macOS for Developers Course

Flavio Copes

Updated August 3, 2026

Contents

Build a reproducible Mac development environment with Apple tools, Homebrew, shell configuration, project runtimes, credentials, and recovery notes.

What you'll learn: Set up, verify, document, and rebuild a Mac development workstation without depending on hidden state or unsafe shortcuts.

Know your Mac

Record the operating system, architecture, developer directory, package-manager prefix, and compatibility boundaries.

Record the workstation baseline

Capture the macOS build, hardware architecture, computer identity, and available storage before changing the development environment.

A setup is easier to debug when you know where it started. Six months from now, a build will break after an update, and you will want to answer "what was on this Mac when it worked?" from a file, not from memory.

Before installing tools, record the operating-system build, processor architecture, host name, and free storage.

Capture four values

Run these commands in Terminal:

```
sw_vers
uname -m
scutil --get ComputerName
df -h /
```

Typical output on an Apple silicon Mac:

```
ProductName:        macOS
ProductVersion:     15.5
BuildVersion:       24F74
arm64
Flavios-MacBook-Pro
Filesystem      Size    Used   Avail Capacity  Mounted on
/dev/disk3s1s1  926Gi   9.8Gi  704Gi     2%    /
```

`sw_vers` describes macOS. The `BuildVersion` line matters more than the marketing version: two Macs can both say "15.5" and still run different builds.

`uname -m` reports the architecture of the current system. You will see `arm64` on Apple silicon and `x86_64` on an Intel Mac. Almost every install decision in this course depends on this value.

`scutil --get ComputerName` gives the machine a name you can reference in notes. `df -h /` tells you how much room the boot volume has left before you start pulling down SDKs and container images. Passing `/` keeps the output to one line; without it, `df` lists every mounted volume.

Store the baseline

Create a small baseline file in a private setup repository:

```
mkdir -p ~/setup
{ date; sw_vers; uname -m; scutil --get ComputerName; df -h /; } > ~/setup/baseline-2026-08-0
```

Date the filename. After each macOS upgrade, write a new file instead of overwriting the old one. The sequence of baselines becomes a timeline you can line up against "when did this start failing?".

One warning: keep serial numbers and other unique hardware identifiers out of a public repository. `system_profiler SPHardwareDataType` prints the serial number and provisioning identifiers. Collect it locally if you want, but treat that output as private.

The common mistake is skipping this lesson because the machine works today. The baseline costs one minute now. Reconstructing it later, from a machine that has already changed, is guesswork.

Distinguish hardware and process architecture

Tell whether the Mac and the current process use arm64 or x8664 before installing native dependencies.

An Apple silicon Mac can run native arm64 processes and, through the **Rosetta** translation environment, some x86_64 processes built for Intel. That flexibility hides a trap: the hardware architecture and the current process architecture are related, but they are not always identical.

Your Mac can be arm64 while your terminal, and every tool it launches, runs translated as x86_64. Everything you install from that terminal inherits the wrong architecture.

Inspect both views

Run these three checks:

```
uname -m
sysctl -in sysctl.proc_translated 2>/dev/null || true
file "$(command -v node)"
```

In a native shell on Apple silicon you get:

```
arm64
0
/opt/homebrew/bin/node: Mach-O 64-bit executable arm64
```

`uname -m` reports the architecture the current process sees. `sysctl.proc_translated` is the real tell: `0` means native, `1` means the shell is running under Rosetta. On an Intel Mac the key does not exist, which is why we silence the error and continue.

`file` inspects a specific binary. A universal binary lists both architectures; a single-architecture build lists one. This is how you check whether a tool someone shipped you is Intel-only.

How a shell ends up translated

The classic cause is the **Open using Rosetta** checkbox on Terminal or iTerm in the Finder's Get Info panel. Someone enables it to work around one stubborn dependency, forgets it, and from then on `uname -m` reports `x86_64` on an arm64 Mac.

You can also enter a translated shell on purpose:

```
arch -x86_64 zsh
uname -m   # x86_64
```

That is useful for testing, and dangerous as a default.

Why the mismatch hurts

A translated shell can install tools into a different prefix and build native dependencies for the wrong architecture. The failure shows up later and far away: an npm module compiles fine, then Node

refuses to load it with `mach-o file, but is an incompatible architecture (have 'x86_64', need 'arm64')`.

When you see that error, don't reinstall packages first. Check `sysctl.proc_translated` in the shell that did the install. My advice is to open a native terminal unless one project explicitly requires translation, and to keep any Rosetta work in a clearly labeled separate terminal profile.

Inspect the active developer directory

Find the Apple developer tools selected for command-line builds and recognize when Xcode changes the selected path.

Apple command-line builds use one **active developer directory**. Every compile that touches Apple tooling — a C extension for Python, a native npm module, a Ruby gem — resolves its compiler and SDK through that single path.

It may point to the standalone Command Line Tools package or to an installed Xcode application. The two ship different SDK versions, so knowing which one is active explains a lot of "works on my machine" build differences.

Check the selection

```
xcode-select --print-path
xcrun --find clang
clang --version
```

With only the Command Line Tools installed you get:

```
/Library/Developer/CommandLineTools
/Library/Developer/CommandLineTools/usr/bin/clang
Apple clang version 17.0.0 (clang-1700.0.13.3)
```

With Xcode installed, the first line usually reads `/Applications/Xcode.app/Contents/Developer` instead. Installing Xcode after the Command Line Tools often flips the selection without asking you.

`xcrun` resolves tools inside the selected developer directory. That is why `xcrun --find clang` and plain `command -v clang` can disagree: the first follows the selection, the second follows your PATH.

When the selection breaks

After a macOS upgrade, the tools sometimes vanish while the selection stays. The symptom is loud:

```
xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools),
missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun
```

That is a missing package, not a wrong selection: reinstall the Command Line Tools. If the path instead points to an Xcode you deleted, that is a wrong selection: fix it with `sudo xcode-select -s /Library/Developer/CommandLineTools` or reset to the default search order with `sudo xcode-select --reset`.

Record it, don't thrash it

Record the `xcode-select --print-path` output with build reports and in your setup notes. When a native build fails on one machine and passes on another, comparing this path is the first diff to run.

Do not switch it globally just to silence an error before you know which project needs which SDK. The selection is machine-wide: pointing it at a different Xcode to fix project A can quietly break project B's builds the same afternoon.

Write a compatibility note

Turn architecture, operating-system, and tool observations into a short compatibility record for one real project.

You now know how to read the machine: macOS build, architecture, translation state, active developer directory. This lesson turns those observations into something a teammate can use.

A useful project note states what you tested, not what you assume will work. "Works on macOS" is an assumption. "Built and tested on macOS 15.5, arm64, Command Line Tools clang 17" is evidence.

Create the note

Create `docs/macos-setup.md` in one real project you work on. Include the Mac architecture, macOS build, developer directory, runtime version, and any translated dependency. Next to each value, keep the exact command used to collect it, so the next person can compare instead of guessing:

```
# macOS compatibility - api-server

Last verified: 2026-08-03

| Value               | Command                      | Result                              |
| ------------------- | ---------------------------- | ----------------------------------- |
| macOS build         | `sw_vers`                    | 15.5 (24F74)                        |
| Architecture        | `uname -m`                   | arm64, native shell                 |
| Developer directory | `xcode-select --print-path`  | /Library/Developer/CommandLineTools |
| Node                | `node --version`             | v22.17.0                            |
```

Ten lines, and every line is reproducible.

Add the Known boundaries section

Add a section named **Known boundaries** for the edges you actually hit: Intel-only tools, minimum macOS versions, and required permissions.

```
## Known boundaries

- `legacy-report-tool` ships x86_64 only; runs under Rosetta, untested natively
- Requires macOS 14 or later (uses a newer network API)
- Terminal needs Full Disk Access to run the integration tests against ~/Library
```

Every entry here saves someone else the afternoon you already lost discovering it.

Test the note on a person

Ask another developer to compare the note with their machine. Give them the file and nothing else. Any question they cannot answer from the document is a missing setup requirement — add it and try again.

The failure mode to avoid is the note that describes intent instead of observation. ”Should work on Intel” is intent. If nobody ran the project on an Intel Mac, write ”untested on Intel” instead. A note that admits its gaps stays trustworthy; a note that papers over them gets ignored after the first time it's wrong.

Check your understanding: know your mac

Check the commands, decisions, safety boundaries, and verification steps from know your mac.

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

Take this interactive quiz online

Build the toolchain

Install Apple tools, manage Homebrew packages, pin project runtimes, and verify which executable will run.

Install and verify Apple command-line tools

Install Apple command-line tools only when needed, then verify the selected directory and compiler instead of trusting the installer dialog.

The **Command Line Tools** package provides compilers, SDK utilities, Git, and related development commands — without the full Xcode application. For most web and backend work it is all the Apple tooling you need, and Homebrew requires it before it will install anything.

You don't need to download anything manually. Start the Apple installer from Terminal:

```
xcode-select --install
```

macOS opens a dialog, downloads the package, and installs it. The command may instead report that the tools are already installed:

```
xcode-select: note: Command line tools are already installed.
Use "Software Update" in System Settings or the softwareupdate command to install updates
```

That message is a result, not an error. It means you can move straight to verification.

Verify, don't trust the dialog

The installer dialog saying ”done” is weak evidence. Verify the result with `xcode-select --print-path` and `xcrun --find clang`:

```
xcode-select --print-path
# /Library/Developer/CommandLineTools
xcrun --find clang
# /Library/Developer/CommandLineTools/usr/bin/clang
git --version
# git version 2.50.1 (Apple Git-155)
```

Three healthy signs: the selected directory exists, the compiler resolves inside it, and Git answers. If you run `git` or `clang` on a Mac without the tools, macOS itself offers to install the package — that prompt is this same installer wearing a different hat.

After updates, verify before reinstalling

macOS upgrades occasionally leave the tools broken or removed while the selection still points at them. The symptom is `invalid active developer path` on the first `git` command after the update.

After a macOS or Xcode update, repeat the verification before reinstalling anything. A missing selection and a missing package are different problems: a missing package needs `xcode-select --install` again, while a missing selection needs `sudo xcode-select --reset`. Reinstalling a multi-gigabyte package to fix a one-line selection problem is the classic time waster here.

Once the three verification commands pass, record their output in your setup notes. That snapshot is what you will compare against when a native build fails months from now.

Understand the Homebrew prefix

Use Homebrew's reported prefix instead of hard-coding an Intel or Apple silicon installation path.

Homebrew installs everything under one root directory, its **prefix**. The prefix is not the same everywhere: on Apple silicon it is `/opt/homebrew`, on Intel Macs it is `/usr/local`. Same tool, two layouts, and the difference is exactly the architecture split from the previous module.

Scripts that assume `/usr/local` or `/opt/homebrew` fail as soon as the environment changes — a teammate on older hardware, a CI runner, your own next Mac.

Ask Homebrew, don't guess

Homebrew will tell you its own paths:

```
brew --prefix
# /opt/homebrew
brew --repository
# /opt/homebrew
brew --cellar
# /opt/homebrew/Cellar
```

The prefix is where binaries get linked (`/opt/homebrew/bin`), the repository is Homebrew's own checkout, and the Cellar is where each formula's versioned files actually live.

When a script needs the location of something Homebrew installed, use command substitution:

```
export LDFLAGS="-L$(brew --prefix openssl@3)/lib"
```

`brew --prefix openssl@3` resolves the real, current path of that formula on this machine. The same line works on Intel and Apple silicon.

Use command substitution only where a tool genuinely needs an absolute prefix, like linker flags. Prefer `command -v` when you only need the executable selected by the current PATH:

```
command -v node
# /opt/homebrew/bin/node
```

The two-prefix trap

On Apple silicon, `/opt/homebrew/bin` is not on the default PATH. The installer tells you to add this line to `~/.zprofile`:

```
eval "$(/opt/homebrew/bin/brew shellenv)"
```

`brew shellenv` prints the exports for the correct prefix; `eval` applies them. Skip it and every `brew`-installed tool is invisible to new shells.

Here is the realistic failure: a Mac that once ran a translated x86_64 shell can end up with two Homebrew installations — an Intel one in `/usr/local` and a native one in `/opt/homebrew`. Both work, and PATH order silently decides which one serves each command. The tell is `brew --prefix` saying `/usr/local` on an arm64 Mac, or `file $(command -v node)` reporting `x86_64`. When you see that, pick the native install, migrate your formulas to it, and remove the Intel one deliberately.

Capture packages with a Brewfile

Describe command-line tools and applications as reviewable Homebrew Bundle input without treating the file as a complete machine backup.

After a few months, a Mac accumulates dozens of Homebrew packages, and nobody remembers why half of them are there. A **Brewfile** records formulas, casks, taps, and selected other packages in one plain-text file. It makes the intended workstation easier to review and rebuild.

Dump what you have

Create and inspect one deliberately:

```
brew bundle dump --file Brewfile
brew bundle check --file Brewfile
```

`dump` writes the currently installed packages; `check` confirms everything the file lists is satisfied. The output is readable Ruby-flavored text:

```
tap "hashicorp/tap"
brew "git"
brew "node"
brew "postgresql@16", restart_service: true
cask "visual-studio-code"
cask "raycast"
```

`brew` lines are command-line formulas, `cask` lines are GUI applications, `tap` lines are extra package sources.

Now edit it. The dump is a snapshot of what happened to be installed, including experiments you abandoned. Remove accidental personal applications before committing the file — a work setup file that installs your personal chat apps is a smell. What survives the edit is your declared toolchain, and the file belongs in your setup repository where changes show up in diffs and code review.

Rebuild from it

On a new machine, or after pruning:

```
brew bundle install --file Brewfile
brew bundle check --file Brewfile
# The Brewfile's dependencies are satisfied.
```

That final line is your verification. There is also `brew bundle cleanup --file Brewfile`, which lists installed packages the file does not mention — useful as a drift report. It only uninstalls when

you add `--force`, so run it bare first and read the list.

Know what it does not do

`brew bundle` installs declared software, but it does not restore application data, licenses, credentials, or operating-system settings. Your database contents, your SSH keys, your app preferences: none of that lives in Homebrew's world.

The realistic failure is treating the Brewfile as a backup, wiping a Mac, and discovering the difference afterward. Treat it as one column of your recovery plan — the "rebuildable" column — and inventory the rest separately, which is exactly what a later module does.

Pin and verify project runtimes

Declare runtime versions per project and prove the shell resolves the intended executable before installing dependencies.

Projects often require specific Node.js, Python, Ruby, or other runtime versions. One global Node works until the day you maintain two projects that disagree, and upgrading for one breaks the other.

The fix has two halves: a version manager that can hold several runtimes side by side (nvm, mise, and similar tools), and a **version file** committed to each project. The file gives tools and teammates one visible requirement instead of tribal knowledge.

Declare the version in the repo

With nvm the file is `.nvmrc`:

```
echo "22.17.0" > .nvmrc
nvm use
# Found '/Users/flavio/dev/api-server/.nvmrc' with version <22.17.0>
# Now using node v22.17.0 (npm v10.9.2)
```

Other managers read their own files — mise and asdf use `.tool-versions` — but the principle is identical: the requirement lives in the repository, next to the code that needs it.

Record the version mechanism in the project README too. "Run `nvm use` before installing" is one line, and it saves every new contributor the same confused half hour.

Prove what the shell resolves

Declaring a version is not the same as running it. After activating your chosen version manager, verify resolution:

```
node --version
# v22.17.0
command -v node
# /Users/flavio/.nvm/versions/node/v22.17.0/bin/node
type -a node
# node is /Users/flavio/.nvm/versions/node/v22.17.0/bin/node
# node is /opt/homebrew/bin/node
```

`command -v` shows the winner. `type -a` exposes shadowed installations — every `node` on the PATH, in order. That second line in the output is a Homebrew Node waiting to take over in any shell where the version manager didn't run.

That is the realistic failure: `node --version` prints the right number in your terminal, but a script, cron job, or editor launches a shell without the manager's setup, silently gets the Homebrew Node, and installs native modules against the wrong version. When dependency builds fail mysteriously, run `command -v node` in the failing context first.

Last piece of advice: avoid a global upgrade that silently changes every project at once. Bump versions per project, through the version file, in a commit that can be reviewed and reverted.

Check your understanding: build the toolchain

Check the commands, decisions, safety boundaries, and verification steps from build the toolchain.

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

Take this interactive quiz online

Shape the shell

Control Zsh startup files, PATH order, environment variables, aliases, functions, and project-specific configuration.

Map Zsh startup files

Choose the correct Zsh startup file by distinguishing login, interactive, and every-shell configuration.

Zsh reads different files for different shell modes, and most configuration problems on macOS trace back to a line living in the wrong one.

The three files that matter:

- `.zshenv` — read by **every** zsh process, including scripts.
- `.zprofile` — read by **login** shells, once per session.
- `.zshrc` — read by **interactive** shells, the ones with a prompt.

They run in that order when all apply. macOS has one quirk worth memorizing: Terminal opens each new window as a login *and* interactive shell, so both `.zprofile` and `.zshrc` run every time. On Linux, login shells are rarer; on a Mac they are the norm.

See which mode you are in

Inspect the current shell modes:

```
print -r -- "login=$options[login] interactive=$options[interactive]"
# login=on interactive=on
```

Run the same line inside a script and both flip to `off`. That script still read `.zshenv` — and nothing else. This is why "it works in my terminal but not in the script" is almost always a startup-file question.

Put each thing where it belongs

Keep `.zshenv` minimal. It runs for every zsh invocation, so anything slow here taxes every script on the system, and anything that prints breaks tools that parse command output.

Use `.zprofile` for login environment setup: PATH additions, `eval "$(/opt/homebrew/bin/brew shellenv)"`, exported variables. Login runs once per session, so the cost is paid once and child shells inherit the result.

Use `.zshrc` for interactive prompts, aliases, and completion — things that only make sense when a human is typing.

```
# quick sanity check of what each file currently does
head -5 ~/.zshenv ~/.zprofile ~/.zshrc 2>/dev/null
```

Putting every command in `.zshrc` can slow interactive terminals, while putting interactive output in `.zshenv` can break scripts. The classic symptom of the first: a noticeable pause before every prompt appears. The classic symptom of the second: an `echo` in `.zshenv` corrupting the output of some automation that captured it.

If a tool's installer forces you to break these rules — some insist on editing `.zshrc` — document any exception with a comment saying which tool demanded it. Future you will want to know whether that line is load-bearing.

Build PATH deliberately

Add tool directories once, preserve a readable order, and inspect command resolution before blaming an installation.

PATH order decides which executable wins when several share a name. On a developer Mac, several always share a name: macOS ships a `python3`, Homebrew installs one, and a version manager may add a third. The one listed first in PATH is the one that runs.

Most PATH problems come from configuration written in a hurry. Repeatedly prepending directories creates duplicates and hides the reason a command was selected.

Read the PATH like zsh does

Zsh exposes PATH as the `path` array, which is much easier to read than the colon-separated string:

```
print -l $path
```

```
/opt/homebrew/bin
/opt/homebrew/sbin
/usr/local/bin
/usr/bin
/bin
```

Then, for any suspicious command, list every candidate in resolution order:

```
type -a python3
# python3 is /opt/homebrew/bin/python3
# python3 is /usr/bin/python3
```

The first line wins. Everything below it is shadowed — installed, present, and never used by this shell.

Add directories deliberately

Make edits in `.zprofile`, once, and guard them:

```
# ~/.zprofile
```

```
typeset -U path
[ -d "$HOME/.local/bin" ] && path=("$HOME/.local/bin" $path)
```

`typeset -U path` tells zsh to keep the array unique, so a re-sourced file cannot stack duplicates. The `[ -d ... ]` guard means you add a directory only when it exists — a PATH full of dead entries is noise you will debug around later.

Prepend when the new directory must override system tools (version managers rely on this). Append when it only needs to be reachable. Every prepend is a statement: "trust this directory over the OS." Make that statement on purpose.

When the wrong tool wins

The classic surprise: you install a new Python with Homebrew, but `python3 --version` still prints the old number. Nothing is broken. `/usr/bin/python3` is winning on order, or your shell cached the old location.

After changing PATH, restart the relevant shell and confirm the winner with `command -v`:

```
command -v python3
# /opt/homebrew/bin/python3
```

Check resolution first, reinstall never. Almost every "the installation is broken" report on a Mac is really "a different installation answered".

Separate environment from secrets

Use environment variables for configuration without turning shell startup files, process lists, logs, or repositories into secret storage.

Environment variables are useful process input: they configure tools without editing code, and every language can read them. They are also the most common place developers accidentally store secrets.

The distinction to hold onto: environment variables are a delivery mechanism, not a secure vault. Child processes inherit them, and diagnostic output can expose them. Every subprocess your shell spawns — build scripts, npm packages, editor plugins — sees the full environment of its parent.

Audit what you already export

Check names without printing values:

```
env | cut -d= -f1 | sort
```

```
EDITOR
HOME
HOMEBREW_PREFIX
PATH
STRIPE_SECRET_KEY
...
```

Scan the list for anything named like a credential. A `STRIPE_SECRET_KEY` sitting there means some line in your startup files exports it to every process you will ever run. Piping through `cut` keeps the values off your screen and out of your terminal scrollback while you audit.

Where each kind belongs

Keep non-secret defaults in project documentation and in startup files: `EDITOR`, feature flags, local port numbers. These are safe to commit, safe to print, safe to inherit.

Load secrets from a password manager, Keychain, or a protected local file excluded from Git. If you use a `.env` file per project, exclude it before creating it:

```
echo ".env" >> .gitignore
chmod 600 .env
```

And scope it: a secret loaded by one project's tooling, in that project's directory, is far smaller a target than a secret exported globally in `.zshrc`.

How the leak actually happens

The realistic failure chain: a token goes into `.zshrc` "temporarily". Months later you publish your dotfiles repository, or you paste `env` output into a GitHub issue to debug something unrelated. The token was inherited, printed, and shipped.

Never add `env` output to a bug report without review. And when a secret does leak, deleting the line or the comment is not enough — the value lives on in Git history and in whoever's cache saw the issue. Revoke the credential itself and issue a new one. That is why the storage decision matters up front: the cheapest leak is the one that cannot happen.

Create a small shell contract

Build a minimal, documented shell configuration and verify it in a fresh login and non-interactive shell.

You have mapped the startup files, built the PATH, and moved secrets out. This lesson assembles those pieces into a **shell contract**: a small, documented configuration you can verify, instead of a `.zshrc` that grew by accretion for five years.

Create one setup repository with focused files for PATH, interactive behavior, and project helpers. Focused files keep each change reviewable:

```
~/setup/zsh/
  profile.zsh      # PATH and exported environment (sourced from .zprofile)
  interactive.zsh  # prompt, aliases, completion (sourced from .zshrc)
  projects.zsh     # helper functions for your repositories
```

Your real `.zprofile` and `.zshrc` shrink to a couple of `source` lines each. When something misbehaves, you comment out one file, not one hundred lines.

Inside `projects.zsh`, prefer small functions over long aliases when arguments or error handling matter. An alias is text substitution; a function can validate input and fail loudly:

```
serve() {
  local port="${1:-3000}"
  [ -f package.json ] || { print -u2 "no package.json here"; return 1 }
  npm run dev -- --port "$port"
}
```

Verify the contract

A shell configuration has two clients: you at a prompt, and every script that runs without you. Test the two important contexts:

```
zsh -lic "command -v node; node --version"
zsh -fc "command -v git"
```

The first spawns a fresh login interactive shell (`-l` login, `-i` interactive, `-c` run this command) and exercises normal Terminal startup, with `.zprofile` and `.zshrc` applied. It should print your version-managed Node.

The second, with `-f`, skips your startup files entirely. The non-interactive test catches configuration that scripts cannot see: if a command only resolves in the first test, automation running outside your interactive setup will not find it.

Keep it quiet

Remove output and prompts from files read by automation. A `.zshenv` or profile that prints a greeting corrupts every tool that captures shell output — the failure looks like a parser bug in some unrelated program, hours away from the actual cause.

The habit that makes the contract durable: run both verification commands after every configuration change. Ten seconds, and you know both clients still get what they were promised.

Check your understanding: shape the shell

Check the commands, decisions, safety boundaries, and verification steps from shape the shell.

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

Take this interactive quiz online

Identity and local trust

Configure Git and SSH identities, use Keychain safely, manage local certificates, and review application permissions.

Separate Git identities by context

Verify repository author identity and use conditional Git configuration when work and personal projects require different accounts.

Git records author and committer identity in every commit, permanently. A global email can leak into the wrong repository when one Mac serves several contexts: your personal address in the company monorepo, or your work address signed onto a weekend open source contribution.

Once pushed, that history is effectively public within its audience. Prevention beats cleanup here, because cleanup means rewriting shared history.

Find out what Git will use

Inspect the effective value and its source:

```
git config --show-origin --get user.email
git config --show-origin --get user.name
```

```
file:/Users/flavio/.gitconfig   flavio@flaviocopes.com
file:/Users/flavio/.gitconfig   Flavio Copes
```

`--show-origin` is the useful part: it names the exact file each value came from. When identity is wrong, this tells you which layer — system, global, or repository — to fix.

Switch identity by directory

Use conditional includes for directory-based identities. In `~/.gitconfig`:

```
[user]
  name = Flavio Copes
  email = flavio@flaviocopes.com

[includeIf "gitdir:~/work/"]
  path = ~/.gitconfig-work
```

And in `~/.gitconfig-work`:

```
[user]
  email = flavio@acme-corp.com
```

Every repository under `~/work/` now commits with the work address; everything else keeps the personal one. The rule is enforced by directory layout, not by memory — which is the point, because memory is what failed in the first place.

Two details trip people up. The trailing slash in `gitdir:~/work/` matters: it makes the pattern match everything under that directory. And the condition is evaluated against the repository you are inside, so testing it from your home directory tells you nothing.

Verify before the first commit

Before the first commit in a new repository, confirm the effective configuration from inside that repository:

```
cd ~/work/new-api
git config --show-origin --get user.email
# file:/Users/flavio/.gitconfig-work  flavio@acme-corp.com
```

The realistic failure: you clone a work project to `~/dev/` instead of `~/work/`, the include never matches, and three weeks of commits carry your personal email. The five-second check above, run once after every clone, is the whole defense.

Configure SSH host aliases

Select the intended key for each Git host account without copying private keys or relying on whichever identity the agent tries first.

The previous lesson separated your Git author identity by directory. Authentication has the same problem one layer down: GitHub identifies you by whichever SSH key you present, and with a personal and a work account on one Mac, ”whichever” is doing a lot of work.

SSH host aliases let one service have separate personal and work identities. You invent a hostname

per identity, and each one pins its own key. The alias becomes the hostname in the Git remote while `HostName` keeps the real server.

Write the alias

Example `~/.ssh/config` entry:

```
Host github-work
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_ed25519_work
  IdentitiesOnly yes
```

`Host github-work` is the made-up name you will type. `HostName github.com` is where the connection really goes. `IdentityFile` names the key for this identity.

`IdentitiesOnly yes` is the line people omit and regret. Without it, the SSH agent offers every loaded key in order, and GitHub accepts the *first* valid one — often your personal key, even on a connection you aliased for work. The setting means: for this host, offer this key and nothing else.

Protect the configuration and private key permissions:

```
chmod 600 ~/.ssh/config ~/.ssh/id_ed25519_work
```

SSH refuses keys that other users could read, and the config file deserves the same treatment. No step here ever copies a private key anywhere — each identity's key stays where it was generated.

Test, then use it in remotes

Test with `ssh -T github-work`:

```
ssh -T github-work
# Hi flavio-acme! You've successfully authenticated, but GitHub does not
# provide shell access.
```

Read the username in that greeting carefully. It names the account your key actually mapped to — this is the verification step, and the greeting saying the *wrong* account is exactly the failure this lesson prevents.

Then use a remote such as `git@github-work:team/project.git`:

```
git remote set-url origin git@github-work:team/project.git
```

The remote now encodes the identity. Anyone (including future you) can read `git remote -v` and know which account this repository speaks as — no agent luck involved.

Store a local secret in Keychain

Use macOS Keychain for a developer secret while keeping retrieval narrow and preventing accidental command output.

Keychain stores passwords, keys, and certificates behind macOS access controls: encrypted at rest, unlocked with your login, with per-item prompts when an unfamiliar process asks. It is a better home for a local credential than `.zshrc` or a committed configuration file, because nothing inherits it and no dotfiles repository can accidentally publish it.

The command-line interface is the `security` tool.

Store a secret

Add a generic password item:

```
security add-generic-password -a "$USER" -s dev.example.token -w
# password data for new item: ********
# retype password for new item: ********
```

`-a` sets the account, `-s` sets the **service name** — the label you will query by. Leaving `-w` bare makes `security` prompt for the value interactively. That is deliberate: typing the token after `-w` on the command line would land it in your shell history, which defeats the purpose.

Give the item a specific service name. `dev.example.token` says which system it belongs to; a generic name like `token` guarantees confusion once you have five of them.

Retrieve it exactly when needed

A script can request one named value when it needs it:

```
security find-generic-password -a "$USER" -s dev.example.token -w
```

The `-w` flag prints only the secret value, so command substitution stays clean:

```
export EXAMPLE_TOKEN="$(security find-generic-password -a "$USER" -s dev.example.token -w)"
```

Run that line inside the script that needs the token, at the moment it needs it — not in `.zprofile`. Retrieval at use time keeps the secret out of every unrelated process, which is the ”narrow” part of this lesson's promise.

The output leak

The realistic failure is not theft, it's echo. Run a script with `zsh -x` for debugging and every expanded command — token included — lands in the trace. Do not run the retrieval command in shared logs or tracing mode, and review captured output before pasting it anywhere.

Two closing pieces of hygiene. Document how to recreate the item: the service name, the account, and where the value comes from, so a new Mac is a five-minute task. And keep a separate account-recovery path — the Keychain copy is a convenience, not the only copy of your ability to log in.

Review local trust and permissions

Audit certificates and privacy permissions added for development, then remove access a project no longer needs.

Development work accumulates trust. Local HTTPS tools install certificates so `https://localhost` stops warning. Terminals and editors request privacy permissions — Full Disk Access, Accessibility, screen recording — so one workflow can run. Debuggers and automation ask for more.

These changes outlive the experiment that created them. The certificate a tool installed for a prototype in March still shapes what your Mac trusts in December, long after you deleted the prototype.

Record grants when you make them

The cheap fix is a log. Record every added trust item and permission with its purpose, owner, and removal command or settings path:

```
## Trust and permission log

- 2026-06-12 - mkcert root CA in login keychain - local HTTPS for shop-frontend
  - remove: `mkcert -uninstall`
- 2026-07-02 - Terminal: Full Disk Access - integration tests read ~/Library
  - remove: System Settings > Privacy & Security > Full Disk Access
```

Without the log, a later audit shows you a list of grants and no way to tell which are load-bearing. With it, cleanup is mechanical.

Review after each project

Review Keychain Access and **Privacy & Security** after finishing the project. In Keychain Access, look through your login keychain's Certificates category for development roots you added. From the command line you can check what carries custom trust settings:

```
security dump-trust-settings
# SecTrustSettingsCopyCertificates: No Trust Settings were found.
```

That output is the healthy state for the user domain: no manually trusted certificates. Anything listed instead deserves a line in your log — or removal.

Then open System Settings, Privacy & Security, and walk Full Disk Access and Accessibility. Toggle off anything belonging to tools you no longer run.

The setup-guide trap

Here is the failure mode to refuse: a copied setup guide says ”just add the root certificate” or ”grant your terminal Full Disk Access” as step one, without saying why. A broad root certificate means your Mac silently trusts every website that certificate's key chooses to sign. Full Disk Access means every script your terminal runs reads everything your user can.

Do not install a broad root certificate or grant Full Disk Access because a copied setup guide says so. Verify the publisher, scope, and exact feature first — and if the guide cannot tell you which feature breaks without the grant, that is your answer.

Check your understanding: identity and local trust

Check the commands, decisions, safety boundaries, and verification steps from identity and local trust.

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

Take this interactive quiz online

Make it reproducible

Capture packages and dotfiles, test a clean setup, protect recovery data, and maintain the workstation over time.

Inventory rebuildable and personal state

Separate declarative setup, source code, application data, credentials, licenses, and caches before designing workstation recovery.

Imagine your Mac dies tonight. A Brewfile can rebuild packages. Git can restore committed source. That covers less of your machine than you think: neither restores uncommitted work, application databases, credentials, device settings, or license records.

Recovery planning starts with knowing which kind of state each thing is. Not backing up — inventorying. The backup strategy falls out of the inventory, not the other way around.

Three columns

Create an inventory with three columns: **rebuild**, **restore**, and **recreate manually**. Put every important workstation item in exactly one column.

- **Rebuild**: regenerated from a declaration. Homebrew packages, cloned repositories, language runtimes pinned in version files.
- **Restore**: unique data that must come from a backup. Local database contents, uncommitted branches, app data, documents.
- **Recreate manually**: things that should never be in a backup file. Credentials you re-issue, licenses you re-download from the vendor, permissions you re-grant.

A concrete starting point:

```
| Item                        | Column            | Source of truth        |
| --------------------------- | ----------------- | ---------------------- |
| CLI tools and apps          | rebuild           | Brewfile in setup repo |
| Project code (pushed)       | rebuild           | GitHub remotes         |
| Postgres dev databases      | restore           | nightly pg_dump        |
| ~/.ssh keys                 | recreate manually | generate new, re-add   |
| API tokens                  | recreate manually | issue new per service  |
```

The "exactly one column" rule is what makes the exercise honest. An item you cannot place is an item you do not understand yet — which is precisely what this inventory exists to surface.

Audit the gaps

Uncommitted work is the sneakiest restore item. Find it before you need to:

```
find ~/dev -maxdepth 1 -type d -exec sh -c 'cd "$1" && [ -n "$(git status --porcelain 2>/dev/
```

Any directory that prints has state existing only on this machine.

Two boundaries keep the inventory sane. Exclude caches and downloaded dependencies unless recovery time justifies them — `node_modules` and `~/Library/Caches` are rebuildable noise that bloats any backup. And protect credentials separately from the machine they unlock: a backup containing your SSH keys, restored onto a compromised or shared machine, turns one lost laptop into a wider incident.

The realistic failure: assuming everything is column one, wiping the Mac, and discovering the local databases and `.env` files were column two. The inventory costs an hour. That discovery costs the data.

Manage dotfiles with clear ownership

Track the small configuration files you understand without replacing an existing file or secret silently.

A dotfiles repository versions the configuration files you shaped earlier in this course — `.zprofile`, `.zshrc`, `.gitconfig`, `.ssh/config`. Track the ones you understand and actively maintain, not everything hiding in your home directory.

A dotfiles repository should make ownership obvious. For every file the repo manages, two questions need answers written down: which destination does it own, and how does it get there? List each managed destination and whether setup copies, links, or generates it:

```
| Repo file      | Destination     | Method   |
| -------------- | --------------- | -------- |
| zsh/zprofile   | ~/.zprofile     | symlink  |
| zsh/zshrc      | ~/.zshrc        | symlink  |
| git/gitconfig  | ~/.gitconfig    | copy     |
| ssh/config     | ~/.ssh/config   | generate |
```

Symlinks keep the file editable in one place. Copies suit files that drift per machine. Generated files handle templates with machine-specific values.

Never replace silently

The dangerous moment is installation on a machine that already has configuration. Before changing a destination, compare it and preserve an existing file:

```
test -e ~/.zshrc && diff -u ~/.zshrc dotfiles/zshrc || true
```

Empty diff: safe to link. Any output: the existing file has something your repo does not, and you decide what to keep before anything is overwritten.

Prefer an explicit installer that stops on conflict:

```
install_link() {
  local src="$1" dest="$2"
  if [ -e "$dest" ] && [ ! -L "$dest" ]; then
    print -u2 "conflict: $dest exists, resolve manually"
    return 1
  fi
  ln -sf "$src" "$dest"
}
```

The realistic disaster this prevents: you run a clever installer on your old Mac, it ”helpfully” force-links everything, and the `.zshrc` carrying two years of local fixes is gone. Stopping on conflict turns that into a message instead of a loss.

Keep secrets out

Dotfiles repositories tend to become public, sometimes years after creation. Do not commit tokens, host-specific identifiers, shell history, or the contents of Keychain. History files are a classic leak: they contain every command you typed, including the ones with passwords in them.

Remember that Git preserves deleted files in history — committing a secret and removing it next commit removes nothing. If it happens, revoke the credential; don't just delete the line.

Test setup in a clean context

Run the workstation instructions in a fresh account or disposable Mac environment and verify outcomes instead of trusting the original machine.

Your current Mac contains years of hidden state. A setup script can appear complete while depending on an old PATH entry, cached credential, or manually installed tool. On the machine where you wrote it, those dependencies are invisible — everything is already there, so nothing fails.

The only honest test runs where none of that state exists.

Get a clean context

Use a fresh local account or disposable test machine. A new macOS user account is the cheap option: System Settings, Users & Groups, add a standard user named `setuptest`. It shares the OS but starts with an empty home directory — no dotfiles, no Homebrew PATH wiring, no agents, no caches. For higher-stakes verification, a wiped spare Mac or a macOS virtual machine is closer to the real "day one" experience.

Two rules protect the test's honesty and your data. Follow only the written prerequisites — if you have to remember something, the document failed. And run the installer without secrets: a clean-context test needs no production credentials, and do not use a production account or irreplaceable data for the test.

Run it like a stranger

Log into the fresh account, open Terminal, and follow your runbook top to bottom. Record every prompt or missing assumption as you hit it:

```
## Test run - 2026-08-03, fresh account

- step 2: `git clone` asked for auth - doc assumes SSH key already exists
- step 4: `brew` not found in new shell - shellenv line missing from doc
- step 6: installer paused on a license prompt the doc never mentions
```

Each line is a bug in the documentation, found for free.

Verify outcomes, not vibes

"It seemed to work" is not a result. Verify commands, runtime versions, Git identity, and a sample project build:

```
command -v brew git node
node --version          # matches the project's .nvmrc?
git config --get user.email
cd ~/dev/sample-project && npm install && npm test
```

The realistic finding — and you should expect one — is a hidden dependency: some tool you installed manually in 2024 that no document mentions, which the sample build needs. That is the test succeeding. Add the missing piece to the setup, wipe the test account, and run it again until a stranger could finish it alone.

Write the workstation recovery runbook

Finish a tested recovery plan that rebuilds tools, restores protected data, verifies projects, and records future maintenance.

This is where the course's pieces become one document: the runbook you would follow on a replacement Mac, under time pressure, with your main machine gone.

Write the runbook in the order a replacement Mac needs it: secure the account, update macOS, install Apple tools, restore setup files, install packages, retrieve credentials, restore data, then verify projects. The order encodes dependencies — Homebrew needs the Command Line Tools, your setup repo clone needs Git, project verification needs everything above it. A runbook in the wrong order strands you mid-recovery.

```
# Workstation recovery - last rehearsed 2026-08-03 (2h 40m)

1. Sign in to Apple Account, enable FileVault, set firmware basics
2. Update macOS: System Settings > General > Software Update
3. `xcode-select --install`, verify `xcrun --find clang`
4. Clone setup repo, run dotfiles installer (stops on conflict)
5. `brew bundle install --file ~/setup/Brewfile`
6. Recreate credentials: new SSH keys → add to GitHub; re-issue API tokens
7. Restore data: databases from dumps, ~/dev uncommitted work from backup
8. Verify: sample project clones, installs, tests green
```

Make each step checkable

Add expected output for each verification and a stopping condition when something differs. Compare these two versions of step 3:

```
BAD:  3. Install the command line tools
GOOD: 3. `xcode-select --install`; then `xcode-select --print-path`
      must print /Library/Developer/CommandLineTools.
      If it errors: STOP, fix selection before Homebrew.
```

The second is executable by a stressed brain at 11pm. That is the reader you are writing for.

Rehearse, then keep it alive

An untested runbook is a guess with formatting. Time a rehearsal in a clean context — the fresh account from the previous lesson — and update the plan with every missing dependency the rehearsal exposes. Write the measured time at the top: knowing recovery takes three hours changes how you plan a broken-laptop day.

Then schedule maintenance. Review the Brewfile and permissions after major upgrades, because both drift: packages accumulate, and macOS updates reset or change permission behavior.

The failure mode is the runbook written once, rehearsed never, and trusted anyway — it fails exactly when you need it, which is the one moment you cannot debug it calmly. A reproducible workstation is maintained evidence, not a setup script written once.

Check your understanding: make it reproducible

Check the commands, decisions, safety boundaries, and verification steps from make it reproducible.

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/macos-for-developers/.