Linux and the Shell

Version Bash 5.3 · verified 2026-11-02

Bash and the core command-line tools — pipes, quoting, grep, find, sed, awk, jq, processes, permissions, SSH and safe scripts.

Contents

Mental model 2

Navigation and files 2

Viewing and searching 3

Pipes and redirection 4

Quoting and expansion 5

Text processing 6

Processes and signals 7

Permissions 7

Environment, PATH and SSH 8

A safe script template 9

Further reading 11

Mental model

Every command reads text from standard input, writes text to standard output, and complains on standard error; the shell's job is to wire those streams together and to expand your words before the command ever sees them. Learn the pipe, redirection and quoting rules and the rest is lookup. Most surprises come from expansion: the command receives what the shell produced, not what you typed.

Navigation and files

Command	Does
`pwd`, `cd -`	Where am I; jump back
`ls -la`	List all, with details
`mkdir -p a/b/c`	Create parents as needed
`cp -R src dst`	Copy a directory
`mv old new`	Move or rename
`ln -s target name`	Symbolic link

Paths starting with `/` are absolute; everything else is relative to the working directory. `~` expands to your home directory, `.` is here, `..` is the parent. These commands and flags behave the same on Linux (GNU) and macOS (BSD).

```
mkdir -p demo/logs && cd demo
touch logs/app.log
ln -s logs/app.log latest.log
```

```
ls -la            # latest.log -> logs/app.log
cd - >/dev/null   # back to the previous dir
```

> **Warning:** `rm -rf "$dir/"` with an empty `$dir` becomes `rm -rf /`. Guard it: `rm -rf "${dir:?}/"` aborts when `dir` is unset or empty.

Viewing and searching

Command	Does
`less file`	Page; `/` searches, `q` quits
`tail -f app.log`	Follow a growing file
`grep -rn 'TODO' src`	Recursive, with line numbers
`grep -v`, `-i`, `-l`	Invert, ignore case, names only
`find . -name '*.ts' -mtime -1`	By name and age
`rg TODO`	ripgrep: fast, respects `.gitignore`

`grep` filters lines, `find` walks the tree. `find` applies its tests left to right, so put cheap name tests before expensive actions. To feed `find` results to another command, use `-exec ... {} +`, which batches files into as few calls as possible.

```
grep -rn --include='*.ts' 'fetch(' src
find . -name '*.log' -size +10M
find . -name '*.tmp' -exec rm {} +
find . -type f -name '*.md' | wc -l
```

> **Gotcha:** Quote the pattern in `find . -name '*.ts'`. Unquoted, the shell expands `*.ts` against the current directory first, and `find` receives `a.ts b.ts` — or fails with `paths must precede expression`.

Pipes and redirection

Syntax	Effect
`a | b`	stdout of `a` into stdin of `b`
`> f`, `>> f`	Write stdout, truncate or append
`2> f`, `2>/dev/null`	Redirect stderr
`> f 2>&1`	Both streams into `f`
`a | tee f`	Show and save
`<<'EOF'`	Here-document as stdin

File descriptor 0 is stdin, 1 is stdout, 2 is stderr. Redirections are processed left to right, and `2>&1` means “make 2 point where 1 points right now”. Quote the here-doc delimiter (`'EOF'`) to stop expansion inside it.

```
ls missing > out.txt 2>&1   # both in out.txt
ls missing 2>&1 > out.txt   # error on terminal
cat <<'EOF' > note.txt
Literal $HOME, not expanded
EOF
```

> **Gotcha:** `cmd 2>&1 > file` sends errors to the terminal, because stderr was duplicated before stdout moved. Put `2>&1` last.

Quoting and expansion

You write	Shell passes
`'$HOME'`	`$HOME` — literal
`"$HOME"`	`/home/me` — one word
`$(date +%F)`	Command output
`*.log`	Matching file names
`{a,b}.txt`	`a.txt b.txt`
`${name:-dev}`	`dev` when unset or empty

Single quotes are fully literal. Double quotes still expand `$var` and `$(...)` but keep the result as one word. Unquoted expansions are split on whitespace and globbed — the source of most shell bugs.

```
file="my report.txt"
touch "$file"
ls $file     # ls: cannot access 'my': ...
ls "$file"   # my report.txt
echo "today is $(date +%F)"
for f in *.txt; do echo "[$f]"; done
```

> **Tip:** Quote every expansion — `"$var"`, `"$(cmd)"`, `"$@"` — unless you specifically want splitting. ShellCheck flags the ones you miss.

Text processing

Tool	Typical use
`cut -d, -f2`	Pick a column by delimiter
`sort | uniq -c | sort -rn`	Count and rank
`sed 's/old/new/g'`	Substitute on each line
`awk '{print $1}'`	Fields and arithmetic
`jq '.items[].name'`	Query JSON
`xargs -0`	Turn input into arguments

Chain small tools in a pipeline. `awk` splits on whitespace by default (`-F` sets the separator) and is the right tool once you need sums or conditions. `jq -r` prints raw strings without JSON quotes.

```
printf '1.1.1.1 GET\n2.2.2.2 GET\n1.1.1.1 POST\n' \
  > access.log
awk '{print $1}' access.log | sort | uniq -c \
  | sort -rn                # 2 1.1.1.1
awk '{n++} END {print n}' access.log   # 3
echo '{"items":[{"name":"a"}]}' \
  | jq -r '.items[].name'               # a
sed -i.bak 's/GET/HEAD/' access.log
```

> **Gotcha:** `sed -i` differs by platform. GNU sed takes an optional suffix; macOS (BSD) sed requires one, so `sed -i 's/a/b/' f` fails there. `sed -i.bak ...` works on both.

Processes and signals

Command	Does
`cmd &`	Run in the background
`jobs`, `fg`, `bg`	Manage background jobs
`ps aux | grep node`	Find a process
`kill PID`	Send `SIGTERM` (15)
`kill -9 PID`	`SIGKILL`: cannot be caught
`trap 'cleanup' EXIT`	Run code when the script exits

Ctrl-C sends `SIGINT`, Ctrl-Z sends `SIGTSTP` (suspend). `$?` holds the last exit status: 0 is success, anything else is failure. `$!` is the PID of the last background job.

```
sleep 30 &
pid=$!
kill "$pid"
wait "$pid"; echo "exit $?"   # exit 143
```

> **Warning:** Reach for `kill -9` last. `SIGKILL` skips the process's handlers, so it never flushes buffers, removes lock files or closes connections cleanly. Send `SIGTERM` and wait first.

Permissions

Mode	Meaning
`755`	`rwxr-xr-x`: scripts, directories

Mode	Meaning
`644`	`rw-r--r--`: ordinary files
`600`	`rw-------`: private keys
`chmod +x f`	Add execute
`chown user:group f`	Change owner

Each digit is owner, group, others; read is 4, write 2, execute 1. On a directory, `x` means “may enter”: without it you cannot `cd` into the directory or open files inside it, even when `r` is set. Ownership changes usually need `sudo`.

```
printf '#!/usr/bin/env bash\necho hi\n' > hi.sh
./hi.sh       # Permission denied
chmod +x hi.sh
./hi.sh       # hi
ls -l hi.sh   # -rwxr-xr-x ...
```

> **Gotcha:** `chmod -R 755 dir` also makes every file executable. Use `chmod -R u=rwX,go=rX dir`: capital `X` adds execute only to directories and files already executable.

Environment, PATH and SSH

Command	Does
`export API_URL=...`	Pass a variable to child processes
`VAR=1 cmd`	Set it for one command only

Command	Does
`command -v node`	Which binary runs
`ssh-keygen -t ed25519`	Create a key pair
`ssh-copy-id user@host`	Install your public key
`~/.ssh/config`	Per-host aliases and keys

A variable without `export` stays in the current shell. `PATH` is a colon-separated list searched left to right, so prepend to win. Put permanent settings in `~/.bashrc` (Bash) or `~/.zshrc` — zsh is the default login shell on macOS.

```
export PATH="$HOME/.local/bin:$PATH"
LOG_LEVEL=debug node -e \
  'console.log(process.env.LOG_LEVEL)'   # debug
ssh-keygen -t ed25519 -C "me@laptop"
ssh-copy-id -i ~/.ssh/id_ed25519.pub me@srv
```

> **Gotcha:** `ssh` refuses a private key that others can read (`UNPROTECTED PRIVATE KEY FILE!`). `chmod 600 ~/.ssh/id_ed25519` and `chmod 700 ~/.ssh`.

A safe script template

Line	Why
`#!/usr/bin/env bash`	Find Bash on `PATH`
`set -e`	Exit on an unhandled failure

Line	Why
`set -u`	Unset variable is an error
`set -o pipefail`	A failing stage fails the pipe
`"$@"`	All arguments, each kept intact

Without `pipefail`, a pipeline's status is its last command's, so `curl ... | jq` "succeeds" when `curl` fails. `set -e` does not fire inside `if` conditions or `&&`/`||` lists, so check those yourself.

```
#!/usr/bin/env bash
set -euo pipefail

tmp="$(mktemp -d)"
trap 'rm -rf "$tmp"' EXIT

main() {
  local name="${1:?usage: $0 NAME}"
  echo "hello, $name" > "$tmp/out"
  cat "$tmp/out"
}
main "$@"
```

> **Tip:** Run `shellcheck script.sh` before committing. It catches unquoted expansions, unused variables and Bash-only syntax in a `#!/bin/sh` script.

Further reading

- GNU Bash Reference Manual
- GNU Coreutils manual
- GNU sed manual
- GNU Findutils manual
- jq manual
- OpenSSH manual pages