The Ultimate Real-world Git Commands Guide

Whether you're freelancing, collaborating in a team, or managing production-ready projects, this guide includes every Git command you'll likely use. More than just commands, you'll learn why, when, and how to use each one, in plain and practical terms.

1. `git config --global user.name "Your Name"`

- **Why**: Tell Git who you are — your name will be attached to every commit you make.
- **When**: First time setting up Git on a machine (global setting for all repositories).

How:

```
git config --global user.name "Tony Stark"
```

- **Example**:
  You just installed Git on your laptop. You set your identity globally to "Tony Stark".

2. `git config --global user.email "your@email.com"`

- **Why**: Set the email address linked to your commits (very important for GitHub contributions).
- **When**: Same as above, during Git setup.

How:

```
git config --global user.email "john@example.com"
```

- **Example**:
  Before pushing code to GitHub, you set your global email to "john@example.com" so your commits are recognized.

3. `git config user.name "Repo Specific Name"`

- **Why**: Override the global username for a specific project.
- **When**: When you work on multiple projects with different identities (e.g., company vs. personal).

How:

```
git config user.name "Company Name"
```

- **Example**:
  You freelance under your personal name but set a different username for your employer's project repo.

4. `git config --list`

- **Why**: View all your Git configurations.
- **When**: Whenever you want to check what settings are currently active.

How:

```
git config --list
```

- **Example**:
  You forgot which email you configured — run `git config --list` to check.

5. `git config --global core.editor "code --wait"`

- **Why**: Set your preferred text editor for Git (e.g., VS Code, Vim, Nano).
- **When**: Before writing commit messages, merge messages, or rebase edits.

How:

```
git config --global core.editor "code --wait"
```

- **Example**:
  You prefer VS Code. Setting this makes Git open commit messages in VS Code instead of Vim.

6. `git init`

- **Why**: Start a new local Git repository.
- **When**: When you create a new project and want version control from scratch.

How:

```
git init
```

- **Example**:
  You created a new folder called `my-app`. You run `git init` to track changes in it.

7. `git init -b master`

- **Why**: Initialize a repo and set the initial branch name (instead of "main").
- **When**: You want your project's default branch to be `master` (traditional default).

How:

```
git init -b master
```

- **Example**:
  Some legacy systems expect `master` instead of `main`. Use this to stay consistent.

8. `git add file.txt`

- **Why**: Stage a single file to be committed.
- **When**: After you modify or create a file and want to commit it.

How:

```
git add file.txt
```

- **Example**:
  You updated `file.txt` and want it included in your next commit.

9. `git add file.txt README.md`

- **Why**: Stage multiple specific files at once.
- **When**: You want to selectively commit certain files.

How:

```
git add file.txt README.md
```

- **Example**:
  You updated two files, and want both added together in the next commit.

10. `git status`

- **Why**: See the current state of your repo — what's staged, what's not, and what's untracked.
- **When**: After making changes or before committing.

How:

```
git status
```

- **Example**:
  You edited files and want to check what's ready to commit.

11. `git add *.java`

- **Why**: Add all Java files at once using wildcard patterns. (i.e., add all similar type of files at once using *.
- **When**: You updated multiple `.java` files and want to stage them quickly.

How:

```
git add *.java
```

- **Example**:
  You refactored 10 Java files and don't want to add them one by one.

12. `git status -s`

- **Why**: Short version of `git status` (compact view).
- **When**: When you want a quick glance at changes without clutter.

How:

```
git status -s
```

- **Example**:
  You want a quick overview — staged files show as `A`, modified files as `M`.

13. `git add .`

- **Why**: Stage **all** changes (new, modified, deleted files).
- **When**: Before committing when you're ready to include everything.

How:

```
git add .
```

- **Example**:
  After finishing a feature, you use `git add .` to stage all changes at once.

14. `git commit -m "Your message"`

- **Why**: Save a snapshot of staged changes into the repository.

- **When**: After adding files and you're ready to record a change with a clear message.

How:

```
git commit -m "Added login feature"
```

- **Example**:
  You built the login system and commit it with the message "Added login feature".

15. `git status --help`

- **Why**: Open detailed help about the `git status` command.
- **When**: If you're unsure what options `git status` supports.

How:

```
git status --help
```

- **Example**:
  You're curious if `git status` has advanced options — check help for full info.

16. `git status -h`

- **Why**: Quick help (shorter than `--help`) for the `git status` command.
- **When**: Need a brief overview of options without full documentation.

How:

```
git status -h
```

- **Example**:
  Quickly remind yourself what flags you can use with `git status`.

17. `git commit -a -m "Your message"`

- **Why**: Skip `git add` — automatically stage all **tracked** changes and commit them in one step.
- **When**: After modifying tracked files and you want a quick commit.

How:

```
git commit -a -m "Fixed bug in signup form"
```

- **Example**:
  You fixed bugs in several `.py` files — use `-a -m` to save time.

Note: Only works for **modified/deleted tracked** files, not new untracked ones.

18. `git rm --cached .classpath`

- **Why**: Remove a file **from Git tracking** but **keep it locally**.
- **When**: You accidentally added a file you don't want in the repo (e.g., `.classpath`, build files).

How:

```
git rm --cached .classpath
```

- **Example**:
  `.classpath` was mistakenly committed. You remove it from Git but it remains on your machine.

19. `git rm -r --cached bin/`

- **Why**: Stop tracking an **entire folder** (like build folders) without deleting it locally.
- **When**: You want to untrack all files in a directory.

How:

```
git rm -r --cached bin/
```

- **Example**:
  You accidentally committed the `bin/` directory with compiled files. Use this to untrack it.

20. `.gitignore`

- **Why**: Tell Git **which files/folders to ignore** permanently (like `node_modules/`, `*.log`, `bin/`).
- **When**: Right at the start of a project or whenever you want to ignore unwanted files.
- **How**:

Create a `.gitignore` file:

```
 node_modules/

*.log

bin/
```

- **Example**:
  You don't want build files, logs, or dependencies to be tracked — list them in `.gitignore`.

21. `git log`

- **Why**: See a detailed history of all commits.
- **When**: To review what changes were made, when, and by whom.

How:

```
 git log
```

- **Example**:
  You want to check the commit history before merging or troubleshooting.

22. `git log -2`

- **Why**: Show only the **last 2 commits**. (Replace 2 with any number.)
- **When**: You want a quick look at recent commits without full history.

How:

```
git log -2
```

- **Example**:
  You want to see what you or a teammate just committed recently.

23. `git log --pretty=oneline`

- **Why**: Show each commit as a **single line** (hash + message) — super clean!
- **When**: When you need a quick, compact overview of all commits.

How:

```
git log --pretty=oneline
```

- **Example**:
  Preparing a changelog, you want a fast, easy-to-read commit list.

24. `git commit --amend`

- **Why**: **Edit** the last commit (modify the message or add more changes).
- **When**: You realize you made a typo or forgot to add something **right after** committing.

How:

```
git commit --amend
```

- **Example**:
  You committed "Fixd login" instead of "Fixed login". Use `--amend` to correct it.

Quick Tip: Use `git commit --amend` carefully if you've already pushed!
(It rewrites history — better avoid on shared branches unless you know what you're doing.)

25. `git commit --amend --no-edit`

- **Why**: Update the **content** of the last commit **without changing** its message.
- **When**: You staged new changes and want to squeeze them into the last commit without editing the message.

How:

```
git commit --amend --no-edit
```

- **Example**:
  You forgot to include a small file — stage it and run `--amend --no-edit` to fix the previous commit.

26. `touch filename.txt`

- **Why**: Create an empty file quickly from the command line.
- **When**: Starting a new file like `.gitignore`, `README.md`, etc.

How:

```
touch README.md
```

- **Example**:
  You want to add a README to your project — run `touch README.md`.

Note: `touch` is a Unix/Linux/macOS command. Windows users can use `echo.> filename.txt` or Git Bash.

27. `git restore --staged src/filename.extension`

- **Why**: **Unstage** a file that you accidentally added with `git add`.
- **When**: Before committing, if you realize a file should not be staged.

How:

```
git restore --staged src/app.js
```

- **Example**:
  You accidentally staged a debug file. Use this to unstage it before committing.

28. `git restore src/filename.extension`

- **Why**: Discard **local changes** in a file — revert it back to the last committed version.
- **When**: You made a mistake and want to throw away all your local edits to that file.

How:

```
git restore src/app.js
```

- **Example**:
  You messed up `app.js` beyond repair — restore it to the last good committed version.

29. `git remote -v`

- **Why**: See the **list of remote repositories** your local repo is connected to.
- **When**: You want to check where your repo will push/pull from.

How:

```
git remote -v
```

- **Example**:
  You cloned a project and want to see the GitHub URL it's connected to.

30. `git remote add origin <repository-URL>`

- **Why**: **Link** your local project to a **remote repository** (like GitHub, GitLab).
- **When**: First time you're pushing your project to a remote server.

How:

```
git remote add origin https://github.com/username/repo.git
```

- **Example**:
  After `git init`, you use this command to connect it to your GitHub repo.

31. `git push origin -u master`

- **Why**: **Push** your code to the remote for the first time and **set upstream** for easy future pushes.
- **When**: First time pushing your code after linking to remote.

How:

```
git push origin -u master
```

- **Example**:
  You created a local project, now you want it on GitHub under `master` branch.

Note: Now `main` is the default branch name in many places. Adjust accordingly:

```
git push origin -u main
```

32. `git clone <repository-URL>`

- **Why**: **Download** a complete copy of a remote repo to your computer.
- **When**: You want to start working on an existing project.

How:

```
git clone https://github.com/username/repo.git
```

- **Example**:
  You join a new project team, and you clone the repo from GitHub to your machine.

33. SSH connection with Git

- **Why**: Use **secure**, **passwordless** connection with GitHub/GitLab.
- **When**: To avoid entering your username/password every time you push.

How:

1. Generate an SSH key: ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
2. Add your SSH key to GitHub/GitLab.
3. Clone or set remote using SSH: git clone git@github.com:username/repo.git

- **Example**:
  You work daily with GitHub — using SSH saves time (and it's safer).

34. `git branch <name>`

- **Why**: **Create** a new branch (for a feature, bug fix, etc).
- **When**: You want to work on something without disturbing the `main` code.

How:

```
git branch feature-login
```

- **Example**:
  You create a `feature-login` branch to build a login page separately.

35. `git branch`

- **Why**: See **all local branches**.
- **When**: Quickly check which branch you're on and what others exist.

How:

```
git branch
```

- **Example**:
  You forgot which branch you're working in — this shows it.

36. `git branch -r`

- **Why**: See **remote branches** only.
- **When**: You want to know what branches exist on GitHub without pulling them.

How:

```
git branch -r
```

- **Example**:
  Checking if a teammate already pushed a `feature-payment` branch.

37. `git branch -a`

- **Why**: See **all branches** — local and remote both.
- **When**: Full overview of the project branches.

How:

```
git branch -a
```

- **Example**:
  You're checking project branches to switch or pull updates.

38. `git checkout -b <branch-name>`

- **Why**: **Create and switch** to a new branch in one step.
- **When**: You want to start new work immediately on a new branch.

How:

```
git checkout -b feature-dashboard
```

- **Example**:
  Save time — no need to separately create then checkout.

39. `git log --all`

- **Why**: Show **commit history** from **all branches**.
- **When**: You want to see everything, not just the current branch.

How:

```
git log --all
```

- **Example**:
  Investigating why a feature disappeared — you want the full picture.

40. `git log <filename>`

- **Why**: Show **only the commits related to a specific file**.
- **When**: You want to track the history of a particular file.

How:

```
git log app.py
```

- **Example**:
  See who changed `app.py` and what changes they made over time.

41. `git checkout master`

- **Why**: Switch back to the **master/main branch**.

- **When**: After working on feature branches, you want to return to main work.

How:

```
git checkout master
```

- **Example**:
  You finished your feature, now you need to merge it into `master`.

42. `git checkout <commit-hash>`

- **Why**: Go back to a **specific commit state** temporarily.
- **When**: Debugging or checking how the code looked at a certain point.

How:

```
git checkout 0a1b2c3d
```

- **Example**:
  You suspect a bug was introduced — checkout an older commit to investigate.

43. `git branch -d <branch-name>`

- **Why**: **Delete** a branch locally (only if fully merged).
- **When**: After merging a feature branch and cleaning up.

How:

```
git branch -d feature-login
```

- **Example**:
  After merging `feature-login` into `master`, you delete it to keep things clean.

If the branch isn't merged, use `-D` (force delete), but be careful.

44. `git push origin --delete <branch-name>`

- **Why**: **Delete** a branch from the remote (GitHub, GitLab).
- **When**: Cleaning up remote branches after work is finished.

How:

```
git push origin --delete feature-login
```

- **Example**:
  After cleaning up your local branch, you remove it from GitHub too.

45. `git switch master`

- **Why**: **Switch** to another branch easily.
- **When**: Move between branches.

How:

```
git switch master
```

- **Example**:
  You finished work on `feature-payment` and now switch back to `master`.

46. `git switch -c <new-branch-name>`

- **Why**: **Create and switch** to a new branch (shortcut).
- **When**: Start a new feature without disturbing existing work.

How:

```
git switch -c feature-profile
```

- **Example**:
  You immediately create `feature-profile` branch and start working.

47. `git switch -`

- **Why**: **Quickly switch back** to the previous branch.
- **When**: You often jump back and forth between two branches.

How:

```
git switch -
```

- **Example**:
  Moving quickly between `master` and `feature-dashboard` during testing.

48. `git fetch`

- **Why**: **Download** changes from remote **without merging**.
- **When**: You want to see what’s changed remotely without affecting your local branch yet.

How:

```
git fetch
```

- **Example**:
  Check for new commits on GitHub before deciding to merge or pull.

49. `git merge`

- **Why**: **Combine** another branch into your current branch.
- **When**: You want to bring completed features into your main branch.

How:

```
git merge feature-login
```

- **Example**:
  Merge the finished `feature-login` branch into `master`.

50. `git pull = git fetch + git merge`

- **Why**: **Update** your local branch with changes from remote in **one step**.
- **When**: Regularly syncing your work with teammates.

How:

```
git pull
```

- **Example**:
  You run `git pull` before starting your day to get everyone's latest work.

51. `git reset --hard HEAD~1`

- **Why**: **Delete** last commit and **discard changes** completely.
- **When**: You made a bad commit and want to erase it totally.

How:

```
git reset --hard HEAD~1
```

- **Example**:
  Oops — you committed wrong code and want to rewind like it never happened.

52. `git pull --rebase origin master`

- **Why**: **Update your branch by replaying your changes** on top of latest master.
- **When**: You want a **cleaner** history without unnecessary merge commits.

How:

```
git pull --rebase origin master
```

- **Example**:
  Before pushing your feature, you rebase to make it look like you developed it after the latest master.

53. `git rebase --continue`

- **Why**: **Continue rebasing** after solving conflicts.
- **When**: You fix a merge conflict during rebase and want to proceed.

How:

```
git rebase --continue
```

- **Example**:
  You solved a conflict in `app.py` — now continue with the rebase process.

54. `git rebase --skip`

- **Why**: **Skip** the conflicting commit during rebase.
- **When**: You decide a certain commit is no longer needed.

How:

```
git rebase --skip
```

- **Example**:
  If fixing a conflict doesn't make sense for a removed file, you skip it.

55. `git rebase --abort`

- **Why**: **Cancel** the rebase and go back to how things were.
- **When**: You realize rebasing is messy or unnecessary.

How:

```
git rebase --abort
```

- **Example**:
  You started rebasing but ran into a mess — so you cancel and rethink.

56. `git push -f`

- **Why**: **Force push** your local branch over remote.
- **When**: After a rebase or destructive change.

How:

```
git push -f
```

- **Example**:
  After rewriting history (like with `rebase`), you use force push to update GitHub.

Be very careful — it overwrites history.

57. `git push --force-with-lease`

- **Why**: **Safe force push** — it fails if someone else pushed meanwhile.
- **When**: Safer alternative to `git push -f`.

How:

```
git push --force-with-lease
```

- **Example**:
  Protects teammates' commits while still allowing you to push rebased code.

58. `git rebase -i HEAD~3`

- **Why**: **Interactive rebase** to **edit, squash, or reorder commits**.
- **When**: You want a **perfect clean commit history** before pushing.

How:

```
git rebase -i HEAD~3
```

- **Example**:
  Squashing 3 messy commits into a single clean commit before merging feature.

59. `git reset` (soft, mixed, hard)

- **Why**: **Undo commits** in different ways.
- **When**: Depending on whether you want to keep, unstage, or discard your changes.

Mode	What Happens	Command
Soft	Undo commit, keep changes staged	`git reset --soft HEAD~1`
Mixed	Undo commit, keep changes unstaged	`git reset --mixed HEAD~1` *(default)*
Hard	Undo commit, **delete** all changes	`git reset --hard HEAD~1`

Example:

- Soft: Realized a wrong commit but code is fine — fix commit message.
- Mixed: Code is fine but need to edit some files before recommitting.
- Hard: Need to throw away everything.

60. `git fetch --all`

- **Why**: **Fetch all remotes** and all branches without merging.
- **When**: You work with multiple remotes (like upstream + origin).

How:

```
git fetch --all
```

- **Example**:
  Updating local copies of origin, upstream, etc., all at once.

61. `git stash`

- **Why**: Temporarily **save** your uncommitted changes.
- **When**: You need to **quickly switch** to another branch without losing work.

How:

```
git stash
```

- **Example**:
  You're halfway coding a new feature and your boss says "URGENT BUGFIX!" — stash your WIP (Work in Progress) and jump to fix the bug.

62. `git switch -`

- **Why**: **Quick switch** back to your last branch.
- **When**: After stashing, you might want to jump back fast.

How:

```
git switch -
```

- **Example:** You are on feature-branch, then you switch to main for a quick pull. Now, to jump back to feature-branch, just run: ***git switch -*** (Instantly returns you to feature-branch.)

63. `git stash apply`

- **Why**: **Apply** the latest stashed changes back into your working directory (but stash remains).
- **When**: After switching back and you want to continue work.

How:

```
git stash apply
```

- **Example**:
  Apply your saved work without deleting it from stash.

64. `git stash list`

- **Why**: **See** all your stashed changes.
- **When**: You forgot what you stashed and want a reminder.

How:

```
git stash list
```

- **Example**:
  Shows stashes like `stash@{0}`, `stash@{1}`, etc.

65. `git stash apply stash@{0}`

- **Why**: **Apply a specific** stash.
- **When**: You have multiple stashes and want one particular one.

How:

```
git stash apply stash@{1}
```

- **Example:** You saved two stashes, but only want to apply the first one: git stash apply stash@{0}

66. `git stash save "message"`

- **Why**: **Save stash with a description** to remember it easily.
- **When**: You want to document why you stashed.

How:

```
git stash save "WIP: Login feature"
```

67. `git stash pop`

- **Why**: **Apply and delete** the latest stash.
- **When**: You’re ready to continue working and don't need the stash anymore.

How:

```
git stash pop
```

- **Example**:
  "Pop it" back into your working directory and remove from stash list.

68. `git stash -a`

- **Why**: **Stash everything** including **untracked files** (new files).
- **When**: Your branch has many new files not added to Git yet.

How:

```
git stash -a
```

- **Example:** You have both tracked and untracked files that you don't want to commit yet, so you run: ***git stash -a***

  This stashes all changes, including those new files, keeping your working directory clean.

69. `git stash -u`

- **Why**: **Stash tracked + untracked** files (shortcut).
- **When**: You have unfinished files and even new files not added yet.

How:

```
git stash -u
```

- **Example:** You create a new file new-file.txt but haven't added it to staging. To stash it along with your tracked changes: ***git stash -u***

70. `git stash branch <branch-name>`

- **Why**: **Create a new branch** from a stash.
- **When**: You want to resume work later in a separate branch.

How:

```
git stash branch bugfix-navbar
```

71. `git stash drop stash@{0}`

- **Why**: **Delete** a specific stash manually.
- **When**: You know you don't need it anymore.

How:

```
git stash drop stash@{0}
```

72. `git stash clear`

- **Why**: **Delete all** stashes at once.
- **When**: Cleaning time! You want to start fresh.

How:

```
git stash clear
```

73. `git reflog`

- **Why**: **Track your full Git history** — even commits you deleted or moved.
- **When**: You need to **recover a lost commit** or undo some git disaster.

How:

```
git reflog
```

- **Example**:
  You accidentally `reset --hard`, and want to recover your work.

74. `git log -g`

- **Why**: Shortcut to view the reflog history, showing the graphical commit history.
- **When**: When you need to inspect commit history through the reflog with a clearer, graphical view.

How:

```
git log -g
```

- **Example:** View your commit history and changes made to branches over time: ***git log -g*** This will show you a graphical view of all actions that happened in the reflog.

75. `git log <branch-name>`

- **Why**: View commits of a **specific branch**.
- **When**: Understand what happened only on `feature-login`, not everywhere.

How:

```
git log feature-login
```

76. `git reflog --since="1.hour"`

- **Why**: **See reflog** entries from the past X time.
- **When**: Quickly filter recent changes.

How:

```
git reflog --since="1.hour"
```

77. `git log --oneline -3`

- **Why**: See **last 3 commits** in a **short format**.
- **When**: You need a quick, compact view.

How:

```
git log --oneline -3
```

78. `git cherry-pick <commit-hash>`

- **Why**: **Copy a specific commit** from one branch into another.
- **When**: You don't want to merge everything, just that one golden commit.

How:

```
git cherry-pick abc1234
```

- **Example**:
  You made a perfect bugfix on `dev`, now you want it on `master` too — cherry-pick it!

79. `git cherry-pick --abort`

- **Why**: **Cancel cherry-pick** in case of conflict or wrong operation.
- **When**: You changed your mind halfway.

How:

```
git cherry-pick --abort
```

80. `git cherry-pick --continue`

- **Why**: **Continue** cherry-picking after solving a conflict.
- **When**: A conflict happens while cherry-picking.

How:

```
git cherry-pick --continue
```

81. `git blame <file>`

- **Why**: Find out **who changed what** and **when** inside a file.
- **When**: Debugging bugs or understanding history.

How:

```
git blame src/app.py
```

- **Example**:
  "Who wrote this suspicious line of code?"
  → `git blame` will show the author's name, commit ID, and date for each line.

82. `git tag`

- **Why**: **Mark** important points like releases (`v1.0`, `v2.0`) in history.
- **When**: Releasing a version or milestone.

How:

```
git tag v1.0
git tag -a v1.0 -m "First stable release"
```

- **Example**:
  Tagging a production release after all tests pass.

83. `git shortlog`

- **Why**: Summarized view of **who made how many commits**.
- **When**: Writing release notes or reviewing contributions.

How:

```
git shortlog
```

- **Example**:
  Shows a neat author-wise commit summary.

84. `git clean`

- **Why**: **Remove untracked files** (not in Git).
- **When**: Cleanup junk files from working directory.

How:

```
git clean -f
git clean -fd   # remove untracked folders too
```

- **Example**:
  After build processes that create unwanted temp files.

85. `git bisect`

- **Why**: **Find the exact commit** that introduced a bug.
- **When**: Hunting down sneaky bugs!

How:

```
git bisect start

git bisect bad          # Mark current commit as bad

git bisect good v1.0    # Mark known good commit
```

- Then Git auto-narrows down by asking "good/bad" after each checkout.
- **Example**:
  Bug appeared between v1.0 and now — bisect finds *which commit* broke it..

86. `git revert <commit>`

- **Why**: **Create a new commit** that undoes the changes of a past commit.
- **When**: You **can't delete history** but want to *reverse* a bad commit.

How:

```
git revert abc1234
```

- **Example**:
  Production bug? Revert the breaking commit without rewriting history.

87. `git worktree`

- **Why**: **Work on multiple branches** *at the same time in different folders without cloning your repo again.*

- **When**: You need to test a bugfix or feature on a different branch without affecting your main working directory.

How:

```
git worktree add ../bugfix-branch bugfix
```

- **Example**:
  Create a new folder for bugfix-branch and start working on it, while your main directory stays focused on your original work.

88. `git archive`

- **Why**: **Export** your project as a **zip/tar** file.
- **When**: Sending code to someone without `.git` history.

How:

```
git archive -o project.zip HEAD
```

89. `git diff origin/main`

- **Why**: Compare your branch with **remote main** branch.
- **When**: Before making pull requests, check differences.

How:

```
git diff origin/main
```

- **Example:** You're on feature-branch. To see what changes differ from main before pushing: ***git diff origin/main***

90. `git fetch`

- **Why**: Download new commits from the remote repository without merging them into your local branch.
- **When**: Stay updated on changes without affecting your local code.

How:

```
git fetch origin
```

- **Example:** You want to check for new commits on main without switching branches: ***git fetch origin***

(Now, you can review the changes before merging or rebasing.)

91. `git remote`

- **Why**: Manage your remote repositories — add, remove, rename, or check remotes.
- **When**: When you're setting up or changing the remote repository links for your project.

How:

```
git remote -v          # List all remotes with their URLs

git remote add origin https://github.com/user/repo.git  # Add a new remote
```

92. `git config`

- **Why**: Set or edit Git configuration settings.
- **When**: Personalize Git behavior (username, email, editor, aliases, etc.)

How:

```
git config --global user.name "Your Name"

git config --global user.email "you@example.com"
```

93. `git credential-cache`

- **Why**: Temporarily **save your GitHub/remote credentials** to avoid repeated logins.
- **When**: Use this when you want to cache your credentials for a specified time, so you don't have to type your username/password on every push/fetch.

How:

```
git config --global credential.helper cache

git config --global credential.helper 'cache --timeout=3600'
```

This saves your credentials for 1 hour (3600 seconds) before asking again.

94. `git submodule`

- **Why**: **Include another Git repo inside yours** as a submodule.
- **When**: Reusing external libraries/projects inside your main project.

How:

```
git submodule add https://github.com/other/project.git vendor/project

git submodule update --init
```

- **Example**:
  Embedding open-source libraries inside your own repo: ***git submodule add https://github.com/example/lib.git vendor/lib***

95. `git apply`

- **Why**: Apply a patch file (changes) generated elsewhere.
- **When**: Someone sends you .patch files instead of pull requests (PRs).

How:

```
git apply patchfile.patch
```

- **Example:** You receive a bugfix.patch from a colleague — apply it to your local repo: ***git apply bugfix.patch***

(Instantly applies the changes from the patch to your working directory.)

96. `git format-patch`

- **Why**: **Create patch files** from commits, useful for sending code contributions.
- **When**: When you need to send commits as patches for review or contribution via email.

How:

```
git format-patch -1
```

```
git format-patch origin/main
```

- **Example:** Create a patch for the latest commit: ***git format-patch -1*** Or create patches from commits since main: ***git format-patch origin/main***

Congratulations! 🎊

You've just mastered the Git essentials that developers use every single day — just like a pilot knows every control in the cockpit.

If you found this guide helpful, let's stay connected and grow together:

📌 Follow me on LinkedIn for more real-world developer insights. 👇

LinkedIn | *https://www.linkedin.com/in/harvilade2102?*

📌 Explore more open-source projects and give a star ⭐ if you find them useful:

GitHub | *https://github.com/Harvi-Lade*

Suggestions welcome!

If you have any tips, corrections, or ideas to make this guide even better —

I'd love to hear them!

- Drop a comment on LinkedIn
- Or open an issue on GitHub

Let's build better resources together!

Thank you for your time and Happy Coding! 😊