Shell Scripting and Automation Course

Flavio Copes

Updated August 3, 2026

Contents

Turn shell commands into dependable Bash programs with safe quoting, structured control flow, temporary files, logging, locks, and tests.

What you'll learn: Write and test maintainable scripts that validate input, preserve data, report useful failures, and run safely without supervision.

Script foundations

Create executable scripts, quote values, read arguments, and make exit status meaningful.

Create an executable script

Write a Bash script with an interpreter line, make it executable, and run it from a predictable directory.

A shell script is a text file containing commands. When you run it, the operating system starts an interpreter and feeds it the file. The **interpreter line** on the first line — often called the shebang — tells the operating system which program should read it.

Write the script

Create a file named `hello`:

```
#!/usr/bin/env bash

printf '%s\n' 'hello from a script'
```

The first line must start with `#!`. I use `#!/usr/bin/env bash` instead of a hardcoded `/bin/bash` because `env` finds Bash wherever it lives on `PATH`. On macOS the system Bash in `/bin` is ancient, and the current one from Homebrew sits somewhere else entirely. `env` picks the right one on each machine.

`printf '%s\n' 'text'` is the predictable way to print. Unlike `echo`, it never reinterprets escapes or flags hidden inside the value.

Make it executable and run it

A fresh file is not executable. Give yourself execute permission, then run it:

```
chmod u+x hello
./hello
# hello from a script
```

If you skip `chmod`, the shell answers `Permission denied`. That error means the file exists but you're not allowed to execute it — the fix is the permission, not the path.

Why the ./ prefix

The `./` prefix names the file in the current directory. Without a slash, the shell searches directories in `PATH`, and the current directory is not on that list:

```
hello
# bash: hello: command not found
./hello
# hello from a script
```

This trips up everyone once. `command not found` for a file sitting right there means you forgot the `./`.

Do not add the current directory to `PATH` globally to ”fix” this. Any directory you `cd` into could then shadow real commands with a malicious file named `ls`. Run local scripts through an explicit path.

The shebang stays in charge

You can also run a script without execute permission by naming the interpreter yourself:

```
bash hello
# hello from a script
```

That works, but it ignores the shebang. If someone runs a Bash-only script with `sh` this way, the syntax breaks in confusing places. Making the file executable and running `./hello` keeps the interpreter line in control.

Use variables and quoting

Store values and expand them without accidental word splitting or wildcard expansion.

Shell variables hold text. You assign with `name=value` — no spaces around the `=` — and read the value back with `$name`.

Two quoting rules do most of the work in shell. **Double quotes** preserve one argument while still expanding variables inside. **Single quotes** preserve literal text, with no expansion at all:

```
project='My Notes'
printf '%s\n' "$project"   # My Notes
printf '%s\n' '$project'   # $project
```

What unquoted expansion does

Here is the classic failure. Compare quoted and unquoted expansion:

```
project='My Notes'
printf '<%s>\n' "$project"
# <My Notes>
printf '<%s>\n' $project
# <My>
# <Notes>
```

The quoted command prints one value. The unquoted command passes two words: after Bash expands `$project`, it splits the result on spaces, tabs, and newlines before the command runs. This is **word splitting**. `printf` received two separate arguments and applied the format to each one.

Now put the same mistake next to a destructive command:

```
file='old notes.txt'
rm $file
# rm: cannot remove 'old': No such file or directory
# rm: cannot remove 'notes.txt': No such file or directory
```

`rm` was asked to delete two files named `old` and `notes.txt`. If a file named `old` had existed, it would be gone now. `rm "$file"` removes exactly the one file you meant.

Wildcards expand too

Word splitting is not the only surprise. After splitting, Bash performs pathname expansion on unquoted results, so a value containing `*` gets matched against files in the current directory:

```
pattern='*'
```

```
printf '%s\n' $pattern    # prints every filename in the directory
printf '%s\n' "$pattern"  # *
```

A variable you thought held a harmless string just enumerated your files.

The habit to build

Quote every expansion unless you can explain why you want splitting: `"$project"`, `"$1"`, `"$HOME/backups"`. Double-quoting a variable you want treated as one value is never wrong.

And never rely on filenames having no spaces, wildcard characters, or leading dashes. Someone eventually creates `My Notes (final) *.txt`, and unquoted scripts break exactly then. ShellCheck flags every risky unquoted expansion — we run it at the end of the course.

Read positional arguments

Accept a required path argument, show usage, and distinguish the script name from user values.

When you run `./check-dir /var/www`, Bash stores the script name in `$0` and arguments in `$1`, `$2`, and so on. `$#` reports how many arguments were supplied. These are the **positional parameters**, and they are how a script receives input without prompting anyone.

Require an argument

A script that needs input should check for it up front and explain itself when the caller gets it wrong. Create `check-dir`:

```
#!/usr/bin/env bash

if (( $# != 1 )); then
  printf 'usage: %s DIRECTORY\n' "$0" >&2
  exit 2
fi

directory=$1
printf 'checking %s\n' "$directory"
```

`(( ... ))` is arithmetic evaluation, the natural way to compare numbers like `$#`. The usage message goes to standard error with `>&2`, not standard output — error text must never mix with real output that another program might consume. Status `2` is the conventional "you called me wrong" code; `grep` and many other tools use it the same way.

Copying `$1` into a named variable is a small kindness to future readers. Twenty lines down, `"$directory"` explains itself; `"$1"` doesn't.

Verify all three cases

Run it with zero, one, and two arguments:

```
./check-dir
# usage: ./check-dir DIRECTORY

./check-dir /var/www
# checking /var/www
```

```
./check-dir /var/www /tmp
# usage: ./check-dir DIRECTORY
```

Check the status too: `echo $?` prints `2` after the invalid calls and `0` after the good one. Invalid usage should print to standard error and return status 2 every time, because schedulers and other scripts act on the number, not the text.

Count is not meaning

The realistic mistake is stopping here. One argument can still name a missing or unsafe path:

```
if [[ ! -d $directory ]]; then
  printf '%s is not a directory\n' "$directory" >&2
  exit 1
fi
```

Validate meaning after count. A script that accepts any string as its ”log directory” and marches on will fail later, somewhere far less obvious than line three.

One more parameter worth knowing now: `"$@"` expands to all arguments, each preserved as its own word. We put it to work in the lesson on loops.

Return a useful exit status

Use zero for success, non-zero for failure, and preserve the command result automation needs.

Every command finishes with an **exit status**, a number from 0 to 255. Status zero means success; other values identify a failed or exceptional outcome. Shell automation composes commands through exit status: `if`, `&&`, `||`, cron, systemd, and CI pipelines all read it. Your script reports one too, whether you thought about it or not.

Read a status

The special parameter `$?` holds the status of the last command:

```
ls /etc/hostname
# /etc/hostname
printf '%s\n' "$?"
# 0

ls /nonexistent
# ls: cannot access '/nonexistent': No such file or directory
printf '%s\n' "$?"
# 2
```

Observe the status with `printf '%s\n' "$?"` immediately afterward. Any command you run in between — even an innocent `echo` — replaces it with its own status.

Fail early and clearly

A script should stop with a non-zero status the moment its requirements aren't met. Check a required command:

```
if ! command -v curl >/dev/null 2>&1; then
```

```
  printf '%s\n' 'curl is required' >&2
  exit 1
fi
```

`command -v curl` succeeds when `curl` exists on `PATH` and fails otherwise. We discard its output because only the status matters here. `exit 1` makes the failure visible to whatever ran the script.

To verify both paths, run the script normally and with a restricted `PATH`:

```
PATH=/nonexistent bash check-tools.sh
# curl is required
printf '%s\n' "$?"
# 1
```

The trap: exiting zero by accident

Without an explicit `exit`, a script returns the status of its last command. That rule causes a classic bug:

```
if ! tar -czf /mnt/backups/site.tar.gz -C /var/www .; then
  printf '%s\n' 'backup failed' >&2
fi
# script ends here - printf succeeded, so the script exits 0
```

The error message went out, but `printf` was the last command and it succeeded, so the script reports success. Cron marks the job green and nobody looks again for months.

Do not print an error and then accidentally exit zero. Callers trust the status more than human prose — put `exit 1` right after the message, every time.

Check your understanding: script foundations

Check the commands, decisions, safety boundaries, and failure cases from script foundations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Control and functions

Use strict checks, conditions, loops, case matching, and small reusable functions.

Enable safer failure behavior

Use unset-variable and pipeline checks while understanding that automatic exit is not a substitute for error handling.

By default, Bash keeps going. A typo expands to an empty string, a failing pipeline reports success, and a broken command in the middle of a script just scrolls past. Bash defaults can hide failures. Two settings tighten this up: `set -u` rejects unset variables, and `set -o pipefail` makes a pipeline reflect failing stages.

Start a script with explicit policy:

```
#!/usr/bin/env bash
```

```
set -u
set -o pipefail
```

What set -u catches

Create an unset variable in a disposable script and watch:

```
target=/var/www
rm -rf "$targt/cache"
# bash: targt: unbound variable
```

Without `-u`, the misspelled `$targt` expands to nothing and the command becomes `rm -rf /cache` — a different directory entirely. With `-u`, the script stops at the typo. This one behavior pays for itself for years.

Legitimately optional variables still work: write `${1:-}` or `${DEBUG:-0}` to supply an explicit default.

What pipefail catches

A pipeline normally reports only the status of its last command:

```
cat /nonexistent/data.txt | wc -l
# cat: /nonexistent/data.txt: No such file or directory
# 0
printf '%s\n' "$?"
# 0 without pipefail - wc succeeded, so cat's failure vanished
# 1 with pipefail - the failing stage comes through
```

With `set -o pipefail`, a failing pipeline returns the status of the last command that failed. Create a failing pipeline like this one in your disposable script and confirm the difference yourself.

What about set -e?

You will see the trio written as `set -euo pipefail`. The `-e` part exits the script when a command fails, and it sounds perfect. Treat `set -e` carefully, though: its exceptions depend on command context. A failure inside an `if` condition, on either side of `&&` or `||`, or in a negated command does not trigger an exit — and that surprises people constantly.

My advice: use `set -u` and `set -o pipefail` everywhere. Add `-e` as a safety net if you like it, but don't call it error handling. Explicit `if` checks are clearer around recovery paths:

```
if ! cp nginx.conf /etc/nginx/nginx.conf; then
  printf '%s\n' 'config copy failed, keeping the old file' >&2
  exit 1
fi
```

Add deliberate checks around failures you expect. Automatic exit can't decide what should happen next — you can.

Branch with if and case

Test command success with if and select a small set of named modes with case.

In shell, `if` doesn't test truth the way other languages do. It runs a command and branches on its

exit status: zero takes the `then` branch, anything else goes to `else`. `case` matches one value against patterns and is clear for command modes.

if runs commands

Because `if` evaluates a command's exit status, any command works as a condition:

```
if grep -q 'server_name' /etc/nginx/nginx.conf; then
  printf '%s\n' 'config mentions server_name'
else
  printf '%s\n' 'no server_name found' >&2
fi
```

Tests on strings and files use `[[ ... ]]`, which is itself a command that exits zero when the test holds:

```
if [[ -f /etc/nginx/nginx.conf && $USER == 'deploy' ]]; then
  printf '%s\n' 'ready to check the config'
fi
```

Prefer `[[ ... ]]` over the older `[ ... ]` for Bash-specific scripts: it doesn't word-split unquoted variables and it supports pattern matching. I still quote arbitrary values in tests — the habit protects you in every other context where quoting does matter.

case for named modes

When a script accepts a small set of subcommands, a chain of `elif` branches gets noisy fast. Dispatch a mode:

```
case ${1:-} in
  check) printf '%s\n' 'checking' ;;
  run) printf '%s\n' 'running' ;;
  *) printf 'usage: %s {check|run}\n' "$0" >&2; exit 2 ;;
esac
```

`${1:-}` expands to an empty string when no argument was given, so the script behaves under `set -u` instead of dying with an unbound-variable error. Each branch ends with `;;`.

The `*` pattern is the part people forget. It catches everything the named patterns missed. Leave it out and unknown input matches nothing — the `case` finishes silently and the script continues as if the argument were fine.

Verify every path

Run every accepted value and one rejected value:

```
./service-ctl check   # checking
./service-ctl run     # running
./service-ctl stop    # usage: ./service-ctl {check|run}
```

Each path should have a documented status. After the rejected call, `echo $?` must print `2` — callers and schedulers branch on that number, not on the usage text.

Loop over arguments and files

Iterate over exact command-line arguments and handle a glob that may match nothing.

`"$@"` expands to one shell word per original argument. It is the safe way to forward or loop over all supplied values — spaces inside an argument stay inside it.

Loop over arguments

Print every input path:

```
for path in "$@"; do
  if [[ -e $path ]]; then
    printf 'exists: %s\n' "$path"
  else
    printf 'missing: %s\n' "$path" >&2
  fi
done
```

Test paths containing spaces, dashes, and wildcard characters. Each must remain one value:

```
./check-paths 'weekly report.txt' notes.md
# exists: weekly report.txt
# missing: notes.md
```

The quotes carry the whole guarantee. Unquoted `$@` or `$*` splits `weekly report.txt` into two words, and the loop checks two files that don't exist. Same inside the loop body: `"$path"` keeps the value whole.

Loop over files with globs

For files on disk, let the shell expand a glob:

```
for logfile in /var/log/nginx/*.log; do
  printf 'compressing %s\n' "$logfile"
done
```

The expansion happens before the loop starts, and each match arrives as one word. Filenames with spaces are safe with no extra effort.

One trap remains: a glob that may match nothing. When no file matches, Bash leaves the pattern in place, and your loop runs once with the literal string `/var/log/nginx/*.log` as its value. The script then claims to compress a file that doesn't exist. Fix it with `nullglob`:

```
shopt -s nullglob
for logfile in /var/log/nginx/*.log; do
  printf 'compressing %s\n' "$logfile"
done
```

With `nullglob` set, an unmatched glob expands to nothing and the loop body never runs. Test both states: a directory with matching files and an empty one.

Do not parse ls

Old tutorials show `for f in $(ls)`. Don't copy it. The output of `ls` is text, and command substitution word-splits it, so any filename with a space shatters into pieces. There is no quoting fix for this pattern — the information is already destroyed.

Work from arguments, globs, `find -print0`, or another structured source. The `find` variant gets its own lesson in the next module.

Write small functions

Extract repeated work into functions that accept arguments and return useful status.

A shell function is a named block of commands. It shares the script environment — same variables, same working directory — so it behaves less like a function in JavaScript or Python and more like a labeled section of your script. Keep it small, use local variables, and return status through command success.

Define one focused helper

```
require_command() {
  local name=$1
  command -v "$name" >/dev/null 2>&1 || {
    printf 'missing command: %s\n' "$name" >&2
    return 1
  }
}

require_command curl || exit 1
```

Inside a function, `$1`, `$2`, and `$#` refer to the function's own arguments, not the script's. `local name=$1` gives the value a readable name that exists only inside the function.

`return` sets the function's exit status the way `exit` sets the script's. A function without an explicit `return` reports the status of its last command — the same rule scripts follow.

Verify both outcomes

Call it with an installed and missing command:

```
require_command curl
printf '%s\n' "$?"
# 0

require_command doesnotexist
# missing command: doesnotexist
printf '%s\n' "$?"
# 1
```

The function's output helps humans; its return status helps the caller. Keep the two channels separate: messages to standard error, results through the status.

The failure mode: hidden globals

Forget `local` and every variable a function touches leaks into the whole script:

```
count_lines() {
  file=$1            # no local - overwrites any outer $file
  wc -l < "$file"
}

file=config.txt
count_lines access.log
```

```
printf '%s\n' "$file"
# access.log - the caller's variable was silently replaced
```

These bugs are miserable to find because the damage appears far from the function that caused it. Avoid hidden global mutations: pass values in as arguments and mark internal variables `local`.

When a function needs to hand a value back, print it and let the caller capture it with command substitution:

```
lines=$(count_lines access.log)
```

That keeps data flowing through explicit channels — arguments in, output and status out — which is what makes a function safe to reuse.

Check your understanding: control and functions

Check the commands, decisions, safety boundaries, and failure cases from control and functions.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Files and data

Manage temporary files, read lines, find files safely, and process JSON without fragile text parsing.

Create and clean temporary files

Use mktemp and a trap so unique temporary data is removed on success, failure, or interruption.

Scripts need scratch space: an archive to assemble, a download to inspect, a sorted copy of a log. Hardcoding a path like `/tmp/work.txt` looks harmless, but predictable temporary filenames can collide or be replaced by another user on a shared machine. `mktemp` asks the operating system for a unique private path instead.

Create a temporary directory with cleanup

```
work=$(mktemp -d)
cleanup() { rm -rf -- "$work"; }
trap cleanup EXIT INT TERM

printf 'working in %s\n' "$work"
# working in /tmp/tmp.X8s2Lk4bQz
```

`mktemp -d` creates a directory only your user can access and prints its path, which we capture with command substitution.

The `trap` line makes this reliable. A **trap** registers a command to run when the shell receives a signal or, with the special `EXIT` condition, whenever the script ends. This one runs `cleanup` on normal exit, on Ctrl-C (`INT`), and on termination (`TERM`) — so the directory disappears whether the script succeeds, fails, or gets interrupted halfway.

The `--` in `rm -rf -- "$work"` ends option parsing, so even a path starting with a dash can't be misread as a flag.

Verify the cleanup runs

Exit normally and interrupt the script. Confirm the exact generated directory is removed both times:

```
./make-report
# working in /tmp/tmp.QpB7wq2c1M
ls /tmp/tmp.QpB7wq2c1M
# ls: cannot access '/tmp/tmp.QpB7wq2c1M': No such file or directory
```

Run it again and press Ctrl-C mid-run, then check again. If the directory survives an interrupt, the trap was registered too late — register it immediately after creating the resource, before any work that can fail.

The failure mode to fear

`rm -rf` next to a variable that might be empty is the classic shell disaster:

```
rm -rf "$workdir/cache"   # if $workdir is empty: rm -rf /cache
```

Validate the variable before any recursive deletion. `set -u` catches unset names, and a one-line guard costs nothing:

```
[[ -n $work ]] || exit 1
```

Never build a destructive target from an empty or broad environment variable. The pattern above — `mktemp` output, straight into a trap — is safe precisely because the value comes from a program you just ran, not from inherited environment.

Read a file line by line

Preserve leading whitespace and backslashes while reading every line, including a final line without newline.

The reliable Bash pattern for reading a file line by line disables backslash escapes and preserves surrounding whitespace. It's more ceremony than you'd expect, and every piece earns its place.

The pattern

Read `servers.txt`:

```
while IFS= read -r server || [[ -n $server ]]; do
  printf 'server=%s\n' "$server"
done < servers.txt
```

Three details do the work:

- `IFS=` clears the field separator for this one `read`, so leading and trailing spaces survive instead of being trimmed.
- `-r` stops `read` from treating backslashes as escape characters. Without it, a `\` in the file arrives mangled or vanishes.
- `|| [[ -n $server ]]` handles a final line without a trailing newline. `read` returns non-zero at end of file, but it still filled the variable — this check processes that last piece instead of silently dropping it.

The redirection `< servers.txt` feeds the file to the loop as a whole.

Verify with a hostile file

Build a test file that covers the edge cases:

```
printf '  indented\nback\\slash\nlast line' > servers.txt
./read-servers
# server=  indented
# server=back\slash
# server=last line
```

Test blank lines, spaces, a backslash, and a final unterminated line. Then decide which lines the application should ignore — many scripts skip blanks and comments with a guard at the top of the loop:

```
[[ -z $server || $server == \#* ]] && continue
```

The failure mode: piping into the loop

This variant looks equivalent and isn't:

```
count=0
cat servers.txt | while IFS= read -r server; do
  count=$((count + 1))
done
printf '%s\n' "$count"
# 0
```

Each part of a pipeline runs in its own subshell, so `count` was incremented in a child process and thrown away when the pipeline finished. The `done < servers.txt` redirection keeps the loop in your shell, and its variables survive. Reach for the redirection first; it's also one process cheaper.

One last thing: a line is still untrusted data. Validate hostnames, paths, and options before using them in commands — a config file is an input, not a promise.

Find files with null delimiters

Process filenames containing spaces and newlines by separating records with a null byte.

A Linux filename can contain any character except the null byte and `/` — including newlines. That makes newline-delimited filename lists ambiguous: a filename may contain a newline, and the consumer can't tell one file from two. GNU `find` and `xargs` support null-separated records, and a null byte can never appear inside a path, so the separator is unambiguous.

Stream filenames safely

Print sizes safely:

```
find ./uploads -type f -print0 | while IFS= read -r -d '' file; do
  wc -c < "$file"
done
```

`-print0` ends each result with a null byte instead of a newline. On the reading side, `read -d ''` sets the delimiter to the null byte, so each filename arrives whole — spaces, newlines, quotes and all.

When you don't need a shell loop, `xargs -0` is the compact partner:

```
find ./uploads -type f -name '*.tmp' -print0 | xargs -0 rm --
```

`xargs -0` splits its input on null bytes and passes each name as one clean argument.

Verify with hostile names

Create authorized test filenames with spaces and unusual characters — in a directory you own, made for this test:

```
mkdir -p /tmp/findlab && cd /tmp/findlab
touch 'plain.txt' 'has space.txt' $'new\nline.txt'
find . -type f -print0 | while IFS= read -r -d '' f; do
  printf '<%s>\n' "$f"
done
```

Confirm every file is handled once: three files in, three records out, and the newline-bearing name stays one record. The angle brackets in the output make any accidental split visible.

The failure this replaces

The naive version is `for f in $(find . -type f)`. Command substitution turns the results into one string, word splitting chops it on every space and newline, and `has space.txt` becomes two broken values. It works in a tidy demo directory and corrupts data the first time a real filename gets creative.

Two placement habits finish the job. Place fixed command options before untrusted filenames, and use `--` where the command supports it, so a file named `-rf` is treated as a name rather than parsed as a flag.

Process JSON with jq

Extract and validate structured data with jq instead of matching JSON using grep or regular expressions.

Sooner or later a script needs data from an API response or a JSON config file. Resist the urge to `grep` it. JSON strings can contain whitespace, escapes, and reordered fields — a regex that works today breaks the day the producer changes formatting. A JSON parser understands the structure. On the command line, that parser is `jq`.

Extract values

Say `services.json` describes what should run:

```
{
  "services": [
    { "name": "web", "active": true },
    { "name": "worker", "active": false },
    { "name": "cron", "active": true }
  ]
}
```

Read active service names:

```
jq -r '.services[] | select(.active == true) | .name' services.json
# web
```

```
# cron
```

The filter reads left to right: `.services[]` streams the array elements, `select` keeps the matching ones, `.name` extracts the field. The `-r` flag prints raw strings instead of quoted JSON, which is what a shell script wants to consume.

Capture the result with command substitution when you need it in a variable:

```
active=$(jq -r '.services[] | select(.active) | .name' services.json)
```

Verify against real variation

Test reordered properties, escaped strings, and an empty array. `jq` handles all three identically because it parses structure, not text: the same data with fields in a different order produces the same output. That is the whole argument against grep, demonstrated in thirty seconds.

Let jq control script success

Add `jq -e` when the filter result should control script success. With `-e`, the exit status is non-zero when the result is `false`, `null`, or missing:

```
if jq -e '.services[] | select(.name == "web")' services.json >/dev/null; then
  printf '%s\n' 'web is configured'
else
  printf '%s\n' 'web is missing from services.json' >&2
  exit 1
fi
```

Without `-e`, `jq` exits zero even when the filter finds nothing, and your check silently passes. That's the realistic mistake: a guard that guards nothing.

One safety rule to close. Treat output as data, not shell code — extracted values go into quoted variables and arguments. Never pass it to `eval`: a hostile `"name": "web; rm -rf ~"` must stay an odd-looking string, not become a command.

Check your understanding: files and data

Check the commands, decisions, safety boundaries, and failure cases from files and data.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Reliable automation

Handle configuration, secrets, logs, retries, locking, and scheduled non-interactive execution.

Load configuration deliberately

Separate defaults, environment input, and arguments without executing an untrusted configuration file.

A script gets its settings from three places: defaults baked into the script, environment variables, and arguments. The tempting fourth is `source config.sh` — but shell `source` executes a file as

code. Anyone who can edit that file owns every run of your script, with your permissions. For simple configuration, prefer environment variables or parse a constrained format.

Environment variables with explicit policy

Bash parameter expansion states requirements and defaults in one line each. Require a destination with a default interval:

```
backup_destination=${BACKUP_DESTINATION:?set BACKUP_DESTINATION}
backup_interval=${BACKUP_INTERVAL:-daily}
printf 'destination=%s interval=%s\n' "$backup_destination" "$backup_interval"
```

`${VAR:?message}` stops the script with your message when the variable is unset or empty. `${VAR:-default}` substitutes a fallback instead. Between the two forms, the script itself documents which settings are required and which have defaults — no separate README to drift out of date.

Verify the three cases

Run with missing, valid, and empty values:

```
./backup.sh
# ./backup.sh: line 3: BACKUP_DESTINATION: set BACKUP_DESTINATION

BACKUP_DESTINATION=/mnt/backups ./backup.sh
# destination=/mnt/backups interval=daily

BACKUP_DESTINATION= ./backup.sh
# ./backup.sh: line 3: BACKUP_DESTINATION: set BACKUP_DESTINATION
```

The empty case matters. The colon in `:?` makes an empty value count as missing — which is what you want, since an empty destination is as useless as none. Without the colon (`${VAR?msg}`), an empty string would pass the check and fail somewhere deeper.

Validate before use

A value that arrived is not yet a value you can trust:

```
case $backup_interval in
  hourly|daily|weekly) ;;
  *) printf 'invalid BACKUP_INTERVAL: %s\n' "$backup_interval" >&2; exit 2 ;;
esac
```

Validate paths and allowed modes before use. A typo like `BACKUP_INTERVAL=dialy` should fail loudly at startup, not quietly select some accidental behavior three functions deep.

Two closing rules. Do not print secret configuration — paths and modes are fine to echo, tokens and passwords are not, because logs outlive credential rotations. And if you do adopt a config file one day, treat it as data to parse line by line, never as code to execute.

Write structured log lines

Add timestamps, levels, and one event per line so human and automated readers can follow a run.

Automation often runs without a terminal. When a backup job fires at 03:00, nobody is watching — the log is the only witness. Logs must show when, what, and whether an operation succeeded, in a shape both a human and `grep` can process.

A small logger function

Create a small logger:

```
log() {
  local level=$1
  shift
  printf '%s level=%s message=%q\n' "$(date -u +%FT%TZ)" "$level" "$*"
}

log info 'backup started'
# 2026-08-03T15:12:09Z level=info message=backup\ started
```

Piece by piece: `shift` drops the level from the arguments so `$*` holds only the message words. `date -u +%FT%TZ` prints a UTC timestamp like `2026-08-03T15:12:09Z` — always log in UTC, or the first daylight-saving change makes your timeline lie. The `%q` format quotes the message so spaces and special characters survive as one parseable field.

One event per line, `key=value` fields. That's the entire format. `grep 'level=error'` finds failures, and a log shipper parses it without custom rules.

Use it through a run

```
log info 'backup started'
if tar -czf "$archive" -C /var/www .; then
  log info 'archive created'
else
  log error 'tar failed'
  exit 1
fi
```

Redirect output to a lab file and generate success and error events:

```
./backup.sh >> /tmp/backup.log 2>&1
grep 'level=error' /tmp/backup.log
```

Verify each event carries a timestamp and a level, and that the error path logged before exiting. A failure the log never mentions is the failure you'll debug blind.

Keep a run identifier when several jobs may overlap. Add `run=$$` — the shell's process ID — as a field on every line, and interleaved runs untangle instantly instead of reading as one confused story.

What must never be logged

Redact tokens, passwords, cookies, and private content before logging. The classic leak is logging a full `curl` command line, `Authorization` header included: the secret then lives in a world-readable file long after it was rotated. Log which endpoint you called and the status you got back, never the credential that authenticated you.

Retry with a budget

Retry transient failures with a maximum attempt count, delay, and final non-zero status.

Networks blip. An endpoint that failed at 03:00:00 is often fine at 03:00:05, and one retry saves a false alarm. But retries help with temporary failures only — unbounded loops hide outages and multiply load against a service that is already struggling. A retry needs a **budget**: a maximum attempt count, a delay, and a final non-zero status when the budget runs out.

Three attempts, growing delay

Retry a health request three times:

```
for attempt in 1 2 3; do
  if curl --fail --silent --max-time 5 https://example.org/ >/dev/null; then
    exit 0
  fi
  sleep "$attempt"
done

exit 1
```

The loop bounds everything. Success leaves immediately with `exit 0`. Each failure sleeps `$attempt` seconds — 1, then 2, then 3 — a small growing backoff that gives a recovering service room to breathe instead of hammering it on a fixed beat. When all attempts fail, the script falls through to `exit 1`.

`--max-time 5` caps each individual attempt, so a hanging connection can't stall the loop. The worst case is now arithmetic: three 5-second attempts plus 1+2+3 seconds of sleep, about 21 seconds, then a clean failure.

Verify the budget

Break the URL and measure the final duration:

```
time ./health-retry
# real    0m21.4s
printf '%s\n' "$?"
# 1
```

The caller should receive failure after the documented budget — not after an afternoon of silent looping. If `time` reports minutes instead, one of the attempts is missing its timeout.

What you must not retry

Reads and health checks are safe to repeat. State-changing operations are a different story. If a payment request times out, the server may have processed it anyway — only the response got lost. Retrying sends the payment twice.

Do not retry state-changing operations unless they are idempotent or protected by an idempotency key. Idempotent means running it twice leaves the same result as once: `rm -f /tmp/report.pdf` qualifies, a `POST /charge` does not. When you're not sure, fail after the first attempt and let a human decide — a missed run is cheaper than a duplicated side effect.

Prevent overlapping runs

Use a lock so a scheduled job does not start another copy while the previous run is active.

A backup that takes 40 minutes, scheduled every 30, will eventually run alongside itself. Backups, deployments, and cleanup jobs can corrupt state or overload systems when two copies overlap — two `tar` processes writing the same archive, two deployments moving the same symlink. The fix is a **lock**: the first run takes it, and later runs see it's held and stop.

Use flock on Linux

```
exec 9>/tmp/practical-job.lock
if ! flock -n 9; then
  printf '%s\n' 'job already running' >&2
  exit 75
fi
```

`exec 9>` opens file descriptor 9 writing to the lock file and keeps it open for the rest of the script. `flock -n 9` asks the kernel for an exclusive lock on that descriptor; `-n` means don't wait — fail immediately if another process holds it.

The kernel releases the lock when the process exits, however it exits. Crash, Ctrl-C, `kill` — the lock disappears with the process. That is what makes `flock` better than the homemade "write a pidfile and check it" approach: a stale pidfile from a crashed run blocks every future job until a human deletes it.

Exit status 75 is the conventional `EX_TEMPFAIL` — temporary failure, try again later. It tells a scheduler this wasn't an error; another copy was doing the work.

Verify the lock holds

Hold the first process open and start a second:

```
./nightly-backup &      # first run, holds the lock
./nightly-backup
# job already running
printf '%s\n' "$?"
# 75
```

The second should fail quickly with a distinct status. Then wait for the first run to finish and start again — it must acquire the lock and proceed normally. Both checks matter: a lock that never blocks is useless, and a lock that never releases is worse.

Know the lock's limits

A lock needs stable scope and cleanup behavior. The scope of this one is a single machine: the path identifies the job, so two different jobs need two different lock files, and on a shared machine consider a per-user directory instead of `/tmp` so another user can't squat on the name.

For distributed jobs — the same task scheduled on several servers — a local file lock is not enough. Those need coordination all the machines share, such as a database lock.

Check your understanding: reliable automation

Check the commands, decisions, safety boundaries, and failure cases from reliable automation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Automation projects

Build a backup wrapper, health checker, deployment gate, and tested maintainable script.

Build a backup wrapper

Validate source and destination, create a dated archive, verify it, and report a clear result.

This module assembles the course pieces into small real tools, starting with a backup wrapper. The wrapper coordinates existing tools: validate the inputs, run `tar`, verify the result, report clearly. It should not pretend that creating an archive is a complete offsite backup.

The core operation

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

source_directory=${1:?source directory required}
destination=${2:?destination required}
archive="$destination/backup-$(date -u +%Y%m%dT%H%M%SZ).tar.gz"

tar -czf "$archive" -C "$source_directory" .
tar -tzf "$archive" >/dev/null
printf 'created %s\n' "$archive"
```

The `${1:?...}` expansions reject missing arguments with a clear message and a non-zero status, so a half-configured run never starts. The timestamp from `date -u +%Y%m%dT%H%M%SZ` makes archive names sort chronologically, and two runs can't silently overwrite each other.

`tar -C "$source_directory" .` changes into the source before archiving, so the archive stores relative paths. The second `tar -tzf` lists the finished archive and discards the listing: reading every entry back pushes the whole file through gzip, so a truncated or corrupt archive fails here — not on restore day.

```
./backup-wrapper /var/www/site /mnt/backups
# created /mnt/backups/backup-20260803T151530Z.tar.gz
```

Test the unhappy paths

Test with spaces in paths, insufficient permissions, a full destination, and a restore into a temporary directory:

```
./backup-wrapper '/var/www/my site' /mnt/backups   # spaces must survive
./backup-wrapper /root/private /mnt/backups        # tar must fail loudly, status non-zero
mkdir -p /tmp/restore-test
tar -xzf /mnt/backups/backup-20260803T151530Z.tar.gz -C /tmp/restore-test
```

The restore drill is the step people skip. An archive nobody has ever extracted is a hope, not a backup. Diff a few restored files against the originals before you trust the wrapper with anything

real.

Know the wrapper's limits

Do not archive a changing database and assume consistency. `tar` reads files one at a time, so a database writing during the walk produces an archive whose files never coexisted — it may not restore at all. Use the database's backup mechanism, like `pg_dump` or `mysqldump`, and archive its output instead.

And a `.tar.gz` sitting on the same machine protects against mistakes, not disk failure. Copying archives somewhere independent is the missing half of the job.

Build a health checker

Check an HTTP endpoint, capture timing and status, and return a useful result for a scheduler or monitor.

A health checker answers one question on a schedule: is the service actually working? "The process is running" is not that answer, and neither is "the server returned something." A health checker needs a deadline and a small expected contract. It should not accept every HTTP response as success.

Check one page

```
#!/usr/bin/env bash

if curl --fail --silent --show-error --max-time 5 https://example.org/ | grep -q 'Example Dom
  printf '%s\n' 'healthy'
else
  printf '%s\n' 'unhealthy' >&2
  exit 1
fi
```

Each flag carries policy. `--fail` makes curl exit non-zero on HTTP errors like 500 — without it, a pretty error page counts as a successful transfer. `--max-time 5` is the deadline: a hanging service must become a fast failure, not an eternal wait. `--silent --show-error` drops the progress noise but keeps real error messages.

The `grep -q 'Example Domain'` part is the contract: the page must contain text the working service is known to render. A load balancer serving a blank 200 page fails the check, exactly as it should. `-q` suppresses grep's output because only its exit status matters here.

Verify each failure boundary

Break DNS, the expected text, and the timeout. Each fault should fail through a recognizable boundary:

```
curl --fail --silent --show-error --max-time 5 https://nonexistent.example.invalid/
# curl: (6) Could not resolve host: nonexistent.example.invalid

# now change the grep pattern to text that isn't on the page:
./health-check
# unhealthy
```

Also confirm `echo $?` prints `1` on every failure path. The exit status is what makes this script composable: cron, systemd, and monitoring agents branch on the number, not the words. A checker that prints `unhealthy` but exits zero reports success to every machine reading it.

Keep the checker itself safe

Avoid logging response bodies containing private data. A health endpoint that echoes user records into a monitoring log turns a probe into a leak — log the outcome and the status code, not the payload.

If the checker must authenticate, monitoring credentials need narrow read-only scope. A token that can only call the health endpoint is boring to steal; reusing an admin token in a script that runs every minute is not.

Build a deployment gate

Run checks in order, stop on failure, and record exactly which tested revision may proceed.

A deployment gate is a script that decides whether code may ship. It runs checks in order, stops at the first failure, and records exactly which revision passed. A deployment wrapper should make policy visible rather than hide a chain of commands behind one optimistic message.

Gate a Node project

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

revision=$(git rev-parse HEAD)
printf 'testing %s\n' "$revision"

npm test
npm run build
git diff --exit-code

printf 'ready %s\n' "$revision"
```

`set -euo pipefail` is what turns a command list into a gate: the first failing command stops the script, so the final `ready` line prints only when everything above it succeeded.

`git rev-parse HEAD` captures the exact revision under test, and both messages name it. When someone later asks ”what did we actually verify?”, the answer is in the output, not in anyone's memory.

The sneaky check is `git diff --exit-code`. It exits non-zero when tracked files changed — which catches a build step that modified the working tree, like a formatter that rewrote sources or a generated bundle that should have been committed. If the tree changed during the run, you didn't test what you're about to ship.

Verify the gate closes

Introduce a failing test, a build error, and a generated change — one at a time:

```
./deploy-gate
# testing 4f2a9c81c2d0e7b356aa02cbf4f0b3a99d17c4e2
```

```
# ... npm test output ...
# Tests: 1 failed, 41 passed
printf '%s\n' "$?"
# 1
```

Confirm no later success message appears in any of the three cases. A gate that prints `ready` after a failure is worse than no gate, because people trust it and stop reading the output above it.

Test what you ship

The script exposes one more policy question: the artifact. This gate proves a revision passes checks — then many pipelines rebuild from scratch before deploying, and the second build isn't guaranteed to match the first. Do not deploy a fresh rebuild different from the artifact you tested when the pipeline can preserve one artifact. Build once, test that build, ship that same build: hand the tested archive or image forward instead of rebuilding and hoping.

Test and review the script

Run syntax checks, ShellCheck, failure cases, clean-environment tests, and a documented cleanup path.

A script is a program. It needs tests for inputs, failures, side effects, and the environment it assumes. Most shell scripts get none of that: they run once on the author's machine on a good day, then get trusted for years. This review pass closes the course projects.

Run static checks

```
bash -n script.sh
shellcheck script.sh
env -i PATH=/usr/bin:/bin bash script.sh
```

`bash -n` parses the script without running it — a pure syntax check that catches an unclosed `if` before anything executes.

ShellCheck is the linter for shell. It knows every classic from this course: unquoted expansions, `read` without `-r`, parsing `ls`. Each finding has a code you can look up:

```
shellcheck backup.sh
# In backup.sh line 12:
# tar -czf $archive -C $source_directory .
#          ^------^ SC2086: Double quote to prevent globbing and word splitting.
```

The third command is the environment test. `env -i` starts from an empty environment and hands the script only a minimal `PATH`. A script that works in your terminal but depends on your aliases, your `PATH` additions, or a variable you exported months ago fails here — exactly the way it will fail under cron at 03:00.

Exercise real failures

Static checks can't see behavior. Add a test directory with disposable fixtures and run the script against trouble: missing commands, interrupted work (Ctrl-C mid-run), weird filenames like `'has space.txt'` and `-dash.txt`, and repeated execution — a second run must not corrupt what the first produced.

For each case, check the three things every caller sees: exit status, standard error, and side effects on disk. A failure that exits zero is a bug. So is a success that leaves temporary files behind — the cleanup path deserves a test of its own.

Warnings are a conversation

Static analysis is guidance, not proof. Sometimes ShellCheck flags something you did on purpose, like an intentional word split. Understand a warning before suppressing it, and document deliberate exceptions:

```
# shellcheck disable=SC2086  # $TAR_FLAGS is intentionally split into words
tar $TAR_FLAGS -czf "$archive" .
```

A disabled check with a comment is a recorded decision. A disabled check without one is a future bug report with your name on it.

Check your understanding: automation projects

Check the commands, decisions, safety boundaries, and failure cases from automation projects.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/shell-scripting/.