



Git & GitHub

Complete Practical Handbook

Beginner → Intermediate → Advanced

Inside this handbook

Git commands • GitHub repositories • Branching • Merge conflicts • Pull requests • SSH • Team collaboration • Professional workflow • Best practices • Troubleshooting • Interview preparation

NextTechEra - Learning Resources
Your Vision, Our Innovation
nexttechera.in

Table of Contents

1. Version Control: Why It Exists	3
2. Git and GitHub: Know the Difference	3
3. Install and Configure Git	3
4. Create Your First Repository	4
5. Working Tree, Staging Area, and Commits	4
6. Commit Messages and History	4
7. Create a GitHub Repository and Connect It	5
8. Clone, Pull, Push, and Fetch	5
9. Branches: Work Without Breaking Main	6
10. Merge Branches	6
11. Merge Conflicts: Resolve Them Deliberately	6
12. Stash: Pause Work Without a Temporary Commit	7
13. Undo Changes Safely	7
14. .gitignore and Secret Hygiene	7
15. HTTPS vs SSH for GitHub	8
16. Pull Requests: The Collaboration Center	8
17. Code Review: Improve Code Without Making It Personal	9
18. Forks and Open-Source Contribution	9
19. GitHub Issues, README, and Project Documentation	9
20. Professional Team Workflow	10
21. Git Best Practices Checklist	10
22. Common Git Mistakes and How to Avoid Them	11
23. Common Errors and Quick Fixes	11
24. More Troubleshooting Skills	11
25. Git Command Cheat Sheet	12
26. Real-World Practice Project	12
27. Practice Scenarios	13
28. Git & GitHub Interview Questions	13
29. Advanced Interview Notes	14
30. Final Revision: The Git Workflow	14

How to use this handbook

Read the concepts once, type every command yourself, and then complete the real-world practice workflow near the end. Git becomes easy when you use it every day.

1. Version Control: Why It Exists

Version control records changes to files so you can understand what changed, who changed it, and safely return to an earlier version when required. For software teams, it replaces the risky habit of folders such as *project-final*, *project-final-2*, and *project-final-really-final*.

- Keeps a reliable history of source-code changes.
- Lets multiple developers work on the same project without overwriting each other.
- Makes experimentation safer because work can happen on separate branches.
- Helps teams review changes before they reach production.
- Makes it easier to find the exact commit that introduced a bug.

Local, centralized, and distributed version control

Model	Idea	Typical strength
Local	History lives only on one machine.	Simple, but poor for collaboration and disaster recovery.
Centralized	Everyone connects to one central server.	Single source of truth, but the server is a major dependency.
Distributed	Every clone contains repository history.	Fast local work, strong collaboration, resilient history.

Remember

Git is a **distributed** version control system. A normal clone contains the project files plus the repository history needed for everyday work.

2. Git and GitHub: Know the Difference

Git is the version-control tool that runs on your computer. It tracks snapshots, branches, merges, and history. **GitHub** is an online collaboration platform that hosts Git repositories and adds team features such as Pull Requests, Issues, Actions, releases, permissions, and code review.

Git	GitHub
Runs locally on your machine.	Runs as an online service/platform.
Creates commits and branches.	Hosts repositories and collaboration workflows.
Works without internet for local operations.	Internet is needed for remote collaboration.
CLI commands: commit, branch, merge, log.	PRs, Issues, reviewers, Actions, repository settings.

Simple rule

Git = version control engine. GitHub = place where teams host and collaborate on Git repositories.

The everyday Git mental model

Working Directory → edit files → **Staging Area** → choose what will be committed → **Local Repository** → store a commit → **Remote Repository** → share by pushing to GitHub.

3. Install and Configure Git

Install Git from the official package for your operating system. On Windows, Git for Windows also provides Git Bash. On macOS you can use Xcode Command Line Tools or a package manager; on Linux use your distribution package manager.

```
git --version

# Example output
git version 2.51.0
```

Set the identity that Git records inside new commits. This is not your GitHub password; it is author metadata stored in commit history.

```
git config --global user.name "Amit Verma"
git config --global user.email "you@example.com"
git config --list
```

Professional tip

Use the email address associated with your GitHub account if you want GitHub to correctly attribute your commits. For work repositories, follow your organization's identity policy.

Configuration scopes

- **--system**: affects all users on the machine.
- **--global**: affects your user account and is the normal choice for identity.
- **--local**: affects only the current repository and can override global values.

4. Create Your First Repository

A repository is a project directory whose history is managed by Git. **git init** creates a hidden **.git** directory containing Git metadata, object storage, references, configuration, and history.

```
mkdir student-management
cd student-management
git init
git status
```

After initialization, Git can see the files in the folder but does not automatically include them in history. New files begin as **untracked**. You choose what enters the next commit by staging it.

Do not delete .git casually

Removing the **.git** directory removes the repository metadata and local history. Your project files remain, but the folder is no longer a Git repository.

Four file states you will see often

State	Meaning
Untracked	File exists in the folder but Git has not started tracking it.
Modified	A tracked file has changed since the last commit.
Staged	The current version is selected for the next commit.
Committed	The snapshot is stored in local repository history.

5. Working Tree, Staging Area, and Commits

Git separates editing from committing. This is powerful because you can change ten files but stage only the five that belong to one logical change. That keeps commits focused and easier to review.

```
git status
git add app/Services/AttendanceService.php
git add tests/Feature/AttendanceTest.php
git status
git commit -m "Fix duplicate attendance validation"
```

Best practice

A commit should represent one understandable change. Avoid mixing an unrelated bug fix, formatting changes, and a new feature in the same commit.

Useful staging commands

Command	Use
git add file.php	Stage one file.
git add .	Stage changes under the current directory.
git add --all	Stage additions, modifications, and deletions across the repository.
git restore --staged file.php	Remove a file from the staging area without deleting your working changes.

6. Commit Messages and History

A commit message should help another developer understand the intent of the change without opening every file. Prefer a short, specific, action-oriented summary.

Weak message	Better message
update	Update student profile validation
fix	Fix duplicate attendance entry validation
changes	Add pagination to student report
done	Remove deprecated payment callback

```
git commit -m "Add role-based access to fee reports"

git log
git log --oneline
git show <commit-hash>
```

git log shows history including commit hash, author, date, and message. **git show** is useful when you want to inspect the exact patch and metadata for one commit.

Team habit

Before committing, review **git status** and **git diff --staged**. This catches accidental debug code, secrets, generated files, and unrelated edits.

7. Create a GitHub Repository and Connect It

On GitHub, create a repository with a clear name and choose public/private visibility according to the project. If you already have a local project with commits, connecting an empty remote avoids unnecessary first-merge confusion.

```
git remote add origin https://github.com/username/student-management.git
git remote -v
git branch -M main
git push -u origin main
```

origin is the conventional nickname for the primary remote. **-u** sets the upstream relationship, so future **git push** and **git pull** usually work without repeating the remote and branch name.

Remote basics

```
git remote -v
git remote get-url origin
git remote set-url origin <new-url>
```

Common error

If Git says **remote origin already exists**, inspect it with **git remote -v**. Update the URL instead of adding another origin.

8. Clone, Pull, Push, and Fetch

Clone creates a local copy of a remote repository. **Fetch** downloads remote history without integrating it. **Pull** fetches and then integrates updates. **Push** sends your local commits to a remote branch.

```
git clone https://github.com/org/project.git
cd project

git fetch origin
git pull origin main
git push origin main
```

Command	What it does	Working files change?
git fetch	Downloads remote commits and refs.	No
git pull	Fetches, then integrates into current branch.	Usually yes
git push	Uploads local commits to remote.	No local file change

Before you start coding

On a shared branch, update your local view of the remote first. Teams commonly use **git fetch** or **git pull** before starting new work.

9. Branches: Work Without Breaking Main

A branch is a movable reference to a sequence of commits. Branches let you isolate a feature, bug fix, experiment, or release. They are lightweight, so teams create them frequently.

```
git branch
git switch -c feature/student-import
# make changes
git add .
git commit -m "Add student CSV import"
git push -u origin feature/student-import
```

Older Git versions commonly use **git checkout -b feature/name**. Modern Git provides **git switch -c**, which expresses the intent more clearly.

Practical branch naming

Purpose	Examples
Feature	feature/login, feature/student-import
Bug fix	bugfix/payment-rounding, fix/report-filter
Hotfix	hotfix/auth-token-expiry
Release	release/v2.3.0

Keep branches focused

A branch is easier to review when it has one clear purpose and is merged quickly instead of staying alive for weeks.

10. Merge Branches

Merging combines histories. A **fast-forward** merge is possible when the target branch has not moved since the feature branch was created. A **three-way merge** creates a merge commit when histories have diverged.

```
git switch main
git pull origin main
git merge feature/student-import
git push origin main
```

In many professional teams, developers do not merge feature branches directly from the terminal into protected branches. Instead, they push the feature branch and merge through a reviewed Pull Request on GitHub.

Safe habit

Merge only when the feature branch is tested and the target branch is up to date. Review the diff before finalizing the merge.

Delete merged branches

```
git branch -d feature/student-import
git push origin --delete feature/student-import
```

11. Merge Conflicts: Resolve Them Deliberately

A merge conflict happens when Git cannot automatically decide how to combine competing edits. Git marks the conflicting region so a human can decide the correct final content.

```
<<<<<< HEAD
return $user->name;
=====
return $user->full_name;
>>>>>> feature/profile
```

To resolve: open the file, understand both versions, write the correct final code, remove the conflict markers, test the result, stage the resolved file, and continue the merge.

```
git status
# edit conflicted files
git add app/Http/Resources/UserResource.php
git commit
```

Warning

Do not blindly accept "ours" or "theirs" and do not randomly delete conflict markers. A clean merge can still contain a logical bug.

Reduce conflict frequency

- Pull/fetch regularly.
- Keep branches small and short-lived.
- Avoid formatting unrelated files in feature commits.
- Coordinate when several developers edit the same high-conflict file.

12. Stash: Pause Work Without a Temporary Commit

Stash temporarily saves uncommitted working changes and gives you a cleaner working tree. It is useful when you must switch context before your current work is ready to commit.

```
git stash
git stash push -m "WIP attendance report"
git stash list
git stash show -p stash@{0}
git stash pop
```

git stash pop applies the stash and removes it if successful. **git stash apply** applies it but keeps the stash entry. This is useful when you want a safer, repeatable recovery path.

Do not use stash as long-term storage

Important work belongs in meaningful commits on a branch. Stash is best for short-lived context switching.

13. Undo Changes Safely

Git has several undo tools, and they do different things. Choose the command based on whether the change is uncommitted, staged, committed locally, or already shared.

Command	Typical use	History impact
git restore file	Discard unstaged working-tree edits.	No new commit
git restore --staged file	Unstage while keeping working changes.	No new commit
git revert <hash>	Create a new commit that reverses an older commit.	Safe for shared history
git reset --soft <hash>	Move branch pointer but keep changes staged.	Rewrites local history
git reset --hard <hash>	Move branch and discard matching working/staged changes.	Destructive; rewrites local history

Shared repository rule

If a bad commit has already been pushed and other people may depend on it, **git revert** is usually safer than rewriting published history.

```
git revert a1b2c3d
# Git creates a new commit that reverses the selected commit
```

14. .gitignore and Secret Hygiene

.gitignore tells Git which untracked paths should normally be ignored. It prevents dependencies, build output, editor metadata, logs, and local environment files from polluting the repository.

```
# Laravel / PHP
.env
/vendor/
/storage/logs/*.log

# Node / Vue
node_modules/
dist/

# Editors
.idea/
.vscode/
```

Ignoring a path does not remove a file that is already tracked. If a secret was committed, simply adding it to `.gitignore` does not erase it from history. Rotate the exposed credential and follow your organization's incident process.

Never commit

Passwords • database credentials • API keys • private SSH keys • access tokens • production secrets • private certificates.

Check before pushing

```
git status
git diff
git diff --staged
```

15. HTTPS vs SSH for GitHub

GitHub supports remote operations over HTTPS and SSH. HTTPS is easy to understand and usually authenticates with a credential helper or token. SSH uses a public/private key pair and is convenient for frequent developer workflows.

HTTPS	SSH
Remote starts with <code>https://</code>	Remote starts with <code>git@github.com:</code>
Uses browser/token/credential helper depending on setup.	Uses an SSH key accepted by GitHub.
Simple for many beginners.	Very convenient for daily terminal use after setup.

```
ssh-keygen -t ed25519 -C "you@example.com"

# After adding the public key to GitHub:
ssh -T git@github.com
```

Critical security rule

Share your **public** key when needed. Never share the **private** key. Protect it like a password.

```
git remote set-url origin git@github.com:username/project.git
```

16. Pull Requests: The Collaboration Center

A Pull Request (PR) proposes that one branch should be merged into another. It gives the team a place to discuss code, run automated checks, request changes, and preserve review history before merging.

Typical PR workflow

- Create a focused feature branch.
- Commit and push the branch.
- Open a Pull Request with a clear title and description.
- Automated tests and checks run.
- Reviewers inspect the code and request changes if needed.
- Author updates the same branch; the PR updates automatically.
- After approval and passing checks, merge according to team policy.

Good PR title

Add CSV import for student admissions is much more useful than **changes** or **fix stuff**.

A useful PR description answers

- What problem does this solve?
- What changed?
- How was it tested?
- Are there migrations, configuration changes, or deployment notes?
- Are there screenshots for UI changes?

17. Code Review: Improve Code Without Making It Personal

Code review is a technical quality process, not a contest. Review the change against requirements, correctness, security, maintainability, tests, performance, and team conventions.

Reviewer checks	Author responsibilities
Does the logic handle edge cases?	Keep the PR small enough to review.
Is sensitive data protected?	Explain non-obvious decisions.
Are queries/API calls efficient?	Test before requesting review.
Are names and structure clear?	Respond to feedback with context.
Are tests meaningful?	Update docs or migration notes if needed.

Constructive comment

Instead of "This is wrong", try: *Could we move this validation into the Form Request so the controller stays focused on orchestration?*

18. Forks and Open-Source Contribution

A **fork** is a repository copy under your own account or organization. It is common when you do not have direct write access to the original project. A **clone** is a local copy on your machine. They solve different problems.

Typical open-source flow

- Fork the original repository on GitHub.
- Clone your fork locally.
- Add the original repository as an **upstream** remote.
- Create a focused branch.
- Commit and push to your fork.
- Open a Pull Request from your fork branch to the original repository.

```
git clone git@github.com:yourname/project.git
cd project
git remote add upstream git@github.com:original/project.git
git fetch upstream
```

Before contributing

Read CONTRIBUTING.md, coding standards, issue templates, and license information. Every project can have a different workflow.

19. GitHub Issues, README, and Project Documentation

Issues can track bugs, features, tasks, and discussions. Useful issues describe the expected behavior, actual behavior, reproduction steps, environment, and evidence such as logs or screenshots.

Example bug issue

Title: Login button does not respond on mobile

Steps: Open login page on 390px viewport → enter valid credentials → tap Login.

Expected: Request is submitted.

Actual: No request is sent and no validation message appears.

README.md should help someone start quickly

- Project name and one-line purpose.

- Features and technology stack.
- Requirements and installation steps.
- Environment/configuration notes without exposing secrets.
- How to run tests and local development.
- Usage examples, screenshots, API docs, or demo links.
- Contribution and license information where relevant.

```
# Student Management System

## Setup
composer install
cp .env.example .env
php artisan key:generate
php artisan migrate --seed
```

20. Professional Team Workflow

Teams choose workflows based on release process, product size, and deployment frequency. A simple feature-branch model often works well: protected main branch, short-lived feature branches, Pull Requests, required checks, and automated deployment.

```
main
├── feature/login
├── feature/student-import
├── bugfix/attendance-filter
└── hotfix/payment-callback
```

Example developer routine

- Fetch/pull latest target branch.
- Create a new feature branch.
- Implement a small unit of work.
- Run tests and static checks.
- Review diff and commit logically.
- Push branch and open PR.
- Address review feedback.
- Merge after required approvals/checks.
- Delete the branch when finished.

About develop branches

Some teams use **main + develop**; others use only **main + short-lived branches**. Follow the workflow your team has intentionally chosen instead of adding branches by habit.

21. Git Best Practices Checklist

- **Commit often, but commit logically.** Each commit should tell a useful story.
- **Use meaningful messages.** Future debugging depends on readable history.
- **Keep branches short-lived.** Long-running branches accumulate conflicts.
- **Pull/fetch before important work.** Stay aware of upstream changes.
- **Never commit secrets.** Use environment variables and secret stores.
- **Review diffs before commit and push.** Catch accidental files and debug code.
- **Prefer PRs for shared branches.** Review and automation reduce risk.
- **Protect main/production branches.** Limit who can bypass review and checks.
- **Avoid unnecessary force pushes.** Rewriting shared history can disrupt teammates.
- **Delete merged branches.** Keep repository navigation clean.
- **Tag releases intentionally.** A tag gives a stable reference to an important commit.

- **Automate tests.** CI makes quality checks repeatable.

The goal

A good Git history should help a new teammate understand how the system evolved, not merely prove that files changed.

22. Common Git Mistakes and How to Avoid Them

Mistake	Why it hurts	Better habit
Commit .env/secrets	Credentials can leak through history.	Ignore secrets and rotate immediately if exposed.
One huge commit	Hard to review, revert, and debug.	Split work into logical commits.
Direct push to protected main	Bypasses review/checks.	Use feature branch + PR.
Meaningless messages	History becomes useless.	State intent and scope clearly.
Blind force push	Can erase teammates' work.	Use safer workflows; coordinate if rewriting is required.
Mix formatting with feature logic	Creates noisy diffs and conflicts.	Separate mechanical formatting from logic changes.

Before every push

Ask: Is the branch correct? Are tests passing? Is the diff clean? Did I accidentally include a secret, log, binary, or local config?

23. Common Errors and Quick Fixes

remote origin already exists

```
git remote -v
git remote set-url origin <correct-url>
```

non-fast-forward / push rejected

The remote branch contains commits you do not have locally. Fetch/pull, integrate correctly, resolve conflicts if necessary, then push again. Do not use force by reflex.

```
git fetch origin
git pull --rebase origin main # only if your team uses/permits this workflow
git push origin main
```

permission denied (publickey)

Confirm the SSH key is loaded, the correct public key is added to GitHub, the repository URL is correct, and your account has access.

```
ssh -T git@github.com
git remote -v
```

repository not found

Check spelling, owner/organization, repository visibility, authentication, and access permissions. Private repositories require authorized credentials.

branch already exists

```
git branch
git switch existing-branch
```

24. More Troubleshooting Skills

Detached HEAD

You are checked out at a specific commit rather than on a normal branch. If you want to keep new work, create a branch before moving away.

```
git switch -c rescue/my-work
```

Accidentally committed the wrong file

If the commit is local and unpublished, you may amend it. If it is shared, prefer a follow-up commit or revert according to team policy.

```
git restore --staged unwanted-file
# or after correcting the local commit content
git commit --amend
```

Need to find who changed a line

```
git blame path/to/file.php
git log -- path/to/file.php
```

Need to inspect differences

```
git diff
git diff --staged
git diff main...feature/student-import
```

Debugging mindset

Read the exact error, inspect **git status**, identify your current branch and remote state, then choose the smallest safe command. Random commands often make Git problems worse.

25. Git Command Cheat Sheet

git init Create repository in current folder	git status Show working/staging state
git add . Stage changes	git commit -m "msg" Create commit
git log --oneline Compact history	git diff Unstaged diff
git diff --staged Staged diff	git branch List branches
git switch -c name Create + switch branch	git merge name Merge branch
git clone URL Clone remote repository	git fetch origin Download remote refs
git pull Fetch + integrate upstream	git push Upload local commits
git remote -v Show remotes	git stash Temporarily save work
git restore file Discard working changes	git revert HASH Reverse commit with new commit
git tag List/create tags	git blame file Show line-level commit authorship

Learn commands in context

Memorizing every flag is less important than understanding repository state. Use **git status** often and know what result you expect before running a command.

26. Real-World Practice Project

Scenario: You are building a Student Management System with a teammate. Practice the full professional flow instead of typing isolated commands.

Step 1 - initialize and make the first commit

```
mkdir student-management
cd student-management
git init
echo "# Student Management" > README.md
git add README.md
git commit -m "Initialize student management project"
```

Step 2 - connect GitHub

```
git remote add origin git@github.com:username/student-management.git
git branch -M main
git push -u origin main
```

Step 3 - build a feature branch

```
git switch -c feature/login
# create login files and tests
git status
git add .
git commit -m "Add student login flow"
git push -u origin feature/login
```

Step 4 - open PR, review, merge, and clean up

- Open a Pull Request from feature/login to main.
- Explain the change and how it was tested.
- Address review feedback and push updates to the same branch.
- Merge after checks pass.
- Update local main and delete the finished branch.

```
git switch main
git pull origin main
git branch -d feature/login
```

27. Practice Scenarios

Scenario A - urgent bug while feature is unfinished

You have uncommitted feature work, but production needs an urgent fix. Use stash or commit clean WIP on a dedicated branch, switch to the correct base branch, create a hotfix branch, fix/test, push, and open a PR.

Scenario B - your push is rejected

Do not immediately force push. Fetch/pull and inspect what changed remotely. Integrate according to team policy, resolve conflicts if needed, run tests, then push.

Scenario C - secret committed accidentally

Treat the secret as compromised. Rotate/revoke it immediately. Removing the latest file is not enough if the value exists in commit history. Follow the repository/security procedure for history cleanup where required.

Scenario D - one PR contains unrelated changes

Split or reorganize the branch/commits so the PR tells one understandable story. Reviewers should not have to mentally separate three unrelated tasks.

Your exercise

For each scenario, write the exact commands you would run and the safety checks you would perform before changing history.

28. Git & GitHub Interview Questions

Q. What is Git?

A distributed version control system used to track file history and coordinate changes.

Q. Git vs GitHub?

Git is the version-control tool; GitHub hosts Git repositories and adds collaboration features.

Q. What is staging?

Selecting the exact file versions that will be included in the next commit.

Q. What is a commit?

A recorded repository snapshot with metadata such as author, time, parent(s), and message.

Q. What is a branch?

A movable reference to a line of commits, commonly used to isolate work.

Q. What is a merge conflict?

A situation where Git cannot automatically combine competing changes.

Q. fetch vs pull?

Fetch downloads remote history; pull fetches and then integrates it into the current branch.

Q. reset vs revert?

Reset moves history references and may rewrite local history; revert creates a new commit that reverses an older commit.

Q. What is stash?

Temporary storage for uncommitted working changes.

Q. What is .gitignore?

Rules that tell Git which untracked paths should normally be ignored.

Q. What is a Pull Request?

A collaboration workflow for proposing, reviewing, testing, and merging branch changes.

Q. What is origin?

The conventional name for a repository remote, usually the primary clone source.

Q. What is HEAD?

A reference describing the current checked-out commit/branch position.

Q. What is a fast-forward merge?

Moving the target branch pointer forward when there is no divergent target history.

Q. What does git diff show?

Differences between repository states, depending on the arguments used.

29. Advanced Interview Notes

What is rebase?

Rebase replays commits onto a new base, producing a rewritten sequence of commit IDs. It can create a cleaner linear history but should be used carefully because rewriting commits that teammates already depend on can cause disruption.

Merge vs rebase

Merge preserves the true branch relationship and may add a merge commit. Rebase rewrites the feature commits onto another base. Neither is universally better; follow team conventions and avoid casually rebasing shared published history.

What is cherry-pick?

git cherry-pick <hash> applies the change introduced by a selected commit onto the current branch, creating a new commit. It is useful for targeted backports but should not replace a clear branching strategy.

What is a tag?

A tag is a named reference commonly used to mark releases such as v1.4.0. Annotated tags can also include tagger metadata and a message.

```
git tag -a v1.0.0 -m "Release v1.0.0"
git push origin v1.0.0
```

Interview strategy

Explain not only the command but also **when** you use it, **what state changes**, and **what risk exists** in a shared repository.

30. Final Revision: The Git Workflow

1. **Sync** → understand the latest target branch.
2. **Branch** → isolate one task.
3. **Edit** → make small, intentional changes.
4. **Inspect** → use status/diff.
5. **Stage** → select exactly what belongs in the commit.
6. **Commit** → record a meaningful snapshot.
7. **Push** → publish the branch.
8. **Pull Request** → review, test, discuss.
9. **Merge** → integrate approved work.
10. **Clean up** → update local branches and remove completed branches.

```
git fetch origin
git switch main
git pull origin main
git switch -c feature/example
# edit + test
git status
git diff
git add .
git diff --staged
git commit -m "Add example feature"
git push -u origin feature/example
```

Five commands to check first

When confused, start with: **git status**, **git branch --show-current**, **git log --oneline --decorate -10**, **git remote -v**, and **git diff**.

Final checklist

- I know which branch I am on.
- I understand what is staged and unstaged.
- My commit message explains intent.
- No credentials or private files are included.
- Tests/checks pass.
- I have reviewed the diff.
- I am not rewriting shared history without a reason and team agreement.

Quick Git Glossary

Term	Meaning
Repository	Project files plus Git history and metadata.
Commit	A recorded snapshot in repository history.
Branch	A movable reference used to isolate a line of work.
HEAD	The currently checked-out repository position.
Remote	A named reference to another repository location.
Origin	The conventional name for the primary remote.
Upstream	The branch/remote a local branch commonly tracks or syncs from.
PR	Pull Request - proposal to review and merge branch changes.

Mini Self-Check

- Which command shows staged, modified, and untracked files?
- What is the difference between **git fetch** and **git pull**?
- Why is **git revert** usually safer than reset for shared history?
- When should you create a feature branch?
- What should never be committed to a public repository?
- Why should a Pull Request stay focused on one logical change?

Self-Check Answer Key

1. **git status** shows the current branch plus staged, modified, and untracked files.
2. **git fetch** downloads remote history without integrating it; **git pull** fetches and then integrates into the current branch.
3. **git revert** creates a new reversing commit, so published history is preserved instead of rewritten.
4. Create a feature branch when you want to isolate a focused change from the shared target branch.
5. Never commit passwords, access tokens, private keys, production secrets, or other confidential credentials.
6. A focused PR is easier to review, test, understand, revert, and merge with fewer conflicts.

Next step

Create a small GitHub repository today and repeat the full flow: clone → branch → edit → status → add → commit → push → Pull Request → merge. Repetition is what turns commands into skill.

Keep Learning. Keep Building.

Practice Git on real projects. The more you use it, the more natural professional collaboration becomes.

NextTechEra Learning Resources • nexttechera.in