

CSS 262: Linux Administration

Bash Scripting & Automation

Lecture 5: Automate All the Things



Today's Agenda

Part 1: Scripting Fundamentals

- What is a shell script?
- Shebang, execution, and permissions
- Variables & data types
- User input & output

Part 3: Text Processing & Pipelines


- `grep` , `sed` , `awk` , `cut`
- Regular expressions
- Pipelines & redirection

Part 2: Control Structures & Functions

- Conditionals (`if` , `case`)
- Loops (`for` , `while` , `until`)
- Functions & scope
- Arrays & string operations

Part 4: Automation & Best Practices

- Cron jobs & scheduling
- Real-world scripts
- Error handling & debugging
- Security considerations

 **Learning Objective:** Develop automated solutions for system maintenance and log processing using Bash scripting.



Quick Recap: Week 4

Storage, Filesystems & LVM

- **Storage Hierarchy:** Physical disks → Partitions → Filesystems
- **Partition Tables:** MBR (legacy) vs GPT (modern standard)
- **Filesystems:** ext4 (general), XFS (performance), Btrfs (snapshots)
- **Mounting:** Manual `mount` and persistent `/etc/fstab`
- **LVM:** Flexible storage with PV → VG → LV architecture
- **Snapshots:** Point-in-time copies for consistent backups

✓ You should now be comfortable managing storage, filesystems, and LVM volumes!

Part 1: Scripting Fundamentals

Your First Bash Scripts



What is a Shell Script?

A **shell script** is a text file containing a sequence of commands that the shell executes.

Why Script?

- **Repeatability:** Run the same tasks consistently
- **Automation:** Schedule tasks to run unattended
- **Efficiency:** Combine many commands into one action
- **Documentation:** Scripts document your procedures
- **Error Reduction:** Eliminate human mistakes

The Bash Shell

- **Bash** = Bourne Again Shell (default on most Linux distros)
- Superset of the original Bourne shell (`sh`)
- Available at `/bin/bash`
- Check your shell: `echo $SHELL`



Anatomy of a Script

```
1  #!/bin/bash
2  # backup.sh - Simple backup script
3  # Author: sysadmin | Date: 2025-01-15
4
5  set -euo pipefail
6
7  BACKUP_DIR="/backup"
8  SOURCE_DIR="/var/www"
9  DATE=$(date +%Y%m%d_%H%M%S)
10 ARCHIVE="${BACKUP_DIR}/www_${DATE}.tar.gz"
11
12 mkdir -p "$BACKUP_DIR"
13 tar czf "$ARCHIVE" "$SOURCE_DIR"
14
15 echo "Backup complete: $ARCHIVE"
```

Line 1: Shebang (`#!/bin/bash`) – tells the kernel which interpreter to use

Line 5: `set -euo pipefail` – strict mode for safer scripts

Lines 7-9: Variables store configuration – easy to modify

Running Scripts

Method 1: Make it Executable

```
1  chmod +x backup.sh
2  ./backup.sh
```

Method 2: Invoke the Interpreter

```
1  bash backup.sh
```

Method 3: Source (runs in current shell)

```
1  source backup.sh
2  # or
3  . backup.sh
```

⚠ **Key Difference:** `./script.sh` runs in a subshell (isolated), while `source script.sh` runs in the current shell (can modify your environment).



Variables

Assigning Variables (no spaces around =)

```
1  NAME="Linux"           # String
2  COUNT=42               # Integer (still stored as string)
3  FILES=$(ls /tmp)       # Command substitution
4  TODAY=$(date +%F)      # Command substitution
```

Using Variables

```
1  echo "Welcome to $NAME"
2  echo "File count: ${COUNT}"      # Braces for clarity
3  echo "Config: ${NAME}_config"     # Braces to delimit name
```

Environment vs Local Variables

```
1  MY_VAR="local only"      # Local to current shell
2  export MY_VAR="shared"   # Available to child processes
3  env                      # Show all environment variables
```


12
34

Special Variables

Variable	Description
<code>\$0</code>	Script name
<code>\$1</code> - <code>\$9</code>	Positional arguments
<code>\$#</code>	Number of arguments
<code>\$@</code>	All args (preserves quoting)
<code>\$*</code>	All args (single string)
<code>\$?</code>	Exit status of last command
<code>\$\$</code>	Current process PID
<code>\$_</code>	PID of last background process

Example

```
1  #!/bin/bash
2  echo "Script: $0"
3  echo "First arg: $1"
4  echo "All args ($#): $@"
5  echo "Last exit code: $?"
```

Input & Output

Reading User Input

```
1 read -p "Enter your name: " USERNAME
2 echo "Hello, $USERNAME!"
3 read -sp "Password: " PASS      # -s = silent (no echo)
4 echo                          # newline after hidden input
5 read -t 10 -p "Quick! " ANSWER # -t = timeout (seconds)
```

Output Commands

```
1 echo "Simple output"
2 echo -e "Tabs:\tand\nnewlines"    # -e enables escape sequences
3
4 printf "%-20s %5d\n" "Users:" 42  # Formatted output
5 printf "%-20s %5d\n" "Groups:" 7
```

Redirecting Output

```
1 echo "log entry" >> /var/log/app.log # Append
2 echo "fresh start" > /var/log/app.log # Overwrite
3 command 2>/dev/null                 # Discard errors
```



Arithmetic

Arithmetic Expansion `$(())`

```
1  A=10
2  B=3
3  echo "Sum: $((A + B))"          # 13
4  echo "Diff: $((A - B))"         # 7
5  echo "Product: $((A * B))"      # 30
6  echo "Division: $((A / B))"     # 3 (integer only!)
7  echo "Modulo: $((A % B))"       # 1
8  echo "Power: $((A ** B))"       # 1000
```

Increment / Decrement

```
1  ((COUNT++))                   # Increment
2  ((COUNT--))                   # Decrement
3  ((COUNT += 5))                 # Add 5
```

Floating Point (use `bc`)

```
1  RESULT=$(echo "scale=2; 10 / 3" | bc)
2  echo "$RESULT"                 # 3.33
```

Part 2: Control Structures & Functions

Making Decisions and Repeating Tasks



Conditionals: `if` / `elif` / `else`

Basic Syntax

```
1  if [[ condition ]]; then
2      # commands
3  elif [[ condition ]]; then
4      # commands
5  else
6      # commands
7  fi
```

Example: Check Disk Usage

```
1  USAGE=$(df / --output=pcent | tail -1 | tr -d '% ')
2
3  if [[ "$USAGE" -gt 90 ]]; then
4      echo "CRITICAL: Disk usage at ${USAGE}%!"
5  elif [[ "$USAGE" -gt 75 ]]; then
6      echo "WARNING: Disk usage at ${USAGE}%"
7  else
8      echo "OK: Disk usage at ${USAGE}%"
9  fi
```



Test Operators

Numeric Comparisons

Operator	Meaning
<code>-eq</code>	Equal
<code>-ne</code>	Not equal
<code>-gt</code>	Greater than
<code>-ge</code>	Greater or equal
<code>-lt</code>	Less than
<code>-le</code>	Less or equal

String Comparisons

Operator	Meaning
<code>=</code>	Equal
<code>≠</code>	Not equal
<code>-z</code>	Empty string
<code>-n</code>	Non-empty string
<code>< / ></code>	Lexicographic

Logical Operators

Operator	Meaning
<code>&&</code>	AND
<code> </code>	OR
<code>!</code>	NOT



Test Operators: File Tests

File Tests

Operator	Meaning
-e	File exists
-f	Is regular file
-d	Is directory
-r	Is readable
-w	Is writable
-x	Is executable
-s	File is non-empty
-L	Is symbolic link

Example

```
1  if [[ -f "/etc/passwd" ]]; then
2      echo "File exists"
3  fi
4
5  if [[ -d "/tmp" && -w "/tmp" ]]; then
6      echo "Dir is writable"
7  fi
```

The case Statement

Syntax

```
1 case "$variable" in
2     pattern1)  commands ;;
3     pattern2)  commands ;;
4     *)         default commands ;;
5 esac
```

Example: Service Control

```
1 #!/bin/bash
2 case "$1" in
3     start)    systemctl start myapp ;;
4     stop)     systemctl stop myapp ;;
5     restart)  systemctl restart myapp ;;
6     status)   systemctl status myapp ;;
7     *)        echo "Usage: $0 start|stop|restart|status"; exit 1 ;;
8 esac
```




Loops: for

C-Style For Loop

```
1  for ((i = 1; i ≤ 5; i++)); do
2      echo "Iteration $i"
3  done
```

Iterate Over a List

```
1  for USER in alice bob charlie; do
2      echo "Creating user: $USER"
3      useradd "$USER"
4  done
```

Iterate Over Files

```
1  for FILE in /var/log/*.log; do
2      SIZE=$(du -sh "$FILE" | cut -f1)
3      echo "$FILE: $SIZE"
4  done
```

Range

```
1  for i in {1..10}; do
2      echo "Server-$i"
3  done
```



Loops: `while` & `until`

`while` – Loop While Condition is True

```
1  COUNT=0
2  while [[ $COUNT -lt 5 ]]; do
3      echo "Count: $COUNT"
4      ((COUNT++))
5  done
```

Reading a File Line by Line

```
1  while IFS= read -r LINE; do
2      echo "Processing: $LINE"
3  done < /etc/passwd
```

`until` – Loop Until Condition is True

```
1  until ping -c1 -W2 google.com &>/dev/null; do
2      echo "Waiting for network..."
3      sleep 5
4  done
5  echo "Network is up!"
```

Functions

Defining Functions

```
1 greet() {
2     local NAME="$1"
3     echo "Hello, $NAME!"
4 }
5 greet "World"
```

Return Values

```
1 is_root() {
2     [[ $(id -u) -eq 0 ]]
3 }
4 if is_root; then
5     echo "Running as root"
6 else
7     echo "Not root"; exit 1
8 fi
```

Returning Data (stdout capture)

```
1 get_ip() {
2     hostname -I | awk '{print $1}'
3 }
4 MY_IP=$(get_ip)
5 echo "Server IP: $MY_IP"
```

💡 Always use `local` for function variables to avoid polluting the global scope.

Arrays

Indexed Arrays

```
1  SERVERS=("web01" "web02" "db01")
2
3  echo "${SERVERS[0]}"           # web01
4  echo "${SERVERS[@]}"           # All elements
5  echo "${#SERVERS[@]}"           # Length: 3
6
7  SERVERS+=("monitor01")         # Append
8  unset SERVERS[2]               # Remove index 2
```

Looping Over Arrays

```
1  for SERVER in "${SERVERS[@]"; do
2      ping -c1 -W2 "$SERVER" &>/dev/null \
3      && echo "$SERVER: UP" \
4      || echo "$SERVER: DOWN"
5  done
```

Associative Arrays (Bash 4+)

```
1  declare -A PORTS
2  PORTS[http]=80
3  PORTS[https]=443
4  PORTS[ssh]=22
5
6  for SERVICE in "${!PORTS[@]"; do
7      echo "$SERVICE → ${PORTS[$SERVICE]}"
8  done
```

Part 3: Text Processing & Pipelines

Parsing Logs and Transforming Data



grep – Search Text

Basic Usage

```
1  grep "error" /var/log/syslog          # Find lines with "error"
2  grep -i "error" /var/log/syslog      # Case-insensitive
3  grep -n "error" /var/log/syslog      # Show line numbers
4  grep -c "error" /var/log/syslog      # Count matches
5  grep -r "TODO" /home/dev/project/    # Recursive search
```

Regular Expressions

```
1  grep -E "^root:" /etc/passwd         # Lines starting with "root:"
2  grep -E "failed|error" /var/log/auth.log # Match either pattern
3  grep -E "[0-9]{1,3}\.[0-9]{1,3}" access.log # IP-like patterns
4  grep -v "^#" /etc/ssh/sshd_config    # Exclude comments
```

Practical: Failed SSH Logins

```
1  grep "Failed password" /var/log/auth.log | tail -20
2  grep "Failed password" /var/log/auth.log | \
3  grep -oE "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+" | sort | uniq -c | sort -rn
```



cut, sort, uniq – Slice & Dice

cut – Extract Fields

```
1 cut -d: -f1 /etc/passwd          # Usernames (field 1, : delimited)
2 cut -d: -f1,3 /etc/passwd        # Username and UID
```

sort – Order Lines

```
1 sort /etc/passwd                  # Alphabetical sort
2 sort -t: -k3 -n /etc/passwd      # Sort by UID (numeric)
3 du -sh /var/log/* | sort -rh      # Sort by human-readable sizes
```

uniq – Remove Duplicates (requires sorted input)

```
1 sort access.log | uniq -c         # Count occurrences
2 sort access.log | uniq -c | sort -rn # Top occurrences
```

Combined Pipeline

```
1 awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -10
```



sed – Stream Editor

Stream editor: transform text line-by-line (search/replace, print, delete).

Search & Replace

```
1 sed 's/old/new/' file.txt      # First match per line
2 sed 's/old/new/g' file.txt     # All matches (global)
3 sed -i 's/old/new/g' file.txt  # In-place edit
4 sed -i.bak 's/old/new/g' file  # In-place with .bak backup
```

Line Operations

```
1 sed -n '5,10p' file.txt        # Print lines 5–10 only
2 sed '3d' file.txt              # Delete line 3
3 sed '/^#/d' config.txt         # Delete lines starting with #
4 sed '/^$/d' file.txt           # Delete empty lines
```

Practical: Config Editing

```
1 sed -i 's/^#Port 22/Port 2222/' /etc/ssh/sshd_config
2 sed '/^#/d; /^$/d' /etc/ssh/sshd_config # Strip comments and blanks
```




awk – Pattern Processing

Basic Structure: `awk 'pattern { action }' file`

```
1  awk '{print $1}' access.log           # Print first field
2  awk '{print $1, $7}' access.log       # Print IP and URL
3  awk -F: '{print $1, $3}' /etc/passwd  # Custom delimiter
```

Filtering

```
1  awk '$3 > 1000' /etc/passwd           # UID > 1000
2  awk '/error/ {print $0}' /var/log/syslog # Lines matching "error"
3  awk 'NR ≥ 10 && NR ≤ 20' file.txt      # Lines 10-20
```

Built-in Variables & Computation

```
1  # Count lines
2  awk 'END {print NR}' file.txt
3
4  # Sum a column (e.g., bytes transferred)
5  awk '{sum += $10} END {print sum}' access.log
6
7  # Average response time
8  awk '{sum += $NF; n++} END {print sum/n}' access.log
```

Pipelines & Redirection

The Pipe (|)

```
1  grep "error" /var/log/syslog | wc -l
```

File Descriptors

FD	Name	Default
0	stdin	Keyboard
1	stdout	Terminal
2	stderr	Terminal

Redirection

```
1  command > file           # stdout (overwrite)
2  command >> file          # stdout (append)
3  command 2> file          # stderr to file
4  command &> file           # Both stdout & stderr
5  command 2>&1              # stderr to stdout
6  command < file            # File as stdin
```

Process Substitution

```
1  diff <(ls dir1) <(ls dir2)
```



Practical: Log Analysis Pipeline

Analyze Apache/Nginx Access Log

```
1  #!/bin/bash
2  # log_report.sh - Generate access log report
3  LOG="${1:-/var/log/nginx/access.log}"
4
5  echo "=== Access Log Report ==="
6  echo "Generated: $(date)"
7
8  echo "--- Total Requests ---"
9  wc -l < "$LOG"
10
11 echo "--- Top 10 IP Addresses ---"
12 awk '{print $1}' "$LOG" | sort | uniq -c | sort -rn | head -10
13
14 echo "--- HTTP Status Code Summary ---"
15 awk '{print $9}' "$LOG" | sort | uniq -c | sort -rn
16
17 echo "--- Top 10 Requested URLs ---"
18 awk '{print $7}' "$LOG" | sort | uniq -c | sort -rn | head -10
19
20 echo "--- Requests Per Hour ---"
21 awk '{print $4}' "$LOG" | cut -d: -f2 | sort | uniq -c
```

Part 4: Automation & Best Practices

From Scripts to Production



Cron – Scheduling Tasks

Crontab Format

```
1  minute (0-59)
2  hour (0-23)
3  day of month (1-31)
4  month (1-12)
5  day of week (0-7)
6
7  * * * * * command
```

Managing Crontab

```
1  crontab -e           # Edit crontab
2  crontab -l           # List entries
3  sudo crontab -u root -e # Edit root's
```

Examples

```
1  # Every day at 2:30 AM
2  30 2 * * * /opt/scripts/backup.sh
3
4  # Every 15 minutes
5  */15 * * * * /opt/scripts/health_check.sh
6
7  # Monday-Friday at 9 AM
8  0 9 * * 1-5 /opt/scripts/daily_report.sh
9
10 # First day of every month
11 0 0 1 * * /opt/scripts/monthly_cleanup.sh
```



Systemd Timers (Modern Alternative)

Timer Unit: `backup.timer`

```
1 [Unit]
2 Description=Daily backup timer
3
4 [Timer]
5 OnCalendar=daily
6 Persistent=true
7
8 [Install]
9 WantedBy=timers.target
```

Service Unit: `backup.service`

```
1 [Unit]
2 Description=Backup service
3
4 [Service]
5 Type=oneshot
6 ExecStart=/opt/scripts/backup.sh
```

Managing Timers

```
1 sudo systemctl enable --now backup.timer
2 systemctl list-timers --all
3 systemctl status backup.timer
4 journalctl -u backup.service
```

💡 Systemd timers offer logging, dependencies, and better error handling than cron.



Error Handling & Strict Mode

Strict Mode

```
1  #!/bin/bash
2  set -euo pipefail
3  # -e Exit immediately on error
4  # -u Treat unset variables as errors
5  # -o pipefail Catch errors in pipelines
```

Custom Error Handling

```
1  die() {
2      echo "ERROR: $1" >&2
3      exit "${2:-1}"
4  }
5  [[ -f "$CONFIG" ]] || die "Config not found"
```

Trap – Run Cleanup on Exit

```
1  #!/bin/bash
2  set -euo pipefail
3
4  TMPDIR=$(mktemp -d)
5  trap 'rm -rf "$TMPDIR"' EXIT
6
7  cp important_data "$TMPDIR/"
8  process_data "$TMPDIR/"
9  # TMPDIR removed automatically on exit
```



Debugging Scripts

Debug Mode

```
1  bash -x script.sh      # Trace every command
2  bash -n script.sh      # Syntax check only
3  bash -v script.sh      # Print as read
```

Selective Debugging

```
1  #!/bin/bash
2  echo "Normal output"
3  set -x                # Turn on debugging
4  problematic_function
5  set +x                # Turn off debugging
6  echo "Back to normal"
```

Debugging Tips

```
1  echo "DEBUG: VAR=$VAR" >&2
2
3  [[ $# -ge 2 ]] || {
4      echo "Usage: $0 <src> <dest>"
5      exit 1
6  }
7
8  shellcheck myscript.sh
```

💡 **ShellCheck** (shellcheck.net) catches common bugs – always run it on your scripts!

Security Best Practices

Input Validation

```
1  # Sanitize user input
2  if [[ ! "$USERNAME" =~ ^[a-zA-Z0-9_]+$ ]]; then
3      die "Invalid username"
4  fi
5
6  # Avoid eval with user data
7  # BAD:  eval "$user_input"
8  # GOOD: Use arrays for commands
9  cmd=( "ls" "-la" "$dir" )
10 " ${cmd[@]} "
```

Quote Everything

```
1  # BAD:  rm -rf $DIR/*
2  # GOOD: rm -rf "${DIR:?}/*"
3  # ${DIR:?} fails if DIR is empty
```

File Permissions

```
1  # Restrict script permissions
2  chmod 700 admin_script.sh
3
4  # Secure temp files
5  TMPFILE=$(mktemp)
6  chmod 600 "$TMPFILE"
```

Avoid Common Pitfalls

```
1  # Use full paths in cron
2  PATH=/usr/local/bin:/usr/bin:/bin
3
4  # Don't store passwords in scripts
5  # Use: read -sp "Password: " PASS
6  # Or:  PASS=$(cat /etc/myapp/secret)
7
8  # Log actions for audit
9  logger "Backup started by $(whoami)"
```



Real-World Script: System Health Check

Script checks CPU load, memory, and disk usage against thresholds; logs WARN or OK.

```
#!/bin/bash
set -euo pipefail
WARN_CPU=80
WARN_MEM=85
WARN_DISK=90
HOST=$(hostname)
DATE=$(date '+%F %T')
echo "≡ Health: $HOST ≡"
echo "Date: $DATE"

# CPU: load vs cores → WARN or OK
LOAD=$(cat /proc/loadavg | cut -d' ' -f1)
CORES=$(nproc)
CPU_PCT=$((LOAD * 100 / CORES))
if [[ "$CPU_PCT" -gt "$WARN_CPU" ]]; then echo "[WARN] CPU: $CPU_PCT%"; else echo "[ OK ] CPU: $CPU_PCT%"; fi

# Memory: usage % → WARN or OK
MEM_PCT=$(free | awk '/Mem:/ {print int($3/$2*100)}')
if [[ "$MEM_PCT" -gt "$WARN_MEM" ]]; then echo "[WARN] Memory: $MEM_PCT%"; else echo "[ OK ] Memory: $MEM_PCT%"; fi
```



Real-World Script: Backup with Rotation

```
1  #!/bin/bash
2  set -euo pipefail
3  # rotate_backup.sh
4
5  BACKUP_DIR="/backup"
6  SOURCE="/var/www"
7  RETENTION_DAYS=30
8  DATE=$(date +%Y%m%d_%H%M%S)
9  ARCHIVE="${BACKUP_DIR}/www_${DATE}.tar.gz"
10 LOGFILE="/var/log/backup.log"
11
12 log() {
13     echo "$(date '+%F %T') $1" | tee -a "$LOGFILE"
14 }
```

```
1  log "Starting backup of $SOURCE"
2  mkdir -p "$BACKUP_DIR"
3
4  if tar czf "$ARCHIVE" \
5      -C "$(dirname "$SOURCE")" \
6      "$(basename "$SOURCE")"; then
7      SIZE=$(du -sh "$ARCHIVE" | cut -f1)
8      log "Backup complete: $ARCHIVE ($SIZE)"
9  else
10     log "ERROR: Backup failed!"; exit 1
11 fi
12
13 # Rotate old backups
14 DELETED=$(find "$BACKUP_DIR" -name "*.tar.gz" \
15     -mtime +$RETENTION_DAYS -delete -print | wc -l)
16 log "Cleaned up $DELETED old backup(s)"
```








Summary: Bash Scripting & Automation

Key Concepts Covered

1. **Script Basics:** Shebang, execution, variables
2. **Special Variables:** `$1` , `$@` , `$?` , `$$`
3. **Control Structures:** `if` , `case` , `for` , `while`
4. **Functions:** `local` scope, return values
5. **Text Processing:** `grep` , `sed` , `awk`
6. **Pipelines:** `|` , redirection (`>` , `2>&1`)
7. **Automation:** Cron & systemd timers
8. **Best Practices:** Strict mode, security










The Scripting Mindset

-  If you do it twice, script it
-  Always use strict mode (`set -euo pipefail`)
-  Quote your variables
-  Validate inputs
-  Log everything



Learning Objectives: Did We Achieve?

By now, you should be able to:

-  Write and execute Bash scripts with proper structure
-  Use variables, arguments, and arithmetic
-  Implement conditionals and loops for control flow
-  Define and use functions with local scope
-  Process text with `grep` , `sed` , `awk` , and pipelines
-  Schedule automated tasks with cron and systemd timers
-  Apply error handling and strict mode
-  Follow security best practices in scripts
-  Build real-world system administration scripts



Next Week: Linux Networking Basics - Configure IP, DNS, and network interfaces!



Lab Practice: Bash Scripting

Exercise 1: User info script

- Write a script that takes a username as argument and prints: UID, GID, home dir, shell (use `getent` or parse `/etc/passwd`).
- Add a check: exit with an error message if no argument is given.

Exercise 2: Log summary

- In `/var/log` (or a copy), use `grep` , `cut` , `sort` , `uniq` to list the **top 5 most common** words in a log file (ignore case; skip very short words).
- Pipe the result into a small script that prints a one-line summary.

Exercise 3: Safe backup script

- Write a script that tars a given directory into `/tmp/backups` with a timestamp in the name.
- Use `set -euo pipefail` , check that the directory exists, and print the path of the created archive.

Exercise 4: `sed` config tweak

- Take a copy of a config file (e.g. `sshd_config` or any `.conf`). Use `sed` to comment out every line that contains a given keyword (e.g. `Port`), then show a diff.

Additional Resources

Documentation

- [GNU Bash Manual](#) - Official reference
- [Bash Hackers Wiki](#) - Community wiki
- [ShellCheck](#) - Online script linter

Books & Guides

- *"The Linux Command Line"* by William Shotts - Scripting chapters
- [Advanced Bash-Scripting Guide](#) - Comprehensive
- [Google Shell Style Guide](#) - Best practices

Practice

- Write a script to automate user account creation
- Build a log parser for Apache/Nginx access logs
- Create a backup script with rotation and email alerts

Questions?

Next: Linux Networking Basics

