GIT & GITHUB SERIES · COURSE 2

Git & GitHub for Teams

Branching strategies, rebasing, code review, CI with Actions, and two real-world scenarios — force-push dangers and a repo gone messy.

10 Chapters

osztromok.com

CHAPTER 1: BRANCHING STRATEGIES

COURSE 2 · CH 1

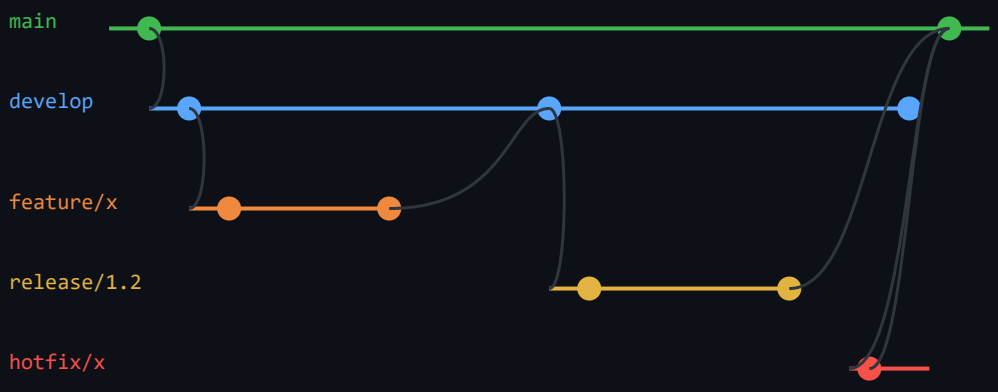
Branching Strategies

Git Flow, GitHub Flow, and trunk-based development — and how to pick the right one for a given project

Course 1 covered branches mechanically — create, switch, merge. What it didn't cover is the bigger question: **which branches should exist, what are they for, and when do they get merged?** That's a branching strategy — a set of conventions a team (or even a solo developer) agrees to follow, so collaboration stays predictable. This chapter covers the three most common approaches and how to choose between them.

Git Flow — The Original, Structured Approach

Git Flow was one of the earliest formalised branching models, built around scheduled releases. It defines several long-lived branch types, each with a specific job.



main = production · develop = integration · feature/* branches off develop · release/* stabilises before merging to main · hotfix/* patches production directly

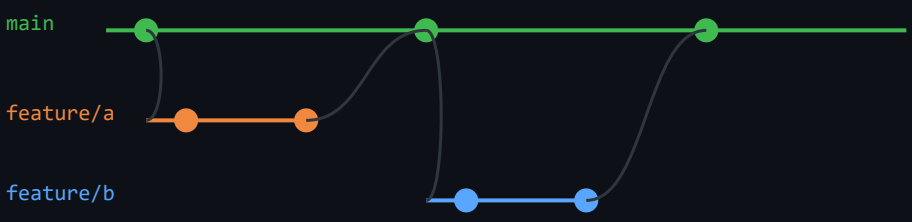
- **main** — always reflects what's currently in production, nothing else.
- **develop** — the integration branch where finished features accumulate between releases.
- **feature/*** — one branch per feature, branched from develop, merged back into develop when done.
- **release/*** — cut from develop when preparing a release; only bug fixes happen here while it stabilises, then merges to both main and develop.
- **hotfix/*** — branched directly from main for urgent production fixes, merged back to both main and develop.

GIT FLOW IS HEAVIER THAN MOST PROJECTS NEED TODAY

Designed in 2010 for software shipped in discrete, infrequent versions (think: installed desktop software, not a constantly-deployed website). For a project that deploys continuously — most web apps and SaaS products today — the release/* branch layer often adds process overhead without a corresponding benefit. Worth knowing because it's still common in larger, versioned software projects, but rarely the right default for a new project today.

GitHub Flow — Simpler, Continuous Deployment

GitHub Flow strips this down to almost nothing: one long-lived branch (`main`), and short-lived feature branches that get reviewed via PR (Course 1, Chapter 7) and merged straight back in.



main is always deployable · each feature branch merges straight back via a reviewed PR, then gets deployed

No `develop`, no `release/*`, no `hotfix/*` — every change follows the exact same path: branch → commit → open PR → review → merge → deploy. This matches how most modern web projects actually ship, including continuous deployment setups where every merge to `main` can trigger an automatic deploy (Chapter 8 covers the GitHub Actions side of this).

Trunk-Based Development — The Leanest Option

Trunk-based development pushes simplicity even further: developers commit directly to `main` (the "trunk") as much as possible, using very short-lived branches — often merged within hours, not days — or sometimes feature flags instead of branches entirely, to hide unfinished work from users without needing a separate branch at all.

- **Branches, when used, live for hours, not days.** The goal is minimising how far any branch diverges from main before merging back.
- **Feature flags hide incomplete work** — code can be merged to main and deployed while disabled behind a flag, separating "merged" from "visible to users" entirely.

- **Requires strong automated testing** to be safe — without a solid CI pipeline catching regressions before they reach main, frequent direct commits to main become risky fast.

> **TRUNK-BASED DEVELOPMENT ISN'T "NO BRANCHES" — IT'S "SHORT BRANCHES"**
>
> A common misconception is that trunk-based means committing straight to main with no review at all. In practice, most teams still use very short feature branches and PRs (matching GitHub Flow closely) — the defining trait is the *short lifespan*, not the total absence of branches.

Comparing the Three

🌳 Git Flow

Best for: versioned releases

Desktop software, mobile apps with app-store release cycles, anything shipped in discrete numbered versions rather than deployed continuously.

Most structure, most overhead — worth it only when that structure maps to how the software actually ships.

🚀 GitHub Flow

Best for: most web projects

Websites, SaaS products, internal tools — anything continuously deployed where "merged to main" and "live" are close together in time.

The sensible default for the vast majority of new projects, including everything else covered in this course.

⚡ Trunk-Based

Best for: mature CI/CD, larger teams

High-velocity teams with strong automated test coverage, where merge conflicts from long-lived branches become a genuine bottleneck.

Needs investment in testing infrastructure first — adopting this without solid CI is how main branches get broken.

Choosing for Your Own Project

QUESTION	IF YES →
Do you ship in discrete numbered versions?	Git Flow

QUESTION	IF YES →
Do you deploy continuously, every merge can go live?	GitHub Flow
Solo project or very small team?	GitHub Flow — simplest to maintain alone
Large team, strong CI, frequent merge conflicts a real pain point?	Trunk-based, with feature flags
Not sure / starting fresh?	GitHub Flow — easiest to adopt, easiest to explain to new contributors

DON'T OVER-ENGINEER THIS DECISION

The branching strategy is a convention, not a technical constraint — switching strategies later is entirely possible if a project's needs change. For almost any new project, especially a solo one, GitHub Flow is the right starting point: simple enough to need no documentation, and exactly what the rest of this series already assumes.

CHAPTER 1 QUICK REFERENCE

- **Git Flow** — main/develop/feature/release/hotfix branches; built for versioned, infrequent releases
- **GitHub Flow** — main + short feature branches + PRs; built for continuous deployment; the sensible default for most projects
- **Trunk-based** — very short-lived branches or direct commits to main, often paired with feature flags; needs strong CI first
- **Choosing:** versioned releases → Git Flow; continuous deploy → GitHub Flow; high-velocity team with strong tests → trunk-based
- **When unsure, default to GitHub Flow** — simplest to adopt and explain
- **Next chapter:** merge vs rebase — what's actually happening under the hood, and when to use each

CHAPTER 2: MERGE VS REBASE

COURSE 2 · CH 2

Merge vs Rebase

What's actually happening under the hood with each, and when to reach for one over the other

Course 1 introduced `git merge` as the way to bring a branch's work back together. There's a second tool for the same general goal — `git rebase` — and the two produce genuinely different history, not just different commands for the same outcome. Understanding that difference is what lets you choose deliberately instead of by habit.

The Setup — A Diverged Branch

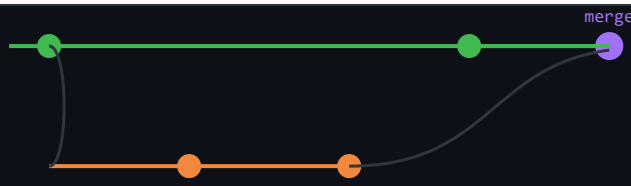
Both examples in this chapter start from the same situation: a feature branch with two commits, while `main` has gained one commit of its own in the meantime.

main gained c2 while feature was being worked on — both branches diverged from c1

Option A: git merge

Merging creates a brand-new commit with **two parents** — one pointing back into `main`'s history, one pointing back into `feature`'s history. Nothing about either branch's existing commits changes; the merge commit simply ties them together.

```
$ git switch main
$ git merge feature
Merge made by the 'ort' strategy.
```



A new merge commit ties both histories together — c1, c2, f1, and f2 are all still visible, exactly as they happened

Option B: git rebase

Rebasing takes a completely different approach: it **replays** your branch's commits, one at a time, as if they'd been written starting from the tip of `main` all along — rather than tying two histories together, it rewrites `feature` to look like it always branched off the newest point.

```
$ git switch feature
$ git rebase main
Successfully rebased and updated refs/heads/feature.
```

f1 and f2 are recreated as f1' and f2' — same changes, brand new commit hashes, now sitting cleanly after c2 in a single straight line

F1' AND F2' ARE NOT THE SAME COMMITS AS F1 AND F2

This is the single most important thing to understand about rebasing: even though the file changes are identical, rebasing creates entirely new commits with new hashes. The originals still technically exist in git's internal storage for a while (recoverable via `git reflog` — Course 3, Chapter 3), but as far as any branch or remote is concerned, `feature` now points at a different, rewritten history. This is exactly why rebasing a branch others have already pulled is dangerous — covered in the warning below.

Comparing the Results

merge	rebase

History: Branching, non-linear — shows exactly when and how branches diverged and came back together.

Safety: Always safe, even on shared/pushed branches — never rewrites existing commits.

Downside: Can produce a tangled-looking history with many merge commits on an active project.

History: Linear, clean — reads top to bottom with no branching/merging visual noise.

Safety: Safe only on branches nobody else has pulled yet — rewrites commit hashes.

Downside: Loses the literal record of when divergence happened; rewritten hashes can conflict with collaborators' copies.

When to Use Each

SITUATION	USE
Bringing a finished feature branch into main via a PR	merge (often via the PR's own "Merge" button — Course 1, Ch7)
Catching your local feature branch up with main before continuing work	rebase — keeps your branch's history clean as you go
The branch has already been pushed and others have pulled it	merge only — never rebase shared history
Cleaning up your own messy local commits before opening a PR	interactive rebase (full coverage in Course 3, Ch2)
Unsure, or working solo on a small project	merge — fewer ways to get into trouble

THE GOLDEN RULE, AGAIN: NEVER REBASE A BRANCH OTHERS HAVE ALREADY PULLED

This is the same underlying principle from Course 1, Chapter 8's revert-vs-reset rule, applied to rebasing: rewriting history that someone else has already built on top of creates a divergence between your rewritten version and their original — the foundation of the force-push problems covered later in this course (Chapter 9) and the disaster-recovery chapter in Course 3.

A common middle ground: rebase locally, merge for the PR

Many developers rebase their own feature branch against main repeatedly while working on it solo (keeping their personal history clean, before anyone else has seen it), then let the actual

PR merge into main happen as a normal merge — getting a clean personal workflow without ever rebasing anything that's been shared.

CHAPTER 2 QUICK REFERENCE

- **git merge** — creates a new commit with two parents; preserves exact history; always safe
- **git rebase** — replays your commits on top of the target branch; creates new commit hashes; linear, clean history
- **Critical distinction:** rebase rewrites commits — the originals are effectively gone from that branch's perspective
- **Golden rule:** never rebase a branch that's already been pushed and pulled by someone else
- **Common pattern:** rebase your own unshared branch to stay current with main; merge for the final PR into main
- **When unsure:** default to merge — it's never the wrong choice, just sometimes the less tidy one
- **Next chapter:** resolving conflicts properly — tools, strategies, and aborting safely when it goes wrong

CHAPTER 3: RESOLVING CONFLICTS PROPERLY

COURSE 2 · CH 3

Resolving Conflicts Properly

Tools that make conflicts easier to read, a real strategy for working through them, and how to abort safely when needed

Course 1, Chapter 10 introduced conflicts at the simplest level — two people, one file, manual marker editing. This chapter goes further: tools that make conflicts genuinely easier to read, a repeatable strategy for working through several at once, and what's different when a conflict happens during a rebase rather than a merge.

A Quick Recap of the Markers

```
<<<<<<< HEAD
your version of this section
=======
their version of this section
>>>>>>> branch-name
```

Everything between the markers is the disputed section — git has already merged everything else in the file automatically; only genuinely overlapping lines end up wrapped like this.

Better Ways to See a Conflict Than Reading Raw Markers

git diff during a conflict

Mid-conflict, plain `git diff` shows a special three-way view — useful for understanding what changed on each side before diving into the markers themselves:

```
$ git diff
diff --cc index.html
index a1b2c3d,9f8e7d6..0000000
```

VS Code's built-in conflict resolution

VS Code (and most modern editors) detect conflict markers automatically and offer inline buttons: **Accept Current Change**, **Accept Incoming Change**, **Accept Both Changes**, or **Compare Changes** in a side-by-side diff view. For most day-to-day conflicts, this is faster and less error-prone than manually editing marker text.

git mergetool

```
$ git mergetool
```

Launches a dedicated three-pane merge tool (configurable — VS Code, Meld, KDiff3, and others all work) showing "yours," "theirs," and the merged result side by side. Worth setting up if conflicts come up often enough that the inline editor view starts feeling cramped.

> **GIT STATUS DURING A CONFLICT TELLS YOU EXACTLY WHICH FILES STILL NEED ATTENTION**
>
> Mid-conflict, `git status` lists every file with unresolved conflicts under "Unmerged paths" — useful for tracking progress when a merge touches several files at once, so nothing gets missed before committing.

A Repeatable Conflict-Resolution Strategy

1. **Run git status first — see the full scope**
   Before touching any file, know how many are conflicted. Resolving one at a time on a multi-file conflict without this context risks losing track of what's left.

2. **Resolve the simplest files first**
   Building momentum on easy ones (often whitespace or formatting-only conflicts) before tackling genuinely complex logical conflicts keeps the process from feeling overwhelming.

3. **For each file: understand BOTH sides before choosing**
   Don't default to "mine" or "theirs" reflexively — read what each side was actually trying to accomplish. Sometimes the correct resolution is a genuine combination of both, not a pure pick.

4. **Stage each resolved file as you finish it**
   `git add <file>` immediately after resolving — it marks that file as done in git status, and you can see your remaining work shrink.

5 **Test before committing the resolution**

A conflict resolution that compiles/runs correctly is far more trustworthy than one that merely "looks right" — especially for logic conflicts, not just text conflicts.

Conflicts Are Slightly Different During a Rebase

Because a rebase (Chapter 2) replays commits one at a time, a conflict can occur on *any* of those commits individually — and you resolve them one at a time too, continuing the rebase after each.

```
$ git rebase main
CONFLICT (content): Merge conflict in index.html
Could not apply f1... Add hero section
# Resolve index.html, then:
$ git add index.html
$ git rebase --continue
# If another commit conflicts too, repeat the same process
```

🔀 Merge conflict

One conflict to resolve, then a single `git commit` finishes the merge.

🪨 Rebase conflict

Potentially one conflict *per replayed commit* — resolve, `git add`, `git rebase --continue`, repeat until done.

Aborting Safely

There is never any pressure to push through a conflict resolution you're not confident about. Both merge and rebase can be cancelled completely, returning you to exactly where you were before starting.

```
$ # Cancel a conflicted merge entirely
$ git merge --abort

$ # Cancel a conflicted rebase entirely (including any already-resolved steps)
$ git rebase --abort
```

```
$ # Cancel a conflicted pull (which is really fetch + merge under the hood)
$ git merge --abort # same command — a pull conflict is a merge conflict
```

--ABORT FULLY RESETS, IT DOESN'T SAVE PARTIAL PROGRESS

If you've resolved three out of five conflicted files and then abort, all five revert to their conflicted (or pre-conflict) state — there's no partial-save. If you're confident in some resolutions but stuck on others, it's often better to commit what you've got resolved in a safe spot (or just take your time) rather than abort and lose completed work.

Command Reference

COMMAND	WHAT IT DOES
`git status`	Lists files with unresolved conflicts under "Unmerged paths"
`git diff`	Shows a three-way diff view during an active conflict
`git mergetool`	Opens a dedicated visual merge tool for the conflicted files
`git add <file>`	Marks a conflict as resolved for that specific file
`git rebase --continue`	Proceeds to the next commit after resolving the current rebase conflict
`git merge --abort`	Cancels a conflicted merge (or pull) entirely, no partial save
`git rebase --abort`	Cancels a conflicted rebase entirely, no partial save

CHAPTER 3 QUICK REFERENCE

- **Better than raw markers:** your editor's inline conflict buttons, or git mergetool for a dedicated three-pane view
- **Strategy:** git status for scope → easiest files first → understand both sides → stage as you go → test before committing
- **Rebase conflicts** can occur per-commit — resolve, git add, git rebase --continue, repeat
- **git merge --abort / git rebase --abort** — fully cancel, return to the pre-conflict state, no partial save
- **A pull conflict is just a merge conflict** — same markers, same tools, same abort command
- **No time pressure** — aborting and asking a collaborator is always a legitimate option over guessing

- **Next chapter:** fork workflow vs shared-repo workflow — contributing to projects you don't have direct push access to

CHAPTER 4: FORK WORKFLOW VS SHARED-REPO WORKFLOW

COURSE 2 · CH 4

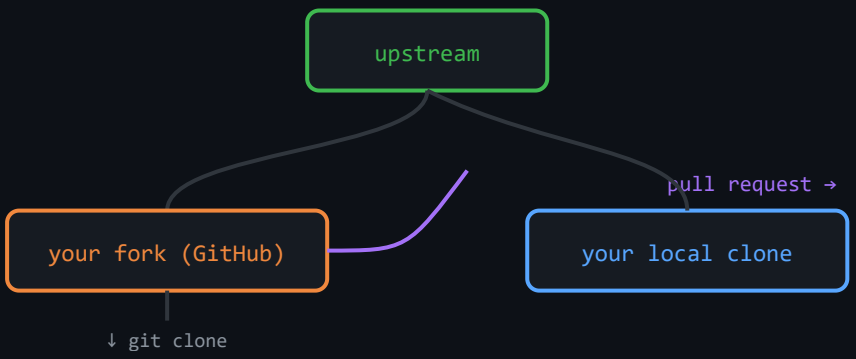
Fork Workflow vs Shared-Repo Workflow

Contributing to projects you don't have direct push access to — and when forking makes sense even on your own repos

Every workflow covered so far assumed you have push access to the repository — you branch directly off it, push your branch, and open a PR within the same repo. That breaks down the moment you want to contribute to someone else's project (an open-source library, a colleague's personal repo) where you don't have, and shouldn't have, push access. That's exactly what forking solves.

What a Fork Actually Is

A **fork** is your own complete copy of someone else's repository, created on GitHub, that you fully own and control — you can push to it freely, exactly like any repository you created yourself. The fork stays connected to the original ("upstream") repository in GitHub's interface, which is what makes opening a PR back to the original possible.



You fork upstream on GitHub → clone YOUR fork locally → push to your fork → open a PR from your fork back to upstream

The Fork Workflow, Step by Step

1. **Fork the repository on GitHub**
   The "Fork" button on any repo page creates a complete copy under your own account in seconds.

2 **Clone YOUR fork, not the original**

`git clone https://github.com/yourname/project.git` — note it's your username in the URL, not the original maintainer's.

3 **Add the original repo as a second remote, conventionally named "upstream"**

`git remote add upstream https://github.com/originalowner/project.git` — this is the second-remote scenario flagged back in Course 1, Chapter 5.

4 **Branch, commit, push to YOUR fork (origin)**

Exactly the Course 1 workflow — `git switch -c`, commit, `git push -u origin <branch>`.

5 **Open a PR from your fork's branch into upstream's main branch**

GitHub detects the fork relationship automatically and offers this as an option when opening a new PR.

Keeping your fork up to date

Your fork doesn't automatically stay in sync with upstream as the original project keeps moving — you periodically need to pull upstream's changes into your fork.

```
$ git fetch upstream
$ git switch main
$ git merge upstream/main
$ git push origin main # updates your fork on GitHub too
```

GIT FETCH VS GIT PULL, PROPERLY DISTINGUISHED

`git fetch` downloads a remote's commits without merging them into your current branch — it just updates what your local git knows about that remote. `git pull` is really `git fetch` + `git merge` combined. Using fetch explicitly (as above) lets you inspect upstream's changes with `git log upstream/main` before deciding to merge them in.

Fork Workflow vs Shared-Repo Workflow

🍴 Fork Workflow	🤝 Shared-Repo Workflow

Used when: contributing to a project you don't have push access to — open source, a colleague's personal project, a project where access is intentionally restricted.

Trust model: maintainers don't need to grant you any access at all — they just review and merge (or decline) your PR.

Used when: you're already a collaborator with push access — most internal team projects, everything in Course 1's branching and PR chapters.

Trust model: you're trusted to push branches directly; branch protection rules (Course 3, Ch6) still gate what reaches main.

FORKING YOUR OWN REPO CAN BE USEFUL TOO, EVEN WITH FULL ACCESS

Some teams deliberately use a fork-based workflow even internally — it keeps the main repository's branch list clean (no accumulation of personal branches from every contributor) and adds a clear boundary between "experimental personal work" and "the shared repo." Not the default choice, but a legitimate one for larger teams.

Command Reference

COMMAND	WHAT IT DOES
`git remote add upstream <url>`	Adds the original repo as a second remote, conventionally named upstream
`git fetch upstream`	Downloads upstream's commits without merging them in
`git merge upstream/main`	Brings upstream's changes into your local main branch
`git push origin main`	Pushes the now-updated main to YOUR fork (origin)

CHAPTER 4 QUICK REFERENCE

- **A fork** is your own complete, fully-owned copy of someone else's repo, linked to the original on GitHub
- **Fork workflow:** fork → clone YOUR fork → add upstream remote → branch/commit/push to origin → PR into upstream
- **upstream** = conventional name for the original repo's remote, alongside your own origin
- **git fetch vs git pull:** fetch downloads without merging; pull = fetch + merge combined
- **Use forks** when you lack push access; use the shared-repo workflow when you already have it

- **Keeping a fork current:** fetch upstream → merge upstream/main → push to origin, periodically
- **Next chapter:** stashing, cherry-picking, and tagging releases

CHAPTER 5: STASHING, CHERRY-PICKING, TAGGING

COURSE 2 · CH 5

Stashing, Cherry-Picking, and Tagging Releases

Three independent tools for three specific situations — pausing unfinished work, copying one commit elsewhere, and marking a release permanently

This chapter covers three commands that don't fit neatly into "branching" or "merging" but come up constantly once you're working on real projects: temporarily shelving unfinished changes, taking one specific commit and applying it somewhere else, and permanently marking a specific point in history as a release.

git stash — Pausing Unfinished Work

Imagine you're mid-edit on a feature, and an urgent bug needs fixing on a different branch right now. Your changes aren't ready to commit, but you need a clean working directory to switch branches. `git stash` sets your uncommitted changes aside temporarily, without committing them anywhere.

```
$ # Mid-edit, urgent bug needs attention on a different branch
$ git stash
Saved working directory and index state WIP on feature/login: a1b2c3d Add validation

$ git switch hotfix/urgent-bug
# ...fix the bug, commit, push, switch back...
$ git switch feature/login
$ git stash pop
On branch feature/login
Changes not staged for commit:
modified: login.js
```

Useful stash variations

```
$ # List all stashes — you can have more than one
$ git stash list
stash@{0}: WIP on feature/login: a1b2c3d Add validation
```

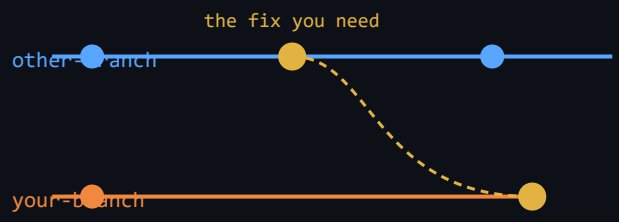
```
$ # Add a description, since "WIP on ..." gets unhelpful with several stashes
$ git stash push -m "half-finished password strength meter"

$ # Apply a stash WITHOUT removing it from the stash list
$ git stash apply

$ # Discard a stash you no longer need
$ git stash drop
```

STASH POP VS STASH APPLY

`git stash pop` applies the most recent stash AND removes it from the list. `git stash apply` applies it but leaves it in the list — useful if you want to apply the same stashed changes to more than one branch.

git cherry-pick — Taking One Specific Commit Elsewhere

Sometimes you need exactly one commit from another branch — not a full merge, not a rebase, just that one specific change — copied onto your current branch. `git cherry-pick` does precisely that.



cherry-pick copies just that one commit's changes onto your-branch as a brand-new commit, without bringing anything else from other-branch

```
$ # Find the commit hash on the other branch first
$ git log other-branch --oneline
9f8e7d6 Fix critical null-check bug
a1b2c3d Add unrelated feature

$ # Apply just that one commit to your current branch
$ git cherry-pick 9f8e7d6
[your-branch 4f3e2d1] Fix critical null-check bug
```

When cherry-picking genuinely earns its place

- **Backporting a fix.** A critical bug fix landed on `main`, but you also need it on an older release branch that's still being maintained — cherry-pick the one commit rather than merging all of main's other changes too.
- **Recovering a commit made on the wrong branch.** Accidentally committed to the wrong branch? Cherry-pick it onto the correct one, then remove it from the wrong one (Course 1, Chapter 8's reset tools).
- **Avoid it as a substitute for merging** — repeatedly cherry-picking individual commits between two branches that should really just be merged tends to create confusing, duplicated history over time.

> **A CHERRY-PICKED COMMIT GETS A BRAND-NEW HASH, JUST LIKE REBASING**
>
> Same underlying mechanism as rebase (Chapter 2) — cherry-pick replays a commit's changes as a new commit, not a copy of the original. The content is identical; the commit identity is not. This matters if you're tracking exactly which commits have landed where.

git tag — Marking a Release Permanently

A tag is a permanent, named pointer to one specific commit — unlike a branch (Course 1, Chapter 4), a tag's pointer never moves once created. Tags are how releases get marked: "this exact commit is version 1.2.0."

TAG TYPE	WHAT IT IS
Lightweight	Just a named pointer to a commit — no extra metadata. Quick, simple, fine for personal/internal markers.
Annotated	A full object with its own message, author, and date — recommended for actual releases, since it carries proper metadata (and can be cryptographically signed — Course 3, Chapter 6).

```
$ # Lightweight tag — quick personal marker
$ git tag v1.2.0-rc1

$ # Annotated tag — the recommended way to mark a real release
$ git tag -a v1.2.0 -m "Release 1.2.0 — login fixes and performance improvements"
```

```
$ # List existing tags
$ git tag
v1.0.0
v1.1.0
v1.2.0

$ # Tags don't push automatically — push them explicitly
$ git push origin v1.2.0
# Or push every local tag at once:
$ git push origin --tags
```

A TAG PUSHED TO GITHUB CAN BECOME A "RELEASE" WITH ONE EXTRA STEP

Pushing a tag is the underlying git operation; GitHub's "Releases" feature (Releases tab → Draft a new release, choosing an existing tag) layers release notes, attached binaries, and a changelog on top of a tag — purely a GitHub convenience feature, same underlying git tag either way.

Command Reference

COMMAND	WHAT IT DOES
`git stash`	Temporarily shelves uncommitted changes
`git stash pop`	Reapplies the most recent stash and removes it from the list
`git stash list`	Shows all currently stashed change-sets
`git cherry-pick <hash>`	Copies one specific commit's changes onto your current branch
`git tag -a <name> -m "msg"`	Creates an annotated tag marking the current commit
`git push origin --tags`	Pushes all local tags to the remote (tags don't push automatically)

CHAPTER 5 QUICK REFERENCE

- **git stash** — temporarily shelves uncommitted work so you can switch branches cleanly; pop to reapply
- **git cherry-pick <hash>** — copies one specific commit onto your current branch as a new commit

- **Cherry-pick is best for:** backporting fixes, recovering commits made on the wrong branch — not as a merge substitute
- **git tag** — a permanent pointer to one commit; unlike branches, never moves once created
- **Annotated tags (-a)** are preferred for real releases — carry message, author, date metadata
- **Tags don't push automatically** — use git push origin <tag> or --tags explicitly
- **Next chapter:** GitHub Issues, Projects, and linking pull requests to issues

CHAPTER 6: ISSUES, PROJECTS, LINKING PRS

COURSE 2 · CH 6

Issues, Projects, and Linking PRs

Tracking bugs and tasks on GitHub itself, organising work visually, and connecting pull requests to the issues they resolve

Everything covered so far happens at the code level — commits, branches, PRs. GitHub Issues and Projects sit one layer above that: tracking *what needs doing*, independent of any specific branch or commit, and Projects gives that a visual board to organise around.

GitHub Issues — Tracking Work and Bugs

An issue is a tracked item — a bug report, a feature request, a task — with its own discussion thread, labels, assignees, and status. Unlike a PR, an issue doesn't require any code at all; it's pure tracking and discussion.

#42 — Login button doesn't respond on mobile Safari

bug good first issue Opened by philipo · 3 comments

Tapping the login button on iOS Safari does nothing on the first tap — works on the second tap. Doesn't reproduce on desktop Chrome or Firefox. Suspect a touch event handler issue.

What makes a useful issue

- **A clear, searchable title** — "Login broken" is far less useful than "Login button doesn't respond on mobile Safari" when someone's searching for duplicates later.
- **Steps to reproduce, for bugs** — what you did, what you expected, what actually happened. The single biggest factor in how quickly a bug gets fixed.
- **Labels** — bug, enhancement, documentation, good first issue, and any custom labels a project defines. Makes filtering and triaging dramatically faster on any project with more than a handful of open issues.
- **Assignees** — who's actually working on it, separate from who opened it.

ISSUES WORK WELL EVEN ON A SOLO PROJECT

It's tempting to think issues are only useful for teams, but tracking your own bug list and feature ideas as issues — rather than a separate notes file — keeps everything in one searchable place, linked directly to the code that eventually resolves them.

Linking Pull Requests to Issues

GitHub recognises specific keywords in a PR's description (or even a commit message) and automatically links the PR to the referenced issue — and, crucially, can close that issue automatically the moment the PR merges.

KEYWORD (IN PR DESCRIPTION)	EFFECT WHEN THE PR MERGES
`Fixes #42`	Automatically closes issue #42
`Closes #42`	Same effect as "Fixes" — either works
`Resolves #42`	Same effect — three equivalent keywords
`Refs #42 / See #42`	Links the PR to the issue for context, but does NOT auto-close it

```
# In a commit message or PR description:
Fix login button touch handler on iOS Safari

Fixes #42
```

THIS WORKS FROM A COMMIT MESSAGE TOO, NOT JUST THE PR DESCRIPTION

Putting "Fixes #42" in the commit message itself (not just the PR's own description box) means the link survives even if commits get squashed (Course 1, Chapter 7) into a single commit on merge — the keyword travels with whichever commit message contains it.

GitHub Projects — Visual Work Organisation

A Project is a customisable board (or table, or roadmap view) that organises issues and PRs visually — most commonly as columns representing stages of work, cards moving left to right

as they progress.

TO DO	IN PROGRESS	DONE
#45 Add dark mode toggle	#42 Fix login button (PR open)	#40 Update README
#46 Improve error messages		#38 Fix typo in footer

Issues and PRs added to a Project automatically reflect their real status where GitHub can infer it — a merged PR linked to an issue can automatically move that issue's card to "Done," reducing the manual board-tidying that plagues most task-tracking tools.

When a Project is worth setting up

- **More than a handful of open issues** — below that, the issues list itself is usually enough; a board adds organisational overhead without much payoff for 3-4 open items.
- **Multiple contributors** needing a shared view of what's being worked on, to avoid duplicate effort.
- **Planning a release** — grouping issues by milestone or target version benefits from a visual board far more than a flat list.

> **A PROJECT BOARD IS NOT A SUBSTITUTE FOR GOOD ISSUE DESCRIPTIONS**
>
> A beautifully organised board full of vaguely-titled cards ("fix the thing", "improve stuff") is still unusable. The board organises issues; it doesn't replace the discipline of writing a clear issue in the first place.

CHAPTER 6 QUICK REFERENCE

- **Issues** — track bugs/features/tasks with discussion, labels, and assignees; no code required
- **Good issues:** clear searchable title, reproduction steps for bugs, appropriate labels, a real assignee
- **Linking keywords:** Fixes/Closes/Resolves #N auto-close the issue on merge; Refs/See #N just link without closing
- **Keywords work in commit messages too** — survives a squash merge, unlike text only in the PR description box
- **Projects** — a visual board/table organising issues and PRs by status, useful once there's enough work to warrant it
- **A board doesn't fix bad issue descriptions** — the discipline of writing clear issues still matters most

- **Next chapter:** code review — practical guidance as both the author and the reviewer

CHAPTER 7: CODE REVIEW

COURSE 2 · CH 7

Code Review

Practical guidance for both sides of the review — as the person whose code is being reviewed, and as the person reviewing it

Course 1, Chapter 7 covered the mechanics of leaving and replying to PR comments. This chapter is about doing review *well* — the judgment calls that separate a review that genuinely improves code from one that just creates friction, from both sides of the table.

As the Author — Making Your PR Easy to Review

1. **Keep PRs small and focused**
   A 50-line PR doing one thing gets reviewed carefully. A 1,500-line PR doing five things gets skimmed and rubber-stamped — reviewers genuinely cannot hold that much context at once. Split unrelated changes into separate PRs.

2. **Write a description that explains WHY, not just what**
   The diff already shows what changed. The description's job is the part the diff can't show: why this approach, what alternatives were considered, what's intentionally out of scope.

3. **Review your own diff before requesting review**
   Catching your own leftover debug code, commented-out blocks, or typos before a reviewer does saves a review round-trip and signals care.

4. **Don't take feedback personally — it's about the code**
   A comment on a function isn't a comment on you. The healthiest review cultures treat critique of code as completely separate from judgment of the person who wrote it.

As the Reviewer — Giving Feedback That Actually Helps

LESS USEFUL

"This is wrong."

MORE USEFUL

"This will throw if `user` is null — looks like that's possible after the change in `getCurrentUser()`. Maybe an early return or optional chaining here?"

LESS USEFUL

"Why did you do it this way?"

MORE USEFUL

"Curious about the reasoning here — would a Set be faster than this array.includes() check for the larger lists we expect? Not blocking, just want to understand the tradeoff."

Distinguishing severity — not every comment is equally important

🚫 Blocking

A genuine bug, security issue, or something that breaks functionality. Must be addressed before merge — say so explicitly.

💭 Suggestion

A better approach exists, but the current one isn't wrong. Worth raising, not worth blocking on — many teams prefix these with "nit:" or "suggestion:" to signal this.

❓ Question

Genuinely seeking to understand the reasoning, not implying disagreement. Often surfaces a real issue, but framed as curiosity rather than correction.

LABELLING SEVERITY EXPLICITLY SAVES A LOT OF BACK-AND-FORTH

A reviewer who prefixes every comment with "nit:", "blocking:", or "question:" makes it immediately obvious to the author what genuinely needs addressing before merge versus what's optional polish — without that, authors often can't tell if a comment is a hard requirement or a passing thought, and either over-react or under-react to it.

What to actually look for as a reviewer

- **Correctness first.** Does this do what it claims to do? Are there edge cases (empty input, null, the boundary conditions) that aren't handled?
- **Security and safety.** Any of the OWASP-style concerns — unsanitised input, exposed secrets, injection risks — covered more deeply in the security-focused parts of this curriculum elsewhere.

- **Consistency with the existing codebase.** Does this match established patterns, or introduce a new one without reason? Inconsistency compounds into real maintenance cost over time.
- **Tests.** Does new behaviour have coverage? Does an existing test actually verify the fix, or just exercise the code path without asserting anything meaningful?
- **What you're NOT reviewing.** Pure style preferences a linter could enforce automatically aren't worth a human reviewer's comment — that's what automated formatting/linting in CI (Chapter 8) is for.

Approving, Requesting Changes, or Commenting

GitHub's review submission offers three distinct outcomes, each sending a different signal:

- **Comment** — feedback with no formal verdict either way. Useful for early-stage or partial reviews.
- **Approve** — "I'm satisfied this is ready to merge." Doesn't mean zero feedback was given, just that none of it is blocking.
- **Request changes** — explicitly blocks merging (if branch protection requires review approval — Course 3, Chapter 6) until addressed and re-reviewed.

> **"REQUEST CHANGES" FOR A MINOR NIT IS USUALLY OVERKILL**
>
> Reserving "Request changes" for things that are genuinely blocking (bugs, security issues, real correctness problems) — and using plain comments or an approval-with-suggestions for everything else — keeps the review process from feeling adversarial over small stuff. Overusing the hard "block" signal trains authors to dread review rather than welcome it.

Review Culture — The Part That's Not About Syntax

- **Assume good intent.** The author wrote what seemed reasonable to them at the time — feedback framed as curious rather than corrective tends to land better and surfaces the same information.
- **Praise good decisions, not just flag problems.** A review that's 100% criticism, even when every point is valid, reads as harsher than intended. A genuine "nice use of early returns here" costs nothing and balances the tone.

- **Respond to review requests reasonably promptly.** A PR sitting unreviewed for days blocks the author's progress and discourages future small, frequent PRs — exactly what Chapter 1's "keep PRs small" advice depends on working well.

CHAPTER 7 QUICK REFERENCE

- **As author:** keep PRs small, explain WHY in the description, self-review before requesting, don't take feedback personally
- **As reviewer:** be specific and explain reasoning, not just "this is wrong"
- **Label severity:** blocking vs suggestion ("nit:") vs genuine question — removes ambiguity about what must change
- **Focus review on:** correctness, security, consistency, test coverage — not style a linter could catch
- **Approve / Comment / Request changes** — reserve "Request changes" for genuinely blocking issues
- **Culture matters:** assume good intent, balance criticism with genuine praise, review promptly
- **Next chapter:** GitHub Actions basics — your first automated CI workflow

CHAPTER 8: GITHUB ACTIONS BASICS

COURSE 2 · CH 8

GitHub Actions Basics

Writing your first automated workflow — running tests on every push, without lifting a finger

Every chapter so far has been manual — you decide when to test, when to check things work. GitHub Actions automates that: a workflow file describes steps to run automatically whenever something happens in your repo (a push, a PR, a schedule), and GitHub runs them on its own servers, for free on a generous tier for public repos and personal use.

The Core Concepts

📄 Workflow

A YAML file in `.github/workflows/` describing what to run and when. A repo can have several, each independent.

⚡ Trigger (on:)

What causes the workflow to run — a push, a PR being opened, a schedule, or several other event types.

🏃 Job

A set of steps that run together on one fresh virtual machine. A workflow can have multiple jobs, which run in parallel by default.

📋 Step

One action within a job — running a command, or using a pre-built "action" someone else published (like checking out your code).

Your First Workflow — Running Tests on Every Push

Create `.github/workflows/test.yml` in your repository:

```yaml
name: Run Tests

on:
  push:
    branches: [main]
  pull_request:
```

```
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - run: npm install
      - run: npm test
```

Reading it line by line:

- **on:** — runs on every push to main, and on every PR targeting main (catching problems before merge, not just after).
- **runs-on: ubuntu-latest** — GitHub spins up a fresh Ubuntu virtual machine for this job, every single run, with nothing left over from previous runs.
- **actions/checkout@v4** — a pre-built action (published by GitHub itself) that clones your repo onto that fresh machine. Almost every workflow starts with this step.
- **actions/setup-node@v4** — another pre-built action, installing the specified Node.js version. Equivalent actions exist for Python, Java, Go, and most other languages.
- **run: npm install / npm test** — ordinary shell commands, exactly as you'd type them locally.

> **THE GITHUB ACTIONS MARKETPLACE HAS A PRE-BUILT ACTION FOR ALMOST EVERYTHING**
>
> Rather than writing raw shell commands for common tasks, search the Marketplace first — deploying to most major hosting providers, sending Slack notifications, running linters, and uploading test coverage reports all have well-maintained existing actions rather than needing custom scripting.

Seeing It Run

Once committed and pushed, the workflow appears under the repository's **Actions** tab — every run shows live logs, step by step, and a pass/fail result.

🟡 **Run Tests** — in progress · triggered by push to main

✅ **Run Tests** — passed in 42s · all checks succeeded

❌ **Run Tests** — failed in 18s · npm test exited with code 1

The same result also shows directly on a PR's page as a status check — and if branch protection requires it to pass (Course 3, Chapter 6), a red ❌ here physically blocks the merge button until fixed.

A Second, Equally Common Workflow — Linting

Running a linter automatically catches the kind of style/consistency issues Chapter 7 flagged as *not* worth a human reviewer's time — letting Actions enforce that instead.

```yaml
name: Lint

on: [push, pull_request]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
      - run: npm install
      - run: npm run lint
```

> **A FAILING WORKFLOW DOESN'T UNDO ANYTHING — IT JUST REPORTS**
>
> GitHub Actions never blocks a push or silently reverts anything on its own; it only runs your defined steps and reports pass/fail. The actual enforcement (blocking a merge on failure) is a separate setting in branch protection rules — without that configured, a failing workflow is informational only.

Command Reference — Concepts, Not Shell Commands

YAML KEY	WHAT IT DOES
`on:`	Defines what event(s) trigger the workflow
`jobs:`	One or more independent sets of steps, run in parallel by default
`runs-on:`	Which virtual machine OS/image the job runs on
`steps:`	The ordered list of actions/commands within a job

YAML KEY	WHAT IT DOES
`uses:`	Runs a pre-built action from the Marketplace or another repo
`run:`	Executes a raw shell command directly

CHAPTER 8 QUICK REFERENCE

- **Workflow files** live in .github/workflows/, written in YAML
- **on:** defines the trigger (push, pull_request, schedule, and others)
- **jobs → steps** — each job runs on a fresh VM; steps run in order within it
- **uses:** runs a pre-built Marketplace action; **run:** runs a raw shell command
- **actions/checkout@v4** is the near-universal first step — clones your repo onto the runner
- **Results appear** on the Actions tab and directly on PRs as status checks
- **A failing workflow only reports** — branch protection (Course 3, Ch6) is what actually enforces it blocking a merge
- **Next chapter (Scenario):** multiple users on the same branch — diverged history, force-push dangers, and branch protection rules

CHAPTER 9: SCENARIO - MULTIPLE USERS, TEAM SCALE

COURSE 2 · CH 9 · SCENARIO

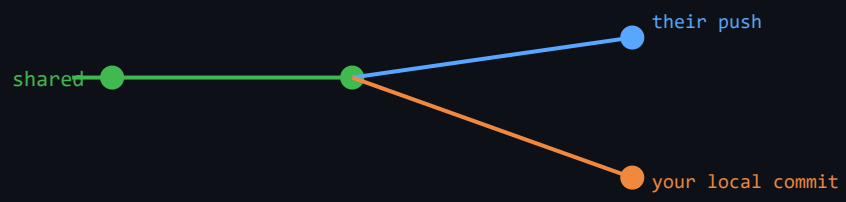
Multiple Users on the Same Branch — Team Scale

Diverged history, why force-push is dangerous, and using branch protection rules to prevent disasters before they happen

Course 1, Chapter 10 covered two people on the same branch as a calm, easy case. At team scale — several people, more frequent pushes, more pressure to move fast — the same situations get higher-stakes, and one extra tool (the force push) enters the picture as something genuinely dangerous if misused. This chapter covers what changes at that scale, and how branch protection rules prevent the worst outcomes structurally rather than relying on everyone remembering the rules.

Diverged History — When Pull Alone Isn't a Clean Fast-Forward

With more people pushing more often, it's common for your local branch and the remote to have genuinely different commits on both sides — not just "they're ahead," but both have commits the other doesn't.



Both sides built on the same shared point, but neither is a superset of the other — pull needs to combine them, not just fast-forward

```
$ git pull
Auto-merging shared.js
Merge made by the 'ort' strategy.
```

This is still entirely normal and resolved the same way as Course 1, Chapter 10 — git creates a merge commit (or you hit a real conflict, resolved per Chapter 3). What changes at team scale is just *how often* this happens, and that's exactly why the next section matters more.

Why Force-Push Is Genuinely Dangerous

`git push --force` overrides git's normal protection (the "fetch first" rejection from Course 1, Chapter 10) and pushes your local branch's history regardless of what's on the remote — **even if that means erasing commits the remote has that you don't.**

⚠️ What it actually does

Overwrites the remote branch to exactly match your local one. Any commit on the remote that isn't also in your local history simply disappears from that branch — for everyone.

🔥 When it goes wrong

You rebased locally (Chapter 2), didn't realize a teammate had pushed new commits in the meantime, and force-pushed — silently deleting their work from the shared branch.

✅ When it's legitimately needed

After rebasing or amending a commit you're CERTAIN no one else has pulled yet — typically your own personal feature branch, never a shared branch like main.

IF YOU MUST FORCE-PUSH, USE --FORCE-WITH-LEASE INSTEAD OF --FORCE

`git push --force-with-lease` adds a safety check: it refuses to force-push if the remote has changed since you last fetched — meaning it won't silently overwrite a teammate's commit you simply haven't seen yet. It's not foolproof, but it converts the most common accidental-disaster scenario into an error message instead of silent data loss. Treat plain `--force` as something to avoid by default.

If someone's work was already overwritten

If a force-push already happened and commits appear to be missing, the situation is usually recoverable — the person who made the lost commits likely still has them locally (their own copy was never touched), and can simply push again. Git's `reflog` (full coverage in Course 3, Chapter 3) can also often recover commits even after a damaging force-push, as long as no one has run aggressive garbage collection in the meantime.

Branch Protection Rules — Preventing This Structurally

Rather than relying on everyone remembering "don't force-push main," GitHub lets you enforce rules at the repository level — Settings → Branches → Add branch protection rule.

🔒 Require a pull request before merging

Disables direct pushes to the protected branch entirely — every change must go through a reviewed PR (Course 1, Ch7), no exceptions, even for admins if configured that way.

✅ Require status checks to pass

Ties directly into Chapter 8's CI workflows — a failing test run physically blocks the merge button until fixed.

👀 Require approving reviews

Specify a minimum number of approvals (commonly one or two) before a PR becomes mergeable — formalises the review culture from Chapter 7.

🚫 Block force pushes

Disables force-pushing to the protected branch entirely, regardless of who's pushing — removes the danger covered above as a possibility altogether.

📜 Require linear history

Forces every merge into the branch to be a rebase or squash, never a merge commit — keeps history reading top-to-bottom with no branching visual noise, if that's the team's preference.

SET THIS UP ON DAY ONE, NOT AFTER THE FIRST INCIDENT

Branch protection costs nothing to configure and prevents an entire category of "someone accidentally force-pushed main" incidents from ever being possible — much cheaper than the alternative of recovering from one after the fact. On any repository more than one person touches, enabling at minimum "block force pushes" and "require a PR before merging" on the main branch is a sensible default from the very start.

Communication — The Non-Technical Half of This Problem

- **Announce before doing anything history-rewriting on a shared branch.** A quick "about to rebase feature/checkout, please pull when I'm done" prevents most surprises entirely.
- **Pull before starting work, not just before pushing** (repeated from Course 1, Chapter 10, worth repeating at team scale) — smaller, more frequent catch-ups produce smaller surprises.
- **If something looks wrong on a shared branch, ask before acting.** A quick message in a team channel beats a guessed fix that makes a confusing situation worse.

CHAPTER 9 QUICK REFERENCE

- **Diverged history** at team scale resolves the same way as Course 1's two-person scenario — just happens more often

- **git push --force** can silently delete a teammate's commits from a shared branch — genuinely dangerous
- **git push --force-with-lease** — safer alternative, refuses if the remote has unseen changes
- **If commits go missing:** the original author likely still has them locally; git reflog can often recover them too
- **Branch protection rules:** require PR before merging, require status checks, require reviews, block force pushes, require linear history
- **Set up protection early** — much cheaper than recovering from the incident it would have prevented
- **Communicate before any history-rewriting operation** on a branch others use
- **Next chapter (final, Scenario):** a repo gone messy after a reorg — broken gitlinks and submodules without .gitmodules

CHAPTER 10: SCENARIO - A REPO GONE MESSY

COURSE 2 · CH 10 · FINAL CHAPTER · SCENARIO

A Repo Gone Messy After a Reorg

Diagnosing and fixing nested repos and broken gitlinks/submodules — the full version of the incident introduced back in Course 1

Course 1, Chapter 9 introduced the broken-gitlink incident from the website→debserver reorg as a teaser — enough to recognise the symptom. This final chapter of Course 2 goes the rest of the way: what a submodule actually is when set up correctly, why a folder can end up tracked as one *by accident*, and the full diagnose-and-fix process.

PICKING THE INCIDENT BACK UP

During the reorg, a subfolder that used to be its own separate git repository lost its `.git` directory in the move. The parent repository, however, still had a recorded entry treating that subfolder as a **gitlink** — git's internal mechanism for "this folder is itself a repository, tracked by commit reference rather than by its actual file contents." With the inner `.git` gone, that reference pointed at a commit that no longer existed anywhere reachable.

The result: `git status` on the parent repo showed a single confusing `M website` line instead of the real, individual file changes inside that folder — because git was treating the entire folder as one opaque unit, exactly as it would a properly configured submodule.

What a Submodule Actually Is, Set Up Correctly

A real git submodule is a deliberate, supported feature: embedding one git repository inside another, where the outer repo tracks *which exact commit* of the inner repo to use — not the inner repo's file contents directly. This is genuinely useful for shared libraries or vendored dependencies that need to stay pinned to a specific version.

```
$ # The correct way to add a submodule
$ git submodule add https://github.com/someone/shared-lib.git libs/shared-lib
Cloning into 'libs/shared-lib'...
```

This creates a proper `.gitmodules` file at the repo root, recording the submodule's URL and path — and the inner folder gets a gitlink entry (mode `160000`, same mode flagged in Course 1, Chapter 9) that's *intentional and documented*.

THE DEFINING SYMPTOM OF A REAL PROBLEM: A GITLINK WITH NO .GITMODULES ENTRY

A properly configured submodule always has a matching entry in `.gitmodules`. A folder tracked with mode `160000` but with *no* corresponding `.gitmodules` entry is the unambiguous signature of an accident — exactly what the reorg produced. This is the single fastest way to distinguish "someone meant to do this" from "something broke."

How a Folder Becomes an Accidental Gitlink

📦 A nested repo existed, then lost its .git

Exactly the reorg scenario — a subfolder used to have its own .git, something (a partial move, a cloud-sync quirk per Course 1 Ch9) removed it, but the parent repo's earlier commit of the gitlink reference remains.

Diagnostic: check if .git ever existed there — old backups, git history of the parent repo itself, or memory of past project structure.

🆕 A nested git init happened by accident

Someone ran git init inside a subfolder of an already-tracked repo (Course 1, Chapter 2's warning about this exact mistake) — creating a brand-new, unintended nested repository.

Diagnostic: does the subfolder's .git/ look freshly created, with little or no history of its own?

📥 A cloned third-party project was dropped in wholesale

Someone copied an entire other project — including its own .git folder — directly into a subfolder of the current repo, rather than removing .git or using a real submodule.

Diagnostic: does the inner .git's remote (git remote -v from inside it, if it still exists) point at a recognisable external project?

The Full Diagnose-and-Fix Process

1. **Confirm the gitlink with git ls-files -s**
   Mode 160000 on the suspicious path confirms git is treating it as a separate repository (Course 1, Ch9's diagnostic command, repeated here as the entry point).

2. **Check for a matching .gitmodules entry**
   `cat .gitmodules` (or note its absence entirely) — confirms whether this was ever intentional.

3. **Check whether the folder's own .git still exists on disk**
   If it does, the fix may just be re-adding it properly as a real submodule, or removing the gitlink reference and re-adding the folder normally if it was never meant to be separate.

4. **If accidental and .git is gone: decide whether the content needs tracking normally**
   Usually yes — the folder's actual files should just be ordinary tracked content in the parent repo, not a gitlink to nothing.

5. **Remove the broken gitlink entry, then re-add the folder as ordinary files**
   See commands below — this is the actual fix for the reorg incident's specific situation.

```
$ # Remove the broken gitlink reference from the index (doesn't touch files on disk)
$ git rm --cached website
rm 'website'

$ # Re-add the folder's actual contents as ordinary tracked files
$ git add website/
$ git status
Changes to be committed:
new file: website/index.html
new file: website/styles/main.css
... (every individual file, now tracked normally)

$ git commit -m "Stop tracking website/ as a broken gitlink; track contents normally"
```

After this, `git status` shows real, individual file changes inside that folder again — exactly the fix this session's actual incident needed, demonstrated here as a repeatable, named procedure rather than a one-off troubleshooting session.

PREVENTION IS SIMPLER THAN THE FIX

Avoiding nested git repositories entirely — one `.git` per project, full stop, unless deliberately using a properly documented submodule — sidesteps this entire category of problem. When reorganising folders (Course 1, Chapter 9), explicitly checking for stray `.git` directories with `find . -name .git -type d` before and after a move catches this before it becomes a confusing future incident.

CHAPTER 10 QUICK REFERENCE — AND COURSE 2 WRAP-UP

- **A real submodule** always has a matching .gitmodules entry — its absence is the signature of an accident
- **git ls-files -s**, mode 160000 — confirms a path is tracked as a gitlink, intentional or not
- **Common causes:** a nested repo lost its .git, an accidental git init inside a tracked folder, a wholesale-copied third-party project with its own .git left in place
- **The fix:** git rm --cached <path> (removes the broken gitlink reference) → git add <path>/ (re-tracks contents normally) → commit
- **Prevention:** one .git per project; check for stray nested .git folders before/after any reorg
- **Course 2 recap:** branching strategies (Ch1) → merge vs rebase (Ch2) → conflicts (Ch3) → forks (Ch4) → stash/cherry-pick/tags (Ch5) → issues/projects (Ch6) → code review (Ch7) → Actions basics (Ch8) → team-scale collaboration (Ch9) → broken gitlinks (Ch10)
- **Next:** Course 3 — Advanced Git & GitHub Mastery (internals, interactive rebase, reflog, proper submodules, advanced Actions, security, large repos, bisect, and full disaster-recovery scenarios)