

Git & GitHub: Commit, Push, Pull

The opener: version control in plain words, your first init/add/commit in the terminal, Git vs GitHub, push and pull, and the one rule — never commit secrets.

Introduction

Every developer has lived this story: a folder full of files named `project-final`, `project-final-2`, `project-ASLI-final` — and no idea which one works, or what changed between them. **Git** ends that story. Git is a **version control system**: it remembers every version of your project, who changed what, and lets you travel back to any earlier point. **GitHub** is Git's online home — the place your Git history lives so a team (and future you, on another computer) can reach it. This opener shows the whole journey once, quickly: `init`, `add`, `commit`, `push`, `pull` — the details fill the rest of the series. (Roman Urdu: *Git aap ke code ki poori history yaad rakhta hai — aur GitHub us history ko online rakhta hai, taake team aur aap khud kahin se bhi kaam kar saken.*)

Key points

- **A commit is a saved checkpoint:** one commit = one meaningful change, saved with a message and a permanent ID. Your project becomes a chain of checkpoints you can inspect or revisit — never a pile of "final" copies.
- **Git lives on your computer; GitHub lives online:** Git works fully offline — commit on a train, no problem. GitHub is a website that hosts Git repositories so people can share, back up and review them. Git ≠ GitHub, the way email ≠ Gmail.
(Roman Urdu: *Git aap ke computer par chalta hai, GitHub internet par — dono alag cheezein hain, bas saath kaam karti hain.*)
- **The daily loop is four verbs:** `git add` (choose what goes into the next checkpoint), `git commit` (save the checkpoint), `git push` (send your commits to GitHub), `git pull` (bring other people's commits down to you).
- **The staging area sits between your files and history:** files you edit are "modified"; `git add` moves them to the **staging area**; `git commit` saves only what is staged. This is how one messy afternoon becomes two clean, separate commits.
- **Never commit secrets:** passwords, API keys and `.env` files must never enter a commit. Once a secret is pushed, treat it as public forever — changing the password is the only real fix (lesson G.6 builds the wall: `.gitignore`).

Code examples

Example 1 — Your first repo, in four commands

```
$ mkdir my-first-repo && cd my-first-repo
$ git init
Initialized empty Git repository in C:/projects/my-first-repo/.git/
$ echo "<h1>Hello Git</h1>" > index.html
$ git add index.html
$ git commit -m "Pehla commit: hello page"
[main (root-commit) a1b2c3d] Pehla commit: hello page
1 file changed, 1 insertion(+)
create mode 100644 index.html
```

What it does: `git init` creates the hidden `.git` folder that turns an ordinary folder into a repository. The `echo` line creates a tiny file, `git add` stages it, and `git commit` saves the first checkpoint with your message.

What you will see: the commit line shows the branch (`main`), a short ID (`a1b2c3d` — yours will differ), your message, and a summary: 1 file changed. Your project now has a history of exactly one checkpoint — and it can never be "lost" the way loose files are.

Example 2 — Push: sending your history to GitHub

```
$ git remote add origin https://github.com/your-username/my-first-repo.git
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 298 bytes | 298.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/your-username/my-first-repo.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

What it does: first you create an empty repo on github.com, then `git remote add origin` teaches your local repo its address (`origin` is the standard nickname). `git push -u origin main` uploads the `main` branch; `-u` links the two branches so later pushes are just `git push`.

What you will see: the objects count up to 100%, and the `[new branch]` line confirms the branch arrived. Refresh the repo page on GitHub — your `index.html` and your commit message are sitting there, visible from any computer on earth.

Example 3 — Pull: bringing a change back down

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
Unpacking objects: 100% (5/5), 1.02 KiB | 130.00 KiB/s, done.
From https://github.com/your-username/my-first-repo
   a1b2c3d..e4f5g6h  main       -> origin/main
Updating a1b2c3d..e4f5g6h
Fast-forward
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

What it does: imagine you edited the file on the GitHub website (or a teammate pushed a fix). `git pull` downloads the new commits and updates your local files to match. "Fast-forward" means your copy simply moved forward along the same chain — no merging drama.

What you will see: the changed file is listed with its diff summary (1 insertion, 1 deletion), and opening `index.html` shows the new text already in place. Run `git pull` again right away and Git answers `Already up to date.` — the friendliest sentence in Git.

Quick recap

- **Git** = version control on your machine: a chain of commits you can inspect and revisit. **GitHub** = the same history hosted online for sharing, backup and teamwork.
- The loop: edit → `git add` → `git commit` → `git push`; and `git pull` to receive. The staging area is what makes commits clean and deliberate.
- One law above all: **secrets never enter a commit** — once pushed, a password is public.

Common mistakes

1. **Saving copies instead of committing** — `project-final-2.zip` thinking. The fix: one folder, one repo, and a commit for every meaningful step; Git already keeps every version, with messages saying what changed.
2. **Committing everything at once, rarely** — one giant commit called "stuff" after a week of work. The fix: small, frequent commits, each one idea ("Add login form", "Fix header on mobile") — future you reads the history like a story.
3. **Committing an `.env` or a key "just this once"** — bots scan GitHub for keys within minutes. The fix: `.gitignore` from day one (G.6), and if a secret ever lands in a commit, change the secret itself — deleting the file does not unpublish it.

Practice

1. Create a folder `git-practice`, run `git init`, add a file `about.txt` with one line about yourself, and make your first commit with the message "About me file". Run `git log --oneline` and find your commit's ID.
2. Change the line in `about.txt`, then run `git status` BEFORE adding anything. Read the output carefully — which section is your file in? Then `git add` it, run `git status` again, and spot exactly what moved. Commit with a message that says what changed.

Watch the Code Scope channel: <https://www.youtube.com/@CodeScopePK> — this lesson's own video joins the channel when it is recorded.

Read online: <https://toolscope.netlify.app/notes/git-github-intro> · More free tools and notes at <https://toolscope.netlify.app>