LESSON WORKBOOK

# Linux Terminal 04 - Reliable Bash Scripts and Backups You Can Restore

## Linux Terminal

Study the lesson, work through the lab, and check your understanding with the quiz. Keep the reference sheet nearby as you practice.

Includes the complete lesson, guided lab, knowledge check, answer explanations, and quick reference.

# Lesson

## Linux Terminal 04 - Reliable Bash Scripts and Backups You Can Restore

**Track:** IT Foundations -> Linux operations **Prerequisites:** Linux Terminal 01-03: paths, quoting basics, pipes, exit statuses, and evidence-led troubleshooting **Format:** 40-minute lesson + 25-minute local lab **Outcome:** validate script inputs, handle expected failures, create a local archive, verify its digest, and prove that its files restore correctly.

### 1. The backup command said "completed"

In Terminal 02, you learned that a script is a sequence of commands. Now imagine saving your learning notes every evening. Your script prints "Backup completed," so you assume the notes are recoverable. A week later you need one file. The archive is missing a directory, or the copied files are from an earlier run.

The weakness was not necessarily the copy command. It was the promise attached to the output. A script should report success only after the checks that success actually requires.

Our running scenario is a small study folder containing ordinary notes, a filename with a space, and a hidden lesson marker. We will archive the folder, calculate an integrity digest, restore into a separate directory, and compare content. Then we will deliberately change a restored copy and prove the comparison detects it.

This is a local teaching utility for trusted, non-changing files. It is not a complete backup system for a running database, an entire operating system, or hostile directories. The boundary lets us learn the mechanics precisely before adding operational complexity.

### 2. Design the script's contract first

A useful contract answers four questions: what does the script accept, what does it create, what counts as failure, and what does it promise after success?

The supplied `safe-backup.sh` accepts two arguments: an existing source directory and a **new** backup directory whose parent already exists. It creates an archive, a checksum file, and a completion marker. It rejects a missing source, an existing destination, and a destination nested inside the source.

Why reject nesting? A backup stored inside its own source may be included in later archives and grow unexpectedly. Why reject an existing destination? A teaching script should not silently replace the only previous copy. A new directory per run makes each result easy to inspect.

The script reports `CREATED`, followed by a reminder that restore verification is separate. The marker says that the archive and digest were created. It does not claim a successful restore or

protection against disk failure. That language is part of the interface.

## 3. Quote values that represent one argument

Consider a source directory named `study notes`. Unquoted expansion can turn one path into multiple words. Double quotes preserve it as one argument while still expanding the variable:

```
printf '%s\n' "$cct_source"
tar -cf "$cct_archive" -C "$cct_source" .
```

Single quotes are useful for literal text, such as a format string. Double quotes are useful when a value should expand but remain together. Prefer `printf '%s\n' "$value"` over treating arbitrary text as a format string. Bash quoting reference

This lesson uses a `cct_` prefix for task variables so they do not overwrite common shell or system variables. The source is resolved to a physical absolute path. The destination's parent is resolved separately before its new name is appended.

Do not parse `ls` output to enumerate backup inputs. Filenames can contain spaces and other characters that make line-oriented parsing unreliable. Archiving `.` from a specific source directory includes the hidden files too, without asking the shell to expand a `*` wildcard.

## 4. Exit status is part of the result

A zero exit status usually means a command completed successfully; a nonzero status signals a different outcome. The meaning depends on the command. For example, `cmp` distinguishes equal files, different files, and errors. A "files differ" result is exactly what an intentional negative test should produce.

Handle an expected failure explicitly:

```
if cmp -s original.txt restored.txt; then
  printf 'Contents match\n'
else
  cct_status=$?
  printf 'Comparison returned %s\n' "$cct_status"
fi
```

Capture `$?` immediately. The next command changes it. If a command is prefixed with `!`, the status seen by the surrounding condition is inverted; do not invert it and then assume `$?` still holds the original failure code.

Error output belongs on standard error. Progress and intended results can use standard output. The distinction lets a caller save a useful result while still seeing a failure. A clear message such as "source must exist" is more actionable than a final "done" after an earlier error scrolled away.

## 5. Shell options help, but checks still matter

`set -u` treats many unset-variable expansions as errors. `set -o pipefail` makes a pipeline's result reflect a failing component instead of only its final command. They reduce some surprises, but they do not validate the meaning of your inputs.

You will often see `set -e`, which exits on many unhandled failures. It has exceptions, including conditions and parts of command lists. A function called in a tested context can also behave differently than a beginner expects. Learn those semantics before relying on it as a complete error policy. Bash set options, pipeline status rules

Our backup utility uses explicit checks for critical steps. If archive creation, listing, finalization, or hashing fails, it reports the failure and exits. The runner uses stricter options while putting intentional negative tests inside `if` statements. Read both files and notice where the responsibilities differ.

A syntax check with `bash -n` catches parsing errors before execution. It does not prove filenames, permissions, disk capacity, or restore behavior are correct. Those need meaningful tests.

## 6. Temporary data needs an owner and a boundary

The lab creates a fresh private directory with `mktemp -d`, then puts its fixtures inside. A fixed location such as `/tmp/my-backup` can collide with another run or existing files. A generated name avoids that simple collision. GNU mktemp reference

The script also sets `umask 077`, reducing access granted to newly created files and directories. That does not encrypt the archive. It does not change permissions on unrelated existing files. It is one useful default for newly created local artifacts.

Cleanup is installed through an `EXIT` trap. Interrupt and terminate handlers exit with appropriate statuses, allowing the exit cleanup to run. No trap can clean up after SIGKILL or a machine power loss, so temporary artifacts can remain after those events. Bash trap reference

Our cleanup lists only exact fixture files, then removes empty directories. It uses no recursive deletion. If something unexpected appears, the last directory removal fails and leaves the material for inspection. That behavior favors an explainable leftover over a broad cleanup command.

## 7. Create the archive in stages

The utility uses a new destination directory as a container for one backup. It creates `archive.tar.partial`, checks that the archive can be listed, then renames it to `archive.tar`.

The central command is:

```
tar -cf "$cct_backup/archive.tar.partial" -C "$cct_source" .
```

`-c` creates, `-f` names the archive, and `-C` changes the input directory for archiving. The final `.` selects that directory's content, including dotfiles. The archive stores relative member names so the later restore can target a separate location. GNU tar manual

Listing with `tar -tf` checks that tar can read the archive structure. It is a useful early check, but it does not prove every restored file matches the source. A readable table of contents and a successful restore are different evidence.

The utility keeps incomplete output on a failed creation for inspection. It writes its completion marker only after archive finalization and digest creation. Multiple files and a marker do not form a production transaction; a power failure or disk problem still requires careful recovery procedures.

## 8. An integrity digest has a specific job

SHA-256 produces a digest derived from the archive's bytes. Save it after creating the archive, then check it when reading or transferring the archive later.

On a common Linux installation:

```
sha256sum archive.tar > SHA256SUMS
sha256sum -c SHA256SUMS
```

On macOS, the usual equivalent is:

```
shasum -a 256 archive.tar > SHA256SUMS
shasum -a 256 -c SHA256SUMS
```

The supplied utility selects whichever supported tool is available. GNU checksum documentation, Perl shasum reference

A matching digest means the archive matches the digest you are checking. It does not prove the original source was complete, that the application can use the files, or that a malicious actor did not replace both archive and checksum. Protect and store verification information appropriately in real systems. For our local exercise, the digest is one integrity check between creation and restore.

## 9. Prove the restore

Restore into an empty directory you created for the test:

```
tar -xf "$cct_backup/archive.tar" -C "$cct_restore"
```

Here `-x` extracts. This exercise extracts only the archive the runner just created from its own fixtures. Do not use arbitrary downloaded archives as interchangeable lab inputs.

Next, compare the original and restored regular files with `cmp`. Silence with exit status zero means the compared contents match. A recursive `diff` provides an additional directory-level comparison for this small fixture. Neither is a complete test of ownership, access controls, extended attributes, every symlink behavior, or application consistency. Those require an expanded

design.

Our test includes `study plan.txt` to exercise spaces and `.lesson` to exercise hidden names. These are meaningful boundary cases because weak shell scripts often mishandle them. After the good restore passes, the runner modifies only the restored notes and expects `cmp` to report a difference.

That negative test proves your verification can fail when content actually differs. A test that always prints "PASS" is only decoration.

## 10. Run the supplied lab

Requirements: Bash 3.2+, tar, common file utilities, and either `sha256sum` or `shasum`. No internet, cloud account, scheduled task, system configuration change, or administrator privileges are needed.

From this lesson's directory:

```
bash -n lab/safe-backup.sh
bash -n lab/run-lab.sh
bash lab/run-lab.sh
```

The runner creates fixtures, invokes the utility, checks the digest, restores, compares, and tests four failure boundaries. Expected stable output includes:

```
archive.tar: OK
PASS: archive digest matches and all three files restore, including a space and a hidden name.
PASS: missing source rejected before output creation.
PASS: existing backup refused; no overwrite.
PASS: destination inside source rejected.
PASS: modified restored copy detected as different.
```

The temporary path varies. At exit the runner removes only the files it created and their empty directories. If you interrupt a run, the same cleanup is attempted. The standalone utility intentionally retains created backups; the lab runner removes its own demonstration copy after testing.

If a prerequisite is missing, the script reports it. Review the indicated tool's official installation instructions separately. The exercise itself installs nothing.

## 11. Know what must change for real operations

A local archive on the same disk does not survive that disk's failure. Repeated local copies do not automatically provide retention, encryption, off-site recovery, access control, monitoring, or reliable scheduling. A live database also needs its own consistency mechanism; archiving changing files may capture an unusable mixture of states.

The next design questions are therefore operational. How much work can be lost? How quickly must service return? Where is another protected copy stored? Who can restore it? How often do you rehearse restoration? These decisions define the backup requirement before you choose

more commands.

For a portfolio entry, include the script contract, actual lab output, and an explanation of one detected failure. State that the fixture restored correctly on your tested platform. Do not claim a production disaster-recovery capability from a three-file demonstration.

**Sources checked:** September 18, 2026. The scripts target Bash 3.2+ with common Linux/macOS tools, avoid GNU-only tar verification flags, and label checksum command differences. The tested fixture covers regular text files, a hidden name, and a space; broader filesystem metadata and live applications are outside scope.

# Knowledge check and answers

## Linux Terminal 04 - Knowledge Check

Eight questions. Suggested pass: 6/8, followed by a successful lab and an explanation of one negative test.

### 1. The archive command exits zero. What has not yet been demonstrated?

A. The command reported success. B. The archive's files restore correctly for the required use. C. The script ran at all. D. The shell can execute commands.

**Answer: B.** Creation success is one observation. A restore test and appropriate comparisons provide stronger evidence about recovery. A production application may need consistency checks beyond file content.

### 2. Why write `"$cct_source"` when passing a source path?

**Answer:** Double quoting preserves the expanded value as one argument instead of allowing ordinary word splitting and pathname expansion. A source path containing a space should remain one path. Quoting does not replace validating the path's meaning or permissions.

### 3. Why reject a backup destination inside its source?

A. Linux cannot create nested directories. B. It can cause backup output to be included in the data being backed up or in later runs. C. Tar only supports one file. D. Hidden files require an external disk.

**Answer: B.** Separating source and output avoids accidental self-inclusion and growing archives. The teaching utility resolves paths and rejects the destination boundary before creating output.

### 4. What does `set -o pipefail` change?

**Answer:** The pipeline status reflects the rightmost failing component, or zero if all components succeed, instead of normally using only the final command's status. It does not stop every component immediately or validate the business meaning of their output.

### 5. Does `set -e` mean "every possible failure is handled correctly"?

**Answer: No.** Its behavior has exceptions, including tested conditions and parts of command lists. It also cannot identify semantic mistakes such as selecting the wrong source. Explicit checks, a clear contract, and meaningful tests remain necessary.

## 6. A SHA-256 check passes. Which statement is accurate?

A. The archive matches the digest being checked. B. The database is guaranteed consistent. C. The archive is encrypted. D. No one could replace both the archive and digest.

**Answer: A.** A digest checks byte integrity against the reference digest. It is neither encryption nor an independent completeness, application-consistency, or authenticity guarantee.

## 7. Why change the restored notes after a good restore?

**Answer:** The deliberate change is a negative test. The comparison should now detect different content. This confirms that the verifier can fail when it should, rather than always printing a success message. The source remains untouched.

## 8. What two limits would you explain before using this as a real backup service?

**Example answer:** "A copy on the same disk does not protect against that disk's loss. Archiving files from a changing database does not by itself establish an application-consistent backup."

**Explanation:** Other valid limits include retention, protected off-site copies, encryption, metadata requirements, permissions, monitoring, scheduling, and measured restore time. Choose requirements deliberately rather than assuming the lab already provides them.

# Quick reference

## Linux Terminal 04 - Reliable Scripts, Verified Restores

**Promise only what you checked:** created archive -> readable structure -> matching digest -> restored content -> application recovery, when required.

| Habit | Example or rule |
|---|---|
| Quote one path as one argument | `tar -cf "$cct_archive" -C "$cct_source" .` |
| Print arbitrary values safely | `printf '%s\n' "$cct_source"` |
| Check required arguments | Validate count, existing source, and new destination |
| Keep output outside input | Reject a backup directory inside the source |
| Catch missing values | `set -u` helps; validation still matters |
| Notice pipeline failures | `set -o pipefail` |
| Handle expected failures | Use `if command; then ... else ... fi` |
| Preserve status | Capture `$?` immediately in the relevant branch |
| Check syntax | `bash -n lab/safe-backup.sh` |
| Bound temporary work | Fresh `mktemp -d` directory; exact-name cleanup |

**Tar:** `-c` create · `-x` extract · `-t` list · `-f` archive filename · `-C` input/output directory. Archiving `.` from the source includes hidden files. Listing successfully does not prove full restore correctness.

**Digest check:**

```
# Common Linux
sha256sum archive.tar > SHA256SUMS
sha256sum -c SHA256SUMS

# Common macOS
shasum -a 256 archive.tar > SHA256SUMS
shasum -a 256 -c SHA256SUMS
```

Run each pair inside the archive directory. A digest checks bytes against the saved digest; it is not encryption or an application-consistency guarantee.

**Restore proof:** extract this lab's own trusted archive into its separate empty restore directory; compare regular-file contents with `cmp`; compare the small fixture tree with `diff -r`. Metadata and application recovery require additional checks.

**Run:** `bash lab/run-lab.sh` - Bash 3.2+, tar, common utilities, and one checksum tool. No internet or administrator access. Expected: digest OK, three files restored, missing source rejected, existing destination rejected, nested destination rejected, changed restore detected. The runner cleans only its own fixtures.

**Contract of the standalone utility:** `bash lab/safe-backup.sh SOURCE_DIR NEW_BACKUP_DIR`. Its parent must exist, destination must be outside source, source should be trusted and quiescent. Created backups remain; incomplete outputs remain for inspection on failure. Intended for learning, not whole-system or live-database backup.

**Production questions:** loss tolerance · restore time · protected copies · retention · encryption · access · consistency · monitoring · restore rehearsals.

**References checked September 18, 2026:** Bash options, Bash quoting, GNU tar, checksums.