Git Fundamentals

What Git is, how it differs from GitHub, and the working tree/staging area/repository mental model -- via git init, status, add, commit, diff, and log. The 1st lesson in both the Git & GitHub course and its Git Fundamentals category.

Beginner 30 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/git-fundamentals

---

Every developer eventually asks the same question: "how do I get my changes back if I mess this up?" Git is the tool the entire industry has settled on to answer that question — not by trusting you to be careful, but by tracking every version of every file so you never actually lose work. This lesson builds the mental model everything else in this course depends on: what Git actually is, how it's different from GitHub, and the three-stage flow — working tree, staging area, repository — that every single Git command operates on.

What Is Git?

Git is a **distributed version control system**: a tool that records snapshots of a project's files over time, so you can see what changed, who changed it, and go back to any earlier snapshot. "Distributed" is the key word — every developer who clones a repository gets the **entire history**, not just the latest files. There is no central server you depend on to see past versions; your own machine already has everything.

This matters in practice: you can commit, browse history, create branches, and undo mistakes completely offline. The only time you need a network connection is when you want to *synchronize* with someone else's copy (which is exactly what GitHub is for — more on that next).

Git vs GitHub

This is the single most common point of confusion for newcomers, so let's be precise:

- **Git** is the version control tool itself — a program that runs on your machine and manages a project's history.
- **GitHub** is a website (and a company) that *hosts* Git repositories on the internet, and adds collaboration features on top: Pull Requests, code review, issue tracking, CI integration.

You can use Git without ever touching GitHub — plenty of solo projects live entirely on a developer's laptop. GitHub becomes useful the moment you need a shared, always-available copy of the repository that other people (or your own other machines) can push to and pull from. Other companies offer the same kind of hosting (GitLab, Bitbucket) — they're all built on the same underlying Git.

Installing and Configuring Git

Git is installed the same way as any other developer tool for your platform (a package manager, or an installer from git-scm.com). Once installed, tell Git who you are — this identity gets attached to every commit you make:

```
git --version
git config --global user.name "Ada Lovelace"
git config --global user.email "ada@example.com"
```

`--global` means this applies to every repository on your machine, not just the current one. You can override it per-project by running the same command without `--global` inside that project's folder — useful if you use a different email for work vs. personal projects.

> Run `git config --list` at any time to see every setting currently in effect, including where each one came from.

Creating a Repository — git init

A **repository** (or "repo") is a project folder that Git is tracking. You turn any folder into a repository with one command:

```
#!/bin/sh
# Turning a plain folder into a Git repository, then confirming it worked.

mkdir task-tracker
cd task-tracker
git init

# Output:
# Initialized empty Git repository in /home/ada/task-tracker/.git/

ls -la

# Output (trimmed):
# drwxr-xr-x  3 ada  staff   96 Aug 24 10:00 .
# drwxr-xr-x  4 ada  staff  128 Aug 24 10:00 ..
# drwxr-xr-x  9 ada  staff  288 Aug 24 10:00 .git

# The .git folder is the repository itself -- it did not exist a moment ago,
# and the "task-tracker" folder had no history until git init created it.
```

`git init` creates a hidden `.git` folder inside your project. That folder *is* the repository — it's where every snapshot, every commit, and the entire history will live. Deleting `.git` deletes the project's Git history (but not your actual files) — it's worth remembering, because it's also how you'd start a project's Git history completely over if you ever needed to.

The Working Tree, Staging Area and Repository

Every Git command you'll learn operates on one of three areas, and understanding the flow between them is the single most important mental model in this entire course:

```
Working Tree            Staging Area              Repository
(your actual files) →   (what will go into  →     (permanent, committed
                          the NEXT commit)          history)
              git add               git commit
```

- **Working Tree** — the actual files on disk, exactly as you see them in your editor. This is where you make changes.
- **Staging Area** (also called the "index") — a holding area for changes you've decided *should* be part of the next commit. You build up the staging area with `git add`.
- **Repository** — the permanent, committed history. Once something is committed, it's a durable snapshot you can always come back to.

The staging area is what makes Git different from simpler "track my changes" tools: you can modify five files but choose to commit only two of them, by staging just those two. This lets you build focused, meaningful commits instead of one giant commit that mixes unrelated changes.

Checking Changes — git status

At any point, `git status` tells you exactly where every file stands relative to these three areas:

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        UserService.java

nothing added to commit but untracked files present (use "git add" to track)
```

Read `git status` output carefully — it tells you, in plain language, exactly which command to run next. This habit (running `git status` before and after almost everything) will save you more confusion than any other single practice in this course.

Staging Changes — git add

`git add` moves a change from the working tree into the staging area:

```
git add UserService.java      # stage one specific file
git add src/                  # stage everything under a folder
git add .                     # stage everything in the current directory
```

Staging is not the same as saving — your file is already saved on disk the moment your editor wrote it. Staging only tells Git "this is one of the changes I want in my *next* commit."

> `git add .` stages *everything* in the current directory, including files you may not have meant to commit (temporary files, local config, build output). Get in the habit of running `git status` right before `git add .` to see exactly what you're about to stage.

Creating Commits — git commit

A **commit** takes everything currently in the staging area and permanently records it as a new snapshot in the repository:

```
#!/bin/sh
# The full Working Tree -> Staging Area -> Repository flow for a small Java
# class, from writing the file to committing it.

# 1. A new file exists on disk, but Git isn't tracking it yet.
echo 'public class UserService {
    public User findById(Long id) {
        return null; // not implemented yet
    }
}' > UserService.java

git status
# Output:
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#         UserService.java

# 2. Move it into the staging area.
git add UserService.java

git status
# Output:
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#         new file:   UserService.java

# 3. Commit what's staged -- this creates a permanent snapshot.
git commit -m "Add UserService with findById stub"

# Output:
# [main (root-commit) 4f2a1c9] Add UserService with findById stub
#  1 file changed, 5 insertions(+)
#  create mode 100644 UserService.java

git status
# Output:
# nothing to commit, working tree clean
```

Every commit requires a message describing what changed and why (we'll cover what makes a *good* commit message in the next lesson). Without `-m`, Git opens your default text editor so you can write a longer message — useful for commits that need more explanation than one line.

Viewing Changes — git diff

`git diff` shows you exactly what changed, line by line, before you commit it:

```
git diff              # unstaged changes: working tree vs. staging area
git diff --staged     # staged changes: staging area vs. last commit
```

Lines starting with `-` are what's being removed, lines starting with `+` are what's being added. Running `git diff --staged` right before you commit is a simple, extremely effective

habit — it's your last chance to catch an accidental change (like a leftover debug line) before it becomes part of permanent history.

Viewing History — git log

`git log` shows the commit history — every snapshot ever recorded, newest first:

```
$ git log
commit 4f2a1c9e8b3d5a6f7e8d9c0b1a2f3e4d5c6b7a8f
Author: Ada Lovelace <ada@example.com>
Date:   Mon Aug 24 10:15:00 2026 +0300

    Add UserService with basic CRUD methods
```

Each commit gets a unique identifier (the long hexadecimal string, called a **hash** or **SHA**) — this is how Git refers to that exact snapshot everywhere, and how you'll reference specific commits in later lessons (undoing changes, cherry-picking, and more all work by hash).

Common Mistakes

- **Running `git commit` without staging anything first.** Git only commits what's in the staging area — a saved file on disk is not automatically included.
- **Forgetting `.git` exists.** Some beginners accidentally commit inside the wrong folder, or delete `.git` thinking it's just a cache. It's the entire project history.
- **Not reading `git status` output.** It almost always tells you exactly what to do next — skipping it leads to confusion that a 5-second read would have prevented.

Best Practices

- Configure `user.name` / `user.email` once, globally, right after installing Git — every commit depends on it.
- Run `git status` liberally. It's free, it's fast, and it never changes anything — there's no reason not to check it constantly.
- Review `git diff --staged` before every commit, as a final check.

Summary, Cheat Sheet, and Glossary

Summary

- Git is a distributed version control tool; GitHub is a website that hosts Git repositories and adds collaboration features on top.
- Every project's history lives in a `.git` folder, created by `git init`.
- Changes flow through three areas: **Working Tree** → (`git add`) → **Staging Area** → (`git commit`) → **Repository**.

- `git status` tells you where things stand; `git diff` shows exact line changes; `git log` shows commit history.

Cheat Sheet

```
git init                      # create a new repository
git config --global user.name "..."
git config --global user.email "..."
git status                    # see current state
git add <file>                # stage a change
git diff                      # see unstaged changes
git diff --staged             # see staged changes
git commit -m "message"       # commit staged changes
git log                       # view commit history
```

Glossary

- **Repository (repo)** — a project folder tracked by Git (identified by its `.git` folder).
- **Working Tree** — the actual files on disk.
- **Staging Area (index)** — changes selected to be part of the next commit.
- **Commit** — a permanent, named snapshot of the staged changes.
- **Hash (SHA)** — the unique identifier of a specific commit.