

git//field-manual

COMMAND REFERENCE

by @contenintech

FOUNDATIONS

01 · Identity & Config

02 · Init & Clone

DAILY DRIVER

03 · Snapshotting

04 · History & Inspection

05 · Branching & Merging

06 · Remotes

FIXING THINGS

07 · Undoing Changes

08 · Stashing

09 · Rebasing

ADVANCED

10 · Tags & Releases

11 · Debugging Tools

12 · Feature Branch Workflow

13 · Aliases & .gitignore

```
~/repo (main) $ cat README
```

The git Field Manual

Every command a software engineer actually reaches for day to day — setup, snapshotting, branching, remotes, undo tools, rebasing, and the debugging commands nobody remembers until they need them. Each entry includes what it does, the exact syntax, and a runnable example.

01 Identity & Configuration

Run once per machine (or per repo) before you commit anything.

git config

[SETUP](#)

Sets your identity so every commit is signed with a name and email. Use `--global` to apply it to every repo on the machine, or drop it to scope the setting to the current repo only.

```
git config --global user.name "Aditi Sharma"
git config --global user.email "aditi@company.com"
git config --global core.editor "code --wait" # set VS Code as commit editor
```

List everything currently configured, or check one value:

```
git config --list
git config user.email
```

02 Init & Clone

Starting a repository from zero, or pulling down one that already exists.

git init

[START](#)

Turns the current folder into a git repository by creating a hidden `.git` directory that tracks history.

```
cd my-project
git init
```

git clone

[START](#)

Downloads a full copy of a remote repository — history included — into a new local folder, and automatically wires up `origin` as the remote.

```
git clone https://github.com/org/repo.git
git clone https://github.com/org/repo.git my-folder-name
git clone --depth 1 https://github.com/org/repo.git # shallow clone, latest commit only
```

Tip: use a shallow clone for CI pipelines where you only need the latest snapshot — it's much faster on large repos.

03 Basic Snapshotting

The add → commit loop you run dozens of times a day.

git status

[INSPECT](#)

Shows the current branch, staged changes, unstaged changes, and untracked files. The first command to run when you're unsure what state you're in.

```
git status
git status -s # short format: M = modified, ?? = untracked, A = added
```

git add

[STAGE](#)

Moves changes from your working directory into the staging area, marking them ready to be committed.

```
git add index.js      # stage one file
git add src/          # stage a whole folder
git add .             # stage everything changed in this directory
git add -p            # interactively choose chunks (hunks) to stage
```

git commit

[SAVE](#)

Records the staged changes as a new snapshot in history, with a message describing what changed and why.

```
git commit -m "Fix null pointer in auth middleware"
git commit -am "Update tests"  # stage tracked files + commit in one step
git commit --amend -m "Corrected message"  # rewrite the last commit
```

Careful: only --amend a commit that hasn't been pushed yet — it rewrites history.

git diff

[INSPECT](#)

Shows line-by-line changes. Plain diff compares working directory vs. the last commit; --staged compares what's staged vs. the last commit.

```
git diff              # unstaged changes
git diff --staged     # staged changes, ready to commit
git diff main feature # compare two branches
```

git rm / git mv

[EDIT](#)

Removes or renames a tracked file and stages that change automatically, keeping the index in sync.

```
git rm old-config.json
git mv utils.js helpers.js
```

04 History & Inspection

Reading the story of a repo — who changed what, and when.

git log

[INSPECT](#)

Lists commits, newest first, with author, date, and message. Formatting flags turn it from a wall of text into a readable graph.

```
git log
git log --oneline           # one line per commit
git log --oneline --graph --all  # ASCII branch graph, all branches
git log -p -- path/to/file.js   # full diffs for one file's history
git log --author="Aditi"        # filter by author
```

git show

[INSPECT](#)

Displays the full diff and metadata for a single commit.

```
git show a1b2c3d
git show HEAD~2  # the commit two before the current one
```

git blame

[INSPECT](#)

Shows which commit and author last touched each line of a file — the go-to command when you need to ask "why does this line exist?"

```
git blame src/auth.js
git blame -L 40,60 src/auth.js  # only lines 40–60
```

05 Branching & Merging

Isolating work in progress, then bringing it back together.

git branch

[BRANCH](#)

Lists, creates, or deletes branches without switching to them.

```
git branch           # list local branches
git branch -a        # list local + remote branches
git branch feature/login  # create a new branch
git branch -d feature/login # delete, only if merged
git branch -D feature/login # force delete, even if unmerged
```

git switch / checkout

BRANCH

switch is the modern, safer command for changing branches. checkout is the older all-purpose command that still works everywhere.

```
git switch feature/login      # move to an existing branch
git switch -c feature/payments # create + move to a new branch
git checkout feature/login    # equivalent, older syntax
git checkout -b feature/payments # equivalent, older syntax
```

git merge

COMBINE

Integrates changes from one branch into the branch you're currently on, creating a merge commit if histories have diverged.

```
git switch main
git merge feature/login
git merge --no-ff feature/login # always create a merge commit, even if fast-forward is possible
```

If files conflict, git marks the file with <<<<<< / ===== / >>>>>> blocks. Edit to resolve, then `git add` the file and `git commit`.

06 Working with Remotes

Syncing your local repo with GitHub, GitLab, or wherever the team's copy lives.

git remote

SETUP

Manages the shortcuts (names) git uses for remote repository URLs.

```
git remote -v                # list remotes and their URLs
git remote add origin git@github.com:org/repo.git
git remote set-url origin git@github.com:org/new-repo.git
```

git fetch

SYNC

Downloads new commits and branches from the remote, but does **not** merge them into your working branch — a safe way to see what changed first.

```
git fetch origin
git fetch --all
```

git pull

SYNC

Fetches and immediately merges (or rebases) the remote branch into your current branch. Equivalent to `git fetch` + `git merge`.

```
git pull origin main
git pull --rebase origin main # replay your commits on top instead of merging
```

Uploads your local commits to the remote branch.

```
git push origin main
git push -u origin feature/login      # push + remember this upstream for future plain "git push"
git push --force-with-lease           # safer force-push, fails if remote has new commits you don't have
```

Avoid plain --force on shared branches — it can silently delete a teammate's commits. Use --force-with-lease instead.

07Undoing Changes

The commands that get you out of trouble — know the difference before you run them.

COMMAND	WORKING DIR	STAGING AREA	HISTORY
reset --soft	unchanged	unchanged	moves HEAD back
reset --mixed	unchanged	cleared	moves HEAD back
reset --hard	wiped to match	cleared	moves HEAD back
revert	unchanged	unchanged	adds a new "undo" commit

Moves the current branch pointer to an earlier commit. Use --hard only when you're certain — it discards uncommitted work permanently.

```
git reset --soft HEAD~1      # undo last commit, keep changes staged
git reset HEAD~1             # undo last commit, unstage changes (default = --mixed)
git reset --hard HEAD~1      # undo last commit, delete the changes entirely
```

Creates a brand-new commit that reverses a previous one — safe to use on shared/pushed history since nothing is rewritten.

```
git revert a1b2c3d
git revert --no-commit HEAD~3..HEAD  # revert a range, stage but don't commit yet
```

The modern, focused command for discarding local edits or unstaging files, without touching commit history.

```
git restore file.js          # discard unstaged edits to a file
git restore --staged file.js  # unstage a file, keep the edits
```

git clean

CLEANUP

Deletes untracked files from the working directory — good for wiping build artifacts.

```
git clean -n      # dry run: show what would be deleted
git clean -fd     # actually delete untracked files + directories
```

08 Stashing

Parking half-finished work so you can switch context and come back later.

git stash

WIP

Temporarily shelves uncommitted changes, giving you a clean working directory, and stores them on a stack you can reapply anytime.

```
git stash          # stash tracked changes
git stash -u       # also stash untracked files
git stash save "wip: login form"
git stash list     # view the stash stack
git stash pop      # reapply latest stash + remove it from the stack
git stash apply stash@{2} # reapply a specific stash, keep it in the stack
git stash drop stash@{2} # delete a stash without applying it
```

09 Rebasing

Rewriting commit history for a cleaner, linear timeline.

git rebase

REWRITE

Replays your branch's commits on top of another branch, avoiding a merge commit and keeping history linear.

```
git switch feature/login
git rebase main          # replay feature commits on top of latest main
git rebase --continue    # after resolving a conflict mid-rebase
git rebase --abort       # bail out and return to the pre-rebase state
```

Golden rule: never rebase a branch that other people have already pulled from — it rewrites commit hashes.

git rebase -i

[CLEANUP](#)

Interactive rebase — lets you squash, reorder, reword, or drop commits before sharing a branch.

```
git rebase -i HEAD~5
```

Opens an editor listing the last 5 commits, each prefixed with pick. Change the keyword to reshape history:

```
pick    a1b2c3d Add login form
squash  e4f5g6h Fix typo
reword  h7i8j9k Add validation
drop    k0l1m2n Remove debug log
```

10 Tags & Releases

Marking specific commits as meaningful points, like version releases.

git tag

[RELEASE](#)

Creates a permanent, named pointer to a specific commit — typically used for version numbers.

```
git tag v1.0.0                # lightweight tag on current commit
git tag -a v1.0.0 -m "First stable release" # annotated tag, recommended
git tag                       # list all tags
git push origin v1.0.0        # push one tag
git push origin --tags        # push all tags
```

11 Debugging & Advanced Tools

The lesser-known commands that solve very specific, very annoying problems.

git cherry-pick

[ADVANCED](#)

Applies one specific commit from another branch onto your current branch, without merging everything else.

```
git cherry-pick a1b2c3d
git cherry-pick a1b2c3d..h7i8j9k # a range of commits
```

git bisect

ADVANCED

Binary-searches through history to find the exact commit that introduced a bug.

```
git bisect start
git bisect bad           # current commit is broken
git bisect good v1.2.0   # this older tag was working
# git checks out a midpoint commit – you test it, then run:
git bisect good  # or
git bisect bad
# repeat until git names the first bad commit
git bisect reset        # when done
```

git reflog

RECOVERY

Logs every place HEAD has pointed, even commits erased by a hard reset — the safety net when you think you've lost work.

```
git reflog
git reset --hard HEAD@{2}  # jump back to a state from the reflog
```

git submodule

ADVANCED

Embeds one git repository inside another as a linked dependency, pinned to a specific commit.

```
git submodule add https://github.com/org/lib.git libs/lib
git submodule update --init --recursive  # after cloning a repo that has submodules
```

git worktree

ADVANCED

Checks out another branch into a separate folder, so you can work on two branches at once without stashing.

```
git worktree add ../hotfix hotfix/urgent-bug
git worktree list
git worktree remove ../hotfix
```

12A Standard Feature Branch Workflow

How the commands above chain together on a real team.



```
git switch main
git pull origin main
git switch -c feature/payment-retry

# ...make changes...
git add .
git commit -m "Add exponential backoff to payment retries"
git push -u origin feature/payment-retry

# open a PR on GitHub/GitLab, get it reviewed

# keep the branch up to date with main before merging
git fetch origin
git rebase origin/main

# after the PR merges
git switch main
git pull origin main
git branch -d feature/payment-retry
```

13 Aliases & .gitignore

Small quality-of-life setups that save keystrokes every single day.

git config alias

PRODUCTIVITY

Creates short custom shortcuts for commands you type constantly.

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.st status
git config --global alias.lg "log --oneline --graph --all"
# now you can just run:
git st
git lg
```

.gitignore

SETUP

A plain-text file listing paths git should never track — dependencies, build output, secrets, editor files.

```
# .gitignore
node_modules/
dist/
.env
*.log
.DS_Store
```

If a file was already tracked before being added to .gitignore, remove it from tracking without deleting it locally:

```
git rm --cached .env
```

