# Git & GitHub Commands Reference Guide

Complete Command Reference

January 2, 2026

## Contents

# 1 Getting Started - Initial Setup

## 1.1 Configuration Commands

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

  **What it does:** Sets your name for all Git commits on your system.

  **When to use:** First time setting up Git on a new computer.

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

  **What it does:** Sets your email for all Git commits on your system.

  **When to use:** First time setup, or when changing your email.

- `git config --list`

  **What it does:** Shows all your Git configuration settings.

  **When to use:** Checking your current Git configuration.

# 2 Starting a Project

## 2.1 Creating Repositories

- `git init`

  **What it does:** Creates a new Git repository in the current directory.

  **When to use:** Starting a new project from scratch.

- `git clone <url>`

  **What it does:** Downloads an existing repository from GitHub/remote to your computer.

  **When to use:** Getting a copy of someone else's project or your own remote repository.

- `git clone <url> <directory-name>`

  **What it does:** Clones a repository into a specific folder name.

  **When to use:** When you want to rename the cloned folder.

# 3 Daily Workflow - Making Changes

## 3.1 Checking Status

- `git status`

  **What it does:** Shows which files are modified, staged, or untracked.

  **When to use:** Before committing to see what changes you've made.

- `git diff`

  **What it does:** Shows exact line-by-line changes in your files.

  **When to use:** Reviewing what you changed before staging.

- `git diff --staged`

  **What it does:** Shows changes that are staged for commit.

  **When to use:** Reviewing staged changes before committing.

### 3.2 Staging Changes

- `git add <filename>`

  **What it does:** Stages a specific file for commit.

  **When to use:** When you want to commit only specific files.

- `git add .`

  **What it does:** Stages all modified and new files in current directory.

  **When to use:** When you want to stage all your changes at once.

- `git add -A`

  **What it does:** Stages all changes (new, modified, deleted) in entire repository.

  **When to use:** When you want everything staged, including deletions.

- `git add *.js`

  **What it does:** Stages all JavaScript files.

  **When to use:** When staging files by extension.

- `git reset <filename>`

  **What it does:** Unstages a file (opposite of git add).

  **When to use:** When you accidentally staged a file.

### 3.3 Committing Changes

- `git commit -m "message"`

  **What it does:** Saves your staged changes with a description.

  **When to use:** After staging files you want to commit.

- `git commit -am "message"`

  **What it does:** Stages and commits all modified files in one step.

  **When to use:** Quick commits of tracked file changes (doesn't add new files).

- `git commit --amend`

  **What it does:** Modifies the last commit (change message or add files).

  **When to use:** Fixing a typo in commit message or adding forgotten files.

## 4 Branching - Parallel Development

### 4.1 Branch Management

- `git branch`

  **What it does:** Lists all local branches (* shows current branch).

  **When to use:** Checking which branches exist and which you're on.

- `git branch <branch-name>`

  **What it does:** Creates a new branch.

  **When to use:** Starting work on a new feature or experiment.

- `git branch -d <branch-name>`
  **What it does:** Deletes a branch (safe - warns if unmerged).
  **When to use:** Cleaning up after merging a feature branch.
- `git branch -D <branch-name>`
  **What it does:** Force deletes a branch (even if unmerged).
  **When to use:** Discarding a branch with unwanted changes.
- `git branch -m <new-name>`
  **What it does:** Renames the current branch.
  **When to use:** Fixing a branch name typo.

### 4.2 Switching Branches

- `git checkout <branch-name>`
  **What it does:** Switches to an existing branch.
  **When to use:** Moving between different features or versions.
- `git checkout -b <branch-name>`
  **What it does:** Creates a new branch and switches to it.
  **When to use:** Starting a new feature in one command.
- `git switch <branch-name>`
  **What it does:** Switches to an existing branch (newer alternative).
  **When to use:** Same as checkout but clearer intent.
- `git switch -c <branch-name>`
  **What it does:** Creates and switches to new branch.
  **When to use:** Same as checkout -b but newer syntax.

## 5 Merging - Combining Changes

### 5.1 Merge Operations

- `git merge <branch-name>`
  **What it does:** Combines another branch into your current branch.
  **When to use:** Incorporating feature branch into main branch.
- `git merge --no-ff <branch-name>`
  **What it does:** Creates a merge commit even if fast-forward is possible.
  **When to use:** Keeping history of branch merges explicit.
- `git merge --abort`
  **What it does:** Cancels a merge in progress (during conflicts).
  **When to use:** When merge conflicts are too complex.

# 6 Remote Repositories - GitHub Sync

## 6.1 Remote Management

- `git remote`

  **What it does:** Lists all remote connections.

  **When to use:** Checking which remotes are configured.

- `git remote -v`

  **What it does:** Shows remote URLs for fetch and push.

  **When to use:** Verifying remote repository URLs.

- `git remote add origin <url>`

  **What it does:** Connects your local repo to GitHub repository.

  **When to use:** First time pushing a local repo to GitHub.

- `git remote remove <name>`

  **What it does:** Removes a remote connection.

  **When to use:** Disconnecting from an old repository.

- `git remote rename <old> <new>`

  **What it does:** Renames a remote connection.

  **When to use:** Changing remote name from origin to something else.

## 6.2 Fetching Changes

- `git fetch`

  **What it does:** Downloads remote changes without merging them.

  **When to use:** Checking what others changed without affecting your work.

- `git fetch origin`

  **What it does:** Fetches all branches from origin remote.

  **When to use:** Getting latest remote state before pull or merge.

## 6.3 Pulling Changes

- `git pull`

  **What it does:** Fetches and merges remote changes into current branch.

  **When to use:** Updating your local branch with remote changes.

- `git pull origin <branch>`

  **What it does:** Pulls changes from specific remote branch.

  **When to use:** Getting updates from a specific branch.

- `git pull --rebase`

  **What it does:** Applies your commits on top of remote commits.

  **When to use:** Keeping a linear commit history.

### 6.4 Pushing Changes

- `git push`

  **What it does:** Uploads your commits to remote repository.

  **When to use:** Sharing your committed changes with team.

- `git push origin <branch>`

  **What it does:** Pushes specific branch to remote.

  **When to use:** First time pushing a new branch.

- `git push -u origin <branch>`

  **What it does:** Pushes and sets upstream tracking.

  **When to use:** First push of new branch (enables simple git push later).

- `git push --force`

  **What it does:** Overwrites remote branch with local (dangerous).

  **When to use:** After rewriting history (use with extreme caution).

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

  **What it does:** Safer force push that checks remote hasn't changed.

  **When to use:** Force pushing after rebase (safer than –force).

## 7 History - Viewing Past Changes

### 7.1 Log Commands

- `git log`

  **What it does:** Shows commit history.

  **When to use:** Reviewing what was done in the project.

- `git log --oneline`

  **What it does:** Shows condensed commit history (one line per commit).

  **When to use:** Quick overview of recent commits.

- `git log --graph`

  **What it does:** Shows commit history with branch visualization.

  **When to use:** Understanding branch and merge history.

- `git log --all --graph --oneline`

  **What it does:** Visual commit history of all branches.

  **When to use:** Seeing complete project history graphically.

- `git show <commit-hash>`

  **What it does:** Shows details of a specific commit.

  **When to use:** Examining what changed in a particular commit.

# 8 Undoing Changes - Fixing Mistakes

## 8.1 Unstaging and Reverting

- `git restore <filename>`

  **What it does:** Discards changes in working directory.

  **When to use:** Reverting a file to last committed state.

- `git restore --staged <filename>`

  **What it does:** Unstages a file (keeps changes in file).

  **When to use:** Removing file from staging area.

- `git checkout -- <filename>`

  **What it does:** Discards changes (older alternative to restore).

  **When to use:** Same as restore (older syntax).

## 8.2 Reset Commands

- `git reset HEAD~1`

  **What it does:** Undoes last commit, keeps changes staged.

  **When to use:** Redoing your last commit differently.

- `git reset --soft HEAD~1`

  **What it does:** Undoes last commit, keeps changes staged.

  **When to use:** Combining multiple commits into one.

- `git reset --mixed HEAD~1`

  **What it does:** Undoes last commit, unstages changes.

  **When to use:** Undoing commit and re-staging selectively.

- `git reset --hard HEAD~1`

  **What it does:** Undoes last commit, deletes all changes (dangerous).

  **When to use:** Completely discarding last commit.

- `git reset --hard origin/<branch>`

  **What it does:** Resets local branch to match remote exactly.

  **When to use:** Discarding local changes to match remote.

## 8.3 Revert Commands

- `git revert <commit-hash>`

  **What it does:** Creates new commit that undoes a previous commit.

  **When to use:** Safely undoing published commits.

# 9 Stashing - Temporary Storage

## 9.1 Stash Operations

- `git stash`

  **What it does:** Temporarily stores uncommitted changes.

  **When to use:** Switching branches with unsaved work.

- `git stash save "message"`

  **What it does:** Stashes changes with descriptive message.

  **When to use:** Organizing multiple stashes.

- `git stash list`

  **What it does:** Shows all stashed changes.

  **When to use:** Checking what you've stashed.

- `git stash pop`

  **What it does:** Applies most recent stash and removes it from stash list.

  **When to use:** Restoring stashed work.

- `git stash apply`

  **What it does:** Applies stash but keeps it in stash list.

  **When to use:** Applying stash to multiple branches.

- `git stash drop`

  **What it does:** Deletes most recent stash.

  **When to use:** Removing unneeded stash.

- `git stash clear`

  **What it does:** Deletes all stashes.

  **When to use:** Cleaning up all stashed changes.

# 10 Tagging - Marking Versions

## 10.1 Tag Management

- `git tag`

  **What it does:** Lists all tags.

  **When to use:** Viewing release versions.

- `git tag <tag-name>`

  **What it does:** Creates lightweight tag at current commit.

  **When to use:** Marking a specific point in history.

- `git tag -a <tag-name> -m "message"`

  **What it does:** Creates annotated tag with message.

  **When to use:** Creating release versions with descriptions.

- `git push origin <tag-name>`
  **What it does:** Pushes specific tag to remote.
  **When to use:** Sharing a tagged release.
- `git push origin --tags`
  **What it does:** Pushes all tags to remote.
  **When to use:** Uploading multiple release tags.
- `git tag -d <tag-name>`
  **What it does:** Deletes local tag.
  **When to use:** Removing incorrect tag.

# 11 Advanced Operations

## 11.1 Rebasing

- `git rebase <branch>`
  **What it does:** Moves your commits to tip of another branch.
  **When to use:** Keeping linear history, updating feature branch.
- `git rebase -i HEAD~n`
  **What it does:** Interactively edit last n commits.
  **When to use:** Squashing, reordering, or editing commits.
- `git rebase --continue`
  **What it does:** Continues rebase after resolving conflicts.
  **When to use:** Resuming rebase after fixing conflicts.
- `git rebase --abort`
  **What it does:** Cancels rebase and returns to original state.
  **When to use:** Abandoning problematic rebase.

## 11.2 Cherry-picking

- `git cherry-pick <commit-hash>`
  **What it does:** Applies specific commit to current branch.
  **When to use:** Copying a bug fix from one branch to another.

## 11.3 Clean Operations

- `git clean -n`
  **What it does:** Shows which untracked files would be removed.
  **When to use:** Previewing cleanup.
- `git clean -f`
  **What it does:** Removes untracked files.
  **When to use:** Cleaning up generated files.

- `git clean -fd`

  **What it does:** Removes untracked files and directories.

  **When to use:** Deep cleaning working directory.

# 12 GitHub-Specific Workflows

## 12.1 Forking Workflow

1. Fork repository on GitHub website
2. `git clone <your-fork-url>`
3. `git remote add upstream <original-repo-url>`
4. Make changes and commit
5. `git push origin <branch>`
6. Create Pull Request on GitHub

## 12.2 Syncing Fork

- `git fetch upstream`

  **What it does:** Gets changes from original repository.

  **When to use:** Updating your fork with upstream changes.

- `git merge upstream/main`

  **What it does:** Merges upstream changes into current branch.

  **When to use:** Updating local branch with upstream.

# 13 Useful Shortcuts & Aliases

## 13.1 Common Aliases to Set Up

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual 'log --all --graph --oneline'
```

# 14 Common Workflows

## 14.1 Daily Development Flow

1. `git pull` - Get latest changes
2. Make your changes in files
3. `git status` - Check what changed
4. `git add .` - Stage changes

5. `git commit -m "message"` - Commit changes
6. `git push` - Push to remote

### 14.2 Feature Branch Flow

1. `git checkout -b feature-name` - Create feature branch
2. Make changes and commit
3. `git push -u origin feature-name` - Push feature branch
4. Create Pull Request on GitHub
5. After merge: `git checkout main`
6. `git pull` - Update main
7. `git branch -d feature-name` - Delete local feature branch

### 14.3 Fixing Merge Conflicts

1. `git merge <branch>` or `git pull` shows conflict
2. Open conflicted files and edit
3. `git add <resolved-files>` - Mark as resolved
4. `git commit` - Complete the merge
5. `git push` - Push resolved merge

## 15 .gitignore Patterns

Common patterns to add to your .gitignore file:

```
# Node modules
node_modules/

# Environment variables
.env
.env.local

# IDE files
.vscode/
.idea/

# OS files
.DS_Store
Thumbs.db

# Build output
dist/
build/
*.log

# Dependencies
vendor/
```

# 16 Best Practices

- **Commit often:** Small, frequent commits are better than large ones
- **Write clear commit messages:** Describe what and why, not how
- **Pull before push:** Always get latest changes first
- **Don't commit sensitive data:** Use .gitignore for credentials
- **Use branches:** Keep main/master stable
- **Review before commit:** Use git diff to check changes
- **Never force push to shared branches:** Unless absolutely necessary

# 17 Emergency Commands

## 17.1 Oh No, I Messed Up!

- **I committed to wrong branch:**
  1. `git log` - Copy commit hash
  2. `git checkout correct-branch`
  3. `git cherry-pick <hash>`
  4. `git checkout wrong-branch`
  5. `git reset --hard HEAD~1`
- **I need my last commit back:**
  `git reflog` - Find commit hash, then `git checkout <hash>`
- **I want to undo everything:**
  `git reset --hard origin/<branch>`