the contents of the compressed archive (.tar.gz)
3. **sed -n '2p':** Extracts the second line of the CSV file (2p) with model output
4. Appends the extracted data ($log) to the CSV file created earlier.

```
1 for file in "$process_dir"/*
  do
2 out=$((tar -xOzf $file job_results/model_parameters_adjust_log.csv) >&1)
3 log=$(echo -e "$out" | sed -n '2p')
4 echo $log >> results/model_tuning_logs/$process_dir.csv
  done
```

03

How to Bash

Bash Scripting Tutorial
for beginners

First steps for beginners

To find out where is your Bash interpreter located you can type:

```
$ which bash
```

The output is supposed to be something like this:

```
/bin/bash
```

To create the Bash script, simple create a file with extension .sh

```
vim my_first_script.sh
```

First steps for beginners

The Bash Script first line starts with the #! characters (**shebang**) and the path to the Bash interpreter. You can use two variants:

#!/bin/bash: Specifies the exact path to the bash interpreter.

If bash is installed in a non-standard location , the script will fail to execute.

#!/usr/bin/env bash: Uses the **env** command to locate bash in **PATH**, dynamically finds the bash interpreter, whether it's in /bin, /usr/local/bin, or another location. Important is the correctness of the user's PATH variable.

If bash isn't in the PATH, the script will fail to execute.

Understanding stdin and stdout in Bash

In Bash (and Unix-like systems), **stdin** (standard input) and **stdout** (standard output) are two primary streams that handle input and output for commands and scripts.

STDIN: Represents the input stream used by a command or script.

By default, it takes input from the keyboard. Stdin is associated with file descriptor 0.

$ 01_sdtin_01.sh U ×

$ 01_sdtin_01.sh

```
1  #!/bin/bash
2  echo "Enter your name:"
3  read name  # Takes input from stdin (keyboard by default)
4  echo "Hello, $name!"
5
```

sh 01_sdtin_01.sh

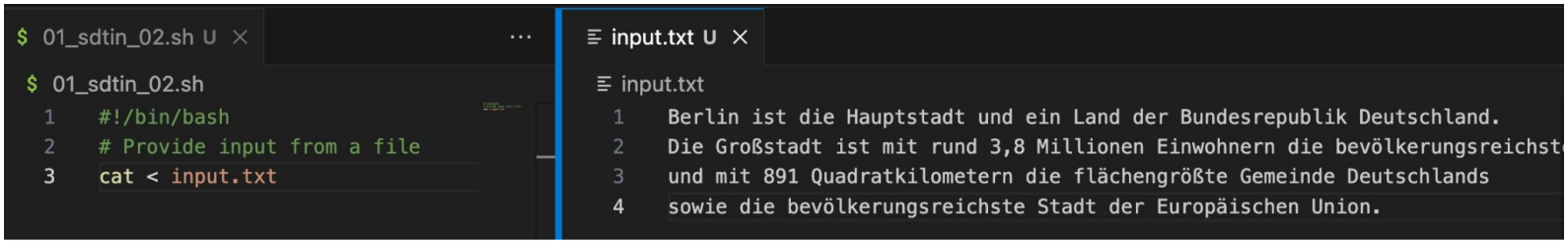
```
Enter your name:
Alexandra
Hello, Alexandra!
```

Understanding stdin and stdout in Bash

In Bash (and Unix-like systems), **stdin** (standard input) and **stdout** (standard output) are two primary streams that handle input and output for commands and scripts.

STDIN: You can provide input to a command from a file or another command.

cat by default sends the input it receives to stdout (standard output), which is the terminal.



Understanding stdin and stdout in Bash

STDOUT: Represents the output stream where commands send their results.

By default, this is displayed in the terminal. **You can redirect it similar to stdin as showed before.** Stdout is associated with file descriptor 1.

$ 01_sdtout.sh

```
1  #!/bin/bash
2  echo "This is stdout, you see it in the terminal window!"
3
4  # Redirect stdout to a file
5  echo "Let's write a first line to the file" > output.txt
6
7  # Append stdout to a file
8  echo "And another line so that the previous line is kept in the file!" >> output.txt
```

Combining stdin and stdout

You can use redirection to chain commands, process files, or manage streams:

```
1    # Sort the content of a file in reverse order
2    # and save the result to another file
3    sort -r < input.txt > sorted_output.txt
```

sorted_output.txt

```
1    und mit 891 Quadratkilometern die
2    sowie die bevölkerungsreichste St
3    Die Großstadt ist mit rund 3,8 Mi
4    Berlin ist die Hauptstadt und ein
5
```

Pipe (|): Connects one command to another:

```
5    cat input.txt | grep "Deutschland" | sort
```

```
Berlin ist die Hauptstadt und ein Land der Bundesrepublik Deutschland.
und mit 891 Quadratkilometern die flächengrößte Gemeinde Deutschlands
```

Use pipe to combine several commands

Pipe (|): an useful example using piping in Bash, for file processing and analysis — something developers can often deal with ;) **We will display the top 10 most frequent words from file.**

```
find . -type f -name "*.txt" | xargs cat | tr -s '[:space:]' '\n'
| grep -E '.{3,}' | sort | uniq -c | sort -nr | head -10
```

Finds all .txt files in the current directory and its subdirectories.

Combines the list of files found by find and feeds them into cat to concatenate their contents.

Transforms all whitespace (spaces, tabs, etc.) into newlines, effectively splitting the text into individual words.

Filters out words shorter than 3 characters (e.g., if, in, an).

Sorts the words alphabetically.

Counts occurrences of each unique word and finally sorts words numerically in descending order.

04 Error handling

Creating robust and reliable scripts

Check Exit Status

Every command in Bash returns an exit status:

0: Success. 👌 **Non-zero: An error occurred.** 🙈

You can check the exit status of a command using `$?`:

`$?` stores the exit status of the last executed command.

If we print `$?` after an `if` statement, the value will be `0` because the comparison succeeded!

```
2   cp file1.txt file2.txt
3
4   # Capture the exit status of the last command
5   exit_status=$?
6
7   # Use the stored exit status later in the script
8   if [ $exit_status -ne 0 ]; then
9     echo "Error: The last command failed with exit status $exit_status."
10  else
11    echo "The last command succeeded with exit status $exit_status."
12  fi
```

I don't have these files and cp command returns 1 (fail)

```
cp: file1.txt: No such file or directory
Error: The last command failed with exit status 1.
```

Exit on errors

I still don't have these files and cp command still fails :)

set -e: Exit on Errors

Automatically terminates the script if a command exits with a non-zero status.

```
set -e
# If cp command fails, the script stops.
cp file1.txt file2.txt
echo "This line won't be printed"
```

```
Options:
  -a  Mark variables which are modified or created for export.
  -b  Notify of job termination immediately.
  -e  Exit immediately if a command exits with a non-zero status.
  -f  Disable file name generation (globbing).
  -h  Remember the location of commands as they are looked up.
  -k  All assignment arguments are placed in the environment for a
      command, not just those that precede the command name.
  -m  Job control is enabled.
  -n  Read commands but do not execute them.
  -o option-name
```

```
cp: file1.txt: No such file or directory
Error: The last command failed with exit status 1.
```

https://linuxcommand.org/lc3_man_pages/seth.html

Exit on errors

set -u: Treat Undefined Variables as Errors

Exits the script if you reference an undefined variable.

```
1  #!/bin/bash
2  # This won't cause an error.
3  echo "Line 3 -> The value is: $undefined_var"
4
5  set -u
6  # This will cause an error.
7  echo "Line 7 -> The value is: $undefined_var"
```

```
Line 3 -> The value is:
03_set_u.sh: line 7: undefined_var: unbound variable
```

Line 7 was not executed because the script exited when an undefined variable was referenced.

https://linuxcommand.org/lc3_man_pages/seth.html

Use trap for Cleanup

The trap command lets you specify actions to take when the script exits or encounters signals (like SIGINT).

```bash
# Define the trap for cleaning up when the script exits
trap 'echo "Cleaning up..."; rm -f tempfile; echo "Cleanup completed!"' EXIT
echo "Starting script... Creating a temporary file."

# Create a temporary file
touch tempfile
echo "Temporary file created at tempfile"
# Simulate some processing
echo "Processing..."

# Loop to keep the script running until 'exit' is typed
while true; do
  # Read user input
  read -p "Type 'exit' to terminate the script: " user_input

  if [ "$user_input" == "exit" ]; then
    echo "User requested to exit. Exiting now."
    exit 0  # This will trigger the trap and clean up the tempfile
  else
    echo "You typed: $user_input"
  fi
done
```

```
Starting script... Creating a temporary file.
Temporary file created at tempfile
Processing...
Type 'exit' to terminate the script: exit
User requested to exit. Exiting now.
Cleaning up...
Cleanup completed!
```

If you terminate a script using `Ctrl+C`, a `SIGINT` (Signal Interrupt) signal is sent to the script.
By default, **this will cause the script to terminate immediately**, but you can **handle this signal using `trap` to perform specific actions,** like cleanup, before the script exits.

Advanced Error Handling

Write reusable functions to handle errors consistently.

Example: Log and Exit on Errors

```
error_exit() {
  echo "Error: $1" >&2
  exit 1
}

cp file1.txt file2.txt || error_exit "Failed to copy file."
```

```
cp: file1.txt: No such file or directory
Error: Failed to copy file.
```

Redirect Errors to a file or suppress them.

```
cp file1.txt file2.txt 2>> error.log
```

error.log

```
1   cp: file1.txt: No such file or directory
```

⚠️ &2 refers to the **standard error (stderr)** stream. It's used for redirecting output to the standard error stream, which is file descriptor 2.

A file descriptor 3 (or any number higher than 2) is simply a custom file descriptor that refers to an open file or stream that isn't one of the standard input/output/error streams.

Advanced Error Handling

Use logical operators to handle errors inline

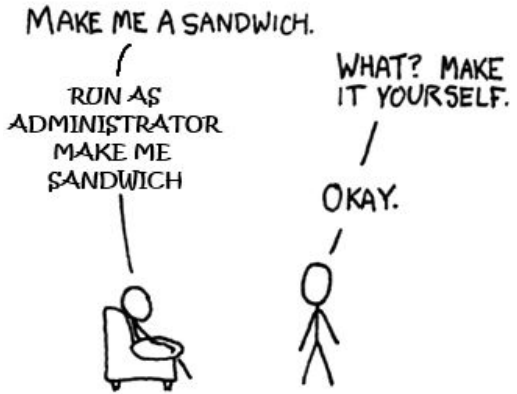
```
mkdir /mydir && echo "Directory created." || echo "Failed to create directory."

mkdir: /mydir: Read-only file system
Failed to create directory.
```

Why this example fails? 😉

The / at the beginning of the path specifies an **absolute path** .

An absolute path starts from the **root directory** (/)
By default, we don't have permission to create a directory in the root directory (/), which is reserved for system files and directories.

It is possible by using **`sudo mkdir /mydir`**



05

Special variables

Hard to read by humans but very powerful

$#: Number of Arguments

Represents the number of positional parameters (arguments) passed to a script or function.

Use Case: To check if the required number of arguments is provided.

```bash
#!/bin/bash
if [ "$#" -lt 2 ]; then
    echo "Usage: $0 arg1 arg2" >&2
    exit 1
fi
echo "Number of arguments: $#"
```

```
Usage: 06_special_variables.sh arg1 arg2
```

$@ and $*: All Arguments

$@ Expands to all arguments as separate strings.

$* Expands to all arguments as a single string.

$ 06_all_arguments.sh

```
1   #!/bin/bash
2   echo "Using \$@:"
3   for arg in "$@"; do
4       echo "$arg"
5   done
6
7   echo "\n"
8
9   echo "Using \$*:"
10  for arg in "$*"; do
11      echo "$arg"
12  done
```

```
Using $@:
user,
password,
location,
e-mail


Using $*:
user, password, location, e-mail
```

Bash can make you cry ;)

$ 06_example.sh

```bash
#!/bin/bash

# Check if at least one argument is provided
[ "$#" -eq 0 ] && echo "Usage: $0 dir1 [dir2 ...]" && exit 1

log_file="$1/process_$$_$!.log"

# Log the current script name and number of arguments
echo "Running $0 with $# arguments" > "$log_file"

for d in "$@"; do
    if [ -d "$d" ]; then
        # Count files in the directory
        count=$(ls -1 "$d" 2>/dev/null | wc -l)
        echo "Directory $d: $count files" >> "$log_file"
    else
        echo "$d is not a directory!" >> "$log_file"
    fi
done

echo "Log file: $log_file"
```

Checks if arguments are provided : Uses $# to check if arguments are passed. If the condition evaluates to true (i.e., no arguments were passed), it executes the next command after the &&.

Creates a unique log file : Uses $$ (script PID) and $! (last background PID, even though it's redundant here) to create a uniquely named log file.

Processes each argument :

- Uses $@ to loop through all arguments.
- Checks if an argument is a directory (-d).
- Counts files in the directory using ls and wc -l.

Logs results : Writes the results to a log file in the directory passed as first argument

Overview

Learn more at: https://linuxhandbook.com/bash-special-variables/

Special Variable	Description
`$0`	Gets the name of the current script.
`$#`	Gets the number of arguments passed while executing the bash script.
`$*`	Gives you a string containing every command-line argument.
`$@`	It stores the list of every command-line argument as an array.
`$1-$9`	Stores the first 9 arguments.
`$?`	Gets the status of the last command or the most recently executed process.
`$!`	Shows the process ID of the last background command.
`$$`	Gets the process ID of the current shell.
`$-`	It will print the current set of options in your current shell.

06

Cool Bash Scripts

Few examples to boost your fantasy using Bash

File Backup

Creates a timestamped backup of a directory using tar compression.

$ 05_file_backup.sh

```bash
#!/bin/bash
source_dir=$1
backup_dir=$2
timestamp=$(date +%Y%m%d%H%M%S)

# Create the backup directory if it doesn't exist
mkdir -p "$backup_dir"

# Define the backup file name
backup_file="backup_$timestamp.tar.gz"

# Create the tarball (backup) file
tar -czvf "$backup_dir/$backup_file" "$source_dir"
```

```
sh 05_file_backup.sh . my_backup
```

Backups current directory . to the my_backup directory

Rename files with pattern

Creates a timestamped backup of a directory using tar compression.

```
18  # Process all files in the directory
19  for file in "$dir"/*; do
20      # Skip if not a file
21      [ -f "$file" ] || continue
22
23      # Get the file name without the path
24      base_name=$(basename "$file")
25
26      # Remove IMG or PANO at the beginning of the file name
27      new_name=$(echo "$base_name" | sed -E 's/^(IMG|PANO)//')
28
29      # Skip renaming if the name remains the same
30      if [ "$base_name" = "$new_name" ]; then
31          continue
32      fi
33
34      # Rename the file
35      mv "$file" "$dir/$new_name"
36      echo "Renamed: $base_name -> $new_name"
37  done
```

Loops over all files in the directory ("$dir"/*).
Skips non-files using [-f "$file"].

Prefix Removal: Extracts the file name using basename.
Uses sed with a **regular expression to remove IMG or PANO** at the start of the file name (s/^(IMG|PANO)//).

Renaming: Compares the original and modified names to avoid unnecessary renaming.
Uses mv to rename the file and keeps the directory path intact.

Bash for Cloud Development: AWS

```
3   # Check if AWS CLI is installed
4   if ! command -v aws &> /dev/null; then
5       echo "AWS CLI is not installed."
6       exit 1
7   fi
8
9   # List all S3 buckets
10  echo "Fetching list of S3 buckets..."
11  aws s3 ls
```

Pre-checks:

Ensures AWS CLI is installed.

Lists all available S3 buckets for reference.

Input Prompts:

Prompts the user for the S3 bucket name and the file path.

File Upload:

Uses the `aws s3 cp` command to upload the specified file to the bucket.

Confirms if the file exists in the bucket after upload using `aws s3 ls`.

Error Handling:

Checks if the file path exists before uploading.

Validates the success of the upload.

```
26  # Upload the file to the specified S3 bucket
27  echo "Uploading $file_name to bucket $bucket_name..."
28  if aws s3 cp "$file_path" "s3://$bucket_name/$file_name"; then
29      echo "Upload successful!"
```

Conclusion

Bash is an incredibly powerful tool for developers, enabling automation, data processing, system management, and cloud integration.

Today we explored key concepts and practical examples:

Special Variables: provide insights into the script's execution and environment, enabling dynamic and efficient scripting.

Error Handling : Bash scripts can handle errors, clean up resources, and ensure robust execution.

Pipes and Redirection: Tools to allow seamless chaining and redirection of commands for filtering, aggregating, or transforming data.

File Management: Scripts can automate file renaming, backups, and more, saving time and avoiding manual errors.

Deloitte.

Thank you for your attention

Alexandra Baga
CTSE Consultant
E-Mail: oleksandra.baga@gmail.com
https://github.com/oleksa-oleksa