Bash Scripting Handout

by [TechWorld with Nana]

Accompanying handout for our tutorial "Bash Scripting" on [TWN Youtube channel]

TECHWORLD WITH NANA

Table of Contents

Ready? Let's do this! 👋

Introduction

💡 What You'll Learn

- ◯ Understand what shell scripting is and why it's powerful
- ◯ Set up a Bash environment on any operating system
- ◯ Write shell scripts that automate repetitive tasks
- ◯ Use variables, loops, and conditionals in your scripts
- ◯ Create practical automation scripts for real-world scenarios

🚀 Real-World Impact

Shell scripting can transform tasks that take 30-45 minutes of manual work into automated processes that run in seconds.

This handout will show you exactly how to build that automation.

Who created this?

I'm Nana, Co-Founder of TechWorld with Nana.

As an engineer and trainer, I'm dedicated to helping engineers build the most valuable and highly-demanded DevOps and Cloud skills.

Through my **YouTube channel** and my comprehensive **DevOps bootcamps**, **I've helped millions of engineers** master the tools and concepts that drive modern software development.

Happy learning!

Nana

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Understanding Shell, Bash & Terminal

Before we start writing scripts, let's decode the techy words you'll hear everywhere. Don't worry - they're simpler than they sound!

Think of your computer like a house with two ways to control everything inside:

Graphical User Interface (GUI)

- The visual way to use your computer - clicking icons, dragging files, using your mouse
- What most people use daily
- Limited for repetitive tasks

Command Line Interface (CLI)

- Text-based interface using typed commands
- Much more powerful than GUI when you know what you're doing
- Example: Create 100 folders with one command vs. clicking 100 times

More key concepts you should know:

Shell

- Program that runs and interprets CLI commands
- Acts as intermediary between you and the operating system
- Multiple "flavors" exist (bash, zsh, sh, etc.)

Bash (Bourne Again Shell)

- Most common shell implementation for Linux systems
- Full programming language, not just command runner
- What we'll be learning here

Terminal

- Graphical window where you type commands
- The "container" that displays the shell interface
- Like a window frame around your CLI workspace

Why Shell Scripting?

✅ **Automation**: Turn repetitive tasks into one-command solutions

✅ **Consistency**: Same steps executed exactly the same way every time

✅ **Speed**: Complete complex tasks in seconds

✅ **Documentation**: Scripts serve as living documentation of processes

✅ **Error Reduction**: Eliminate human mistakes in repetitive tasks

Installation Instructions

For Mac Users - Use Built-in Bash

For macOS Catalina (10.15) and newer:

- The default shell was changed from Bash to Zsh
- Bash is still installed (version 3.2), just not the default
- To use Bash, simply open Terminal and type:

```
bash
```

To make Bash your default shell again:

```
chsh -s /bin/bash
```

For Windows Users

Option 1: Windows Subsystem for Linux (WSL) - Recommended

Open PowerShell as Administrator and run:

```
wsl --install
```

- This installs Ubuntu by default, which includes Bash
- After installation, restart your computer
- Open the Ubuntu application from your Start menu to access Bash

Option 2: Git Bash (Simpler alternative if you already use Git)

Download Git for Windows from: https://git-scm.com/download/win

For Linux Users

You're already set up! Most Linux distributions come with Bash pre-installed.

```
bash --version
```

After setup, verify everything works:

```
echo "Hello, Bash scripting!"
which bash
bash --version
```

Sample Log Files

Create a logs directory and add sample files to follow along with the tutorial.

The sample files contain detailed log entries with timestamps, severity levels, and various system and application messages that we'll analyze in our scripts.

Directory Setup

```
mkdir logs
cd logs
```

Downloading Sample Files

Download the sample files provided.

Or create the files like this:

```
touch application.log
touch system.log
```

After creating the files, you can update their timestamps:

```
touch logs/application.log
touch logs/system.log
```

These sample log files contain various types of messages including INFO, WARNING, ERROR, CRITICAL, and FATAL entries that we'll use throughout our scripting exercises.

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Basic Linux Commands

Viewing Files

```
cat /path/to/logfile.log
less /path/to/logfile.log
head /path/to/logfile.log
tail /path/to/logfile.log
```

Searching Content

```
grep "ERROR" /path/to/logfile.log
grep -c "ERROR" /path/to/logfile.log
grep -i "error" /path/to/logfile.log
grep -n "ERROR" /path/to/logfile.log
```

Finding Files

```
find . -name "*.log"
find /path/to/logs -name "*.log" -mtime -1
find /path/to/logs -name "*.log" -size +100M
find /path/to/logs -name "*.log" -exec grep -l "ERROR" {} \;
```

Revisit the most important Linux commands?

Watch tutorial here: [13 Linux Commands Every Engineer Should Know]

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Basic Linux Commands

MANUAL Log Analysis Workflow

Here's what you would typically do manually (this is what we'll automate):

```
# Step 1: Find recent log files
find /Users/nana/logs -name "*.log" -mtime -1

# Step 2: Search for errors in application log
grep "ERROR" /Users/nana/logs/application.log

# Step 3: Count errors
grep -c "ERROR" /Users/nana/logs/application.log

# Step 4: Get most recent error
grep "ERROR" /Users/nana/logs/application.log | tail -1

# Step 5: Check for fatal errors in system log
grep "FATAL" /Users/nana/logs/system.log

# Step 6: Count fatal errors
grep -c "FATAL" /Users/nana/logs/system.log
```

Problems with Manual Approach:

❌ Repetitive and time-consuming

❌ Prone to human error

❌ No consistent reporting format

❌ Easy to miss steps when interrupted

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Creating Your First Script

Let's start with the basics. A shell script is simply a text file with Linux commands.

First, let's create a new file called `analyse_logs.sh`:

```
touch analyse_logs.sh
```

Now let's open it with a text editor:

```
vim analyse_logs.sh
```

Converting manual commands to Shell Script

First, let's see the basic Linux commands you would run manually to search for errors in log files:

```
# Find all log files from yesterday
find /Users/nana/logs -name "*.log" -mtime -1

# Search for ERROR in a log file
grep "ERROR" /Users/nana/logs/application.log

# Count how many errors are in the file
grep -c "ERROR" /Users/nana/logs/application.log

# Find the most recent error
grep "ERROR" /Users/nana/logs/application.log | tail -1

# Look for FATAL errors in another log file
grep "FATAL" /Users/nana/logs/system.log

# Count FATAL errors
grep -c "FATAL" /Users/nana/logs/system.log
```

The lines starting with `#` are comments - they're ignored when the script runs, but they help us humans understand what the code does. To make script easily readable for us!

Now how to execute - make executable? - other OS programs like python, vim, mkdir - these are all programs, also called executables, cuz like our custom script, they are list of commands that someone wrote that can be run/executed to run commands on our OS!

```
chmod +x analyse-logs.sh
./analyse-logs.sh
```

The `./` tells the shell to look for the script in the current directory. You'll see our message printed to the screen with today's date and your username.

Creating Your First Script

Why do we have `sh` extension at the end?

- The extension helps humans quickly identify the file as a shell script
- Makes it easier to recognize script files in a directory listing
- Provides a visual cue about the file's purpose and execution method
- Text editors and IDEs use file extensions to enable appropriate syntax highlighting
- Consistent with other scripting languages (`.py`, `.rb`, `.js`, etc)
- Extensions Are Not Required: Unix/Linux systems execute files based on permissions and content, not extensions. A shell script will run perfectly fine with no extension if it has executable permissions.

> ⓘ So a script named `script.sh` could actually be a Bash, Zsh, Ksh, or POSIX shell script. The extension itself doesn't dictate which shell interprets the script.

Now, in that case, how does interpreter know which shell program the script is written in. That's where what's called a shebang statement comes in.

The `#!` (shebang) line at the beginning of the script determines which interpreter runs it:

- `#!/bin/bash` for Bash scripts
- `#!/bin/sh` for POSIX shell scripts

So let's add a line at start that tells the system to use the Bash interpreter for this script.

```
#!/bin/bash
```

So a shell script is simply a text file with commands, but with a special first line called a 'shebang' that tells the system this is a script and which interpreter to use.

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Creating Your First Script

Adding `echo` command

Now we made script code easy to read, but the output is not easy to read. The lines aren't separated, so it's hard to see where each command execution and output starts and ends. Instead, we wanna see the output with information about what the output means and separate each command output from each other, so let's print out this information with echo command.

Adding `line breaks` and visual separators

Now let's add line breaks and more visual separators to make reading output easier.

- IMPORTANT: `-e` flag, so that `echo` interprets escape sequences like `\\n` for newlines.

When you use the `-e` flag with `echo`, it tells the command to interpret backslash escape sequences in the string instead of treating them as literal characters.

Without the `-e` flag, `echo` will print the exact characters you provide, including the backslash and the letter following it

Here's how your script should look like now:

```
#!/bin/bash

echo "### analysing logs"
echo "=============="

echo -e "\n### List of log files updated yersterday ###"
find /Users/nana/logs -name "*.log" -mtime -1

echo -e "\n### searching ERROR logs in application logs ###"
grep "ERROR" /Users/nana/logs/application.log

echo -e "\n### Number of errors found in application logs ###"
grep -c "ERROR" /Users/nana/logs/application.log

echo -e "\n### The last error from application logs ###"
grep "ERROR" /Users/nana/logs/application.log | tail -1

echo -e "\n### The FATAL errors from system logs ###"
grep "FATAL" /Users/nana/logs/system.log

echo -e "\n### Number of FATAL errors in system logs ###"
grep -c "FATAL" /Users/nana/logs/system.log
```

Variables

Variables eliminate repetition and make scripts more maintainable.

For example we are repeating the file names and log directory locations.

What if we change the directory to “/var/log”, we would have to re-write each line.

So that’s where variables come in. We will **store repeated values in variables, and reference them.**

```
#!/bin/bash

# Define variables at the top
LOG_DIR="/Users/nana/logs"
APP_LOG_FILE="application.log"
SYS_LOG_FILE="system.log"

echo "### analysing logs ###"
echo "======================"
echo -e "\n### List of log files updated yesterday ###"
find "$LOG_DIR" -name "*.log" -mtime -1

# Rest of script using variables
```

Variable Best Practices

- Use descriptive names (LOG_DIR, not LD)
- Use ALL_CAPS for constants
- Always quote variables: `"$VARIABLE"`
- Define variables at the top of the script

Array Variables - Arrays can hold multiple values, each accessible by an index

```
ERROR_PATTERNS=("ERROR" "FATAL" "CRITICAL")
```

- Create: `ARRAY=("item1" "item2" "item3")`
- Access single element: `${ARRAY[0]}` (first element)
- Access all elements: `${ARRAY[@]}`
- Get array length: `${#ARRAY[@]}`

Command Substitution - Capture command output in variables

```
LOG_FILES=$(find "$LOG_DIR" -name "*.log" -mtime -1)
```

- `VARIABLE=$(command)` - Modern preferred syntax
- `VARIABLE=`command`` - Older syntax (still works)

Loops & Iteration

Why We Need Loops

Let's say we want to expand our script to analyze all log files in the directory - maybe logs from different services are collected here. For each log file, we want to check multiple error patterns like **ERROR**, **CRITICAL**, and **FATAL**.

If we have 10 different log files, we could write the same analysis logic 10 times for each file. But that would mean manually checking the logs folder, copying all the file names, and pasting them in our script. That's definitely not feasible!

Instead, **we want dynamic logic that goes through any number of log files, one by one, and executes the same analysis for each**. That's where loops come in - we loop through (iterate through) files and execute the same logic for each iteration.

Loops eliminate repetitive code and make scripts dynamic.

For Loops with Files

```
#!/bin/bash

LOG_DIR="/Users/nana/logs"
ERROR_PATTERNS=("ERROR" "FATAL" "CRITICAL")

echo -e "\n============================================================"
echo "==== Analysing log files in $LOG_DIR directory ===="
echo "============================================================"

echo -e "\n### List of log files updated in last 24 hours ###"
LOG_FILES=$(find "$LOG_DIR" -name "*.log" -mtime -1)
echo "$LOG_FILES"

# Loop through each log file
for LOG_FILE in $LOG_FILES; do
    echo -e "\n================================================="
    echo "============ $LOG_FILE ============"
    echo "================================================="

    echo -e "\n### searching ${ERROR_PATTERNS[0]} logs in $LOG_FILE ###"
    grep "${ERROR_PATTERNS[0]}" "$LOG_FILE"

    echo -e "\n### Number of ${ERROR_PATTERNS[0]} logs found ###"
    grep -c "${ERROR_PATTERNS[0]}" "$LOG_FILE"

    # Additional commands omitted for brevity
done
```

Loops & Iteration

Nested Loops (Loop within Loop)

Outer Loop: Iterate Through Log Files

```
for LOG_FILE in $LOG_FILES; do
    # Process each log file
done
```

Inner Loop: Iterate Through Error Patterns

```
for PATTERN in "${ERROR_PATTERNS[@]}"; do
    # Process each error pattern
done
```

Loop Benefits

- Automatically handles any number of files
- Easily add new error patterns to the array
- Consistent processing for each item
- Much less code than manual repetition

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

File Operations

Writing Output to Files

Instead of printing to console, save results to a report file:

```
# Write to file (overwrites existing content)
echo -e "\n" > "$REPORT_FILE"

# Append output to file
echo "==== Analysing log files in $LOG_DIR directory ====" >> "$REPORT_FILE"
```

File Redirection:

- `>` - Redirect output to file (overwrites)
- `>>` - Append output to file
- First use `>` to create/clear file, then use `>>` for subsequent writes

Conditionals

Conditionals allow your script to make decisions and respond differently based on specific conditions. Think of them as "if this, then that" logic.

Why We Need Conditionals: In our log analysis script, we don't just want to count errors - we want to be alerted when there are too many errors that need immediate attention. For example, if we find more than 10 ERROR entries in a log file, that might indicate a serious problem requiring action.

Without conditionals, our script would just report numbers. With conditionals, our script becomes intelligent - it can distinguish between normal situations and problems that need our attention.

Conditional Syntax:

- `if [ condition ]; then` - Start conditional
- `fi` - End conditional
- `-gt` - Greater than (numeric comparison)
- `-eq` - Equal to
- `-lt` - Less than
- `-ne` - Not equal to

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Best Practices Summary

1. Script Organization

- ◯ Always use shebang line: `#!/bin/bash`
- ◯ Define variables at the top
- ◯ Use functions for repeated code
- ◯ Add comments for complex logic
- ◯ Use consistent indentation

2. Error Handling

- ◯ Check if required files/directories exist
- ◯ Validate input parameters
- ◯ Use appropriate exit codes
- ◯ Handle edge cases gracefully

3. Security

- ◯ Quote all variables: "$VARIABLE"
- ◯ Avoid hardcoded passwords
- ◯ Use absolute paths when possible
- ◯ Validate user input

4. Maintainability

- ◯ Use descriptive variable names
- ◯ Break long scripts into functions
- ◯ Document complex operations
- ◯ Version control your scripts

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Real World Use Cases for Practice 

ⓘ Each of these tasks **saves significant time** by automating repetitive processes, improving efficiency, and **reducing the likelihood of errors**.

By practicing these use cases, you'll build a stronger foundation in shell scripting and increase your ability to handle real-world automation challenges.

1. Local Development Environment Setup

📃 Write a script that quickly sets up a developer's local machine with all needed tools, configurations, and test data.

With one command, it installs required software versions, configures environment variables, clones relevant repositories, and creates test databases.

✅ This saves hours of manual setup and ensures all developers have consistent environments

2. Automating Log File Rotation

Log files can grow large quickly, consuming valuable disk space.

📃 Write a script to rotate logs—archive old logs, delete the oldest logs, and keep only the most recent ones.

Sample task: Create a shell script to rotate logs and delete logs older than 30 days.

3. Automating Backup Process

📃 Write a script that compresses important files and directories, uploads them to a remote server or cloud storage, and sends a notification once the backup is complete.

Sample Task: Create a script that backs up your home directory and uploads the archive to AWS S3.

4. Automating Server Health Checks

📃 Write a script to monitor the health of your server, check for disk space, CPU load, memory usage, and uptime.

The script can then alert the admin if any parameter crosses a threshold.

Sample Task: Create a script that checks CPU and memory usage and alerts if they exceed a certain threshold.

5. Monitoring Disk Space Usage

📃 Write a script that checks the disk space usage of your system and alerts you if the disk usage exceeds a specified threshold.

The script can then alert the admin if any parameter crosses a threshold.

Sample Task: Create a script that sends an alert when disk usage exceeds 90%

Learning Resources

Next Steps

- **Learn more Bash features**: functions, case statements, regex
- **Explore automation tools**: cron jobs, systemd timers
- **Study configuration management**: Ansible, Puppet, Chef
- **Practice with real scenarios**: monitoring, deployments, backups

Useful Commands to Explore

```
# Advanced text processing
awk, sed, cut, sort, uniq

# System monitoring
ps, top, htop, netstat, ss

# Network operations
curl, wget, ssh, scp

# File operations
rsync, tar, zip, find
```

Online Resources

- Bash manual: `man bash`
- ShellCheck (script linting): https://shellcheck.net
- Bash scripting guide: https://tldp.org/LDP/Bash-Beginners-Guide/html/
- Practice platform: https://cmdchallenge.com

© nnSoftware GmbH - For more DevOps resources and tutorials, visit techworld-with-nana.com and YouTube channel

Good luck on your journey! 🙌

🟢 **You now have the foundation to:**

- Automate repetitive tasks with shell scripts
- Use variables, loops, and conditionals effectively
- Create professional automation solutions
- Continue learning advanced scripting techniques

🟢 **Remember**:

Start small, practice regularly, and gradually build more complex scripts. Every automated task saves time and reduces errors!

🟢 **Share your knowledge:**

If this helped you, share it with a colleague who could benefit from learning shell scripting.

ⓘ This handout accompanies the "Bash Scripting" YouTube tutorial.