

Git Commands

Running config creates the appropriate config file:

Project configs are only available for the current project and stored in .git/config in the project's directory.

Global configs are available for all projects for the current user and stored in ~/.gitconfig.

System configs are available for all the users/projects and stored in /etc/gitconfig.

Use --global for global config, --system for system config, and neither for project config (run inside project directory).

\$ git config --global user.name "Your Name" --global user.email "you@example.com"

\$ git config --global core.excludesfile [file] System-wide ignore pattern for all local repos (like .gitignore)

\$ git config --global --edit Edit config file in editor

\$ git init [project name] If no name given, create a new repo in the current directory.

\$ git add [file] Add a file to the staging area.

\$ git rm [file] Remove file from working directory and staging area.

\$ git clone ssh://user@domain.com/repo.git <OR> ~/existing/repo ~/new/repo

\$ git commit -m [message] (-a for all) Create a new commit from changes added to the staging area

\$ git fetch [remote] Fetch changes from remote, don't merge into HEAD or tracking branches.

\$ git fetch --prune [remote] Delete remote refs that were removed from the remote repo.

\$ git pull [remote] [branch] Get changes from the remote and merge current branch with its upstream.

\$ git pull --rebase <remote> Same as above, but uses git rebase instead of merge

\$ git merge [branch_name] Join specified remote branch into your current HEAD

\$ git branch [branch_name] Create new branch, referencing the current HEAD.

\$ git branch -d [branch_name] Remove branch, if it is already merged into any other. -D forces

\$ git checkout [-b][branch_name] Switch working dir to branch; -b: create branch if it does not exist.

\$ git checkout -b \$new_branch \$other Create \$new_branch based on branch \$other and switch to it

\$ git checkout --track [remote/branch] Create new tracking branch based on a remote branch

\$ git push [remote] [branch] Publish local commits to a remote; --tags to push tags; -u as an upstream

\$ git push --all (means all branches), --tags (means all tags, aren't normally pushed) --force (careful!)

\$ git checkout \$id \$file Checkout the \$id version of a file

\$ git diff [file] Show changes between working directory and staging area.

\$ git diff --staged [file] Diff of what is staged but not yet committed

\$ git diff HEAD Show difference between working directory and last commit.

\$ git diff --cached Show difference between staged changes and last commit

\$ git log --follow [file] how the commits that changed file, even across renames

\$ git diff \$id1 \$id2 What changed between \$ID1 and \$ID2

\$ git checkout [file] Discard local changes in a specific file (replace with remote commit)

\$ git revert [commit] Create new commit that undoes all of the changes made in [commit]

\$ git revert HEAD Revert the last commit

\$ git revert \$id Revert specific commit

\$ git revert [commit sha] Create a new commit, reverting changes from the specified commit.

\$ git reset [file] Unstage a file while retaining the changes in working dir

\$ git reset [commit, file, tag] Reset HEAD pointer to previous commit, preserve all changes as unstaged

\$ git reset --hard [commit] Reset HEAD pointer to previous commit & discard all changes since

\$ git reset --keep [commit] If difference between <commit> and HEAD, has local changes, reset aborted.

\$ git clean -n Shows files that clean would remove. The -f flag executes the clean.

\$ git stash Put current changes in your working directory into stash for later use.

\$ git stash list List stack-order of stashed file changes

\$ git stash pop Apply stored stash content into working directory, and clear stash.

\$ git stash drop Delete a specific stash from all your previous stashes.

\$ git tag -a [name] [commit sha] Create a tag object named name for current commit. (use -d to remove tag)

\$ git tag v1.0 Mark a version / milestone

\$ git tag List all tags.

\$ git remote add [remote] [url] Add new remote repository, named [remote]

\$ git branch -dr [remote/branch] Delete a branch on the remote

\$ git rebase [branch] Apply commits in branch ahead of remote branch. Don't rebase published commits!

\$ git rebase --abort Abort a rebase <AND> git rebase --continue Continue a rebase

\$ git rebase -i Interactively rebase current branch onto <base> specify how to handle each commit

\$ git mergetool Use your configured merge tool to solve conflicts

\$ git commit --amend Replace the last commit with combined staged changes and last commit. Use with nothing staged to edit the last commit's message. Don't amend published commits.

\$ git remote -v	List all currently configured remotes
\$ git remote show [remote]	Show information about a remote
\$ git branch [-a]	List branches. A * notes the currently active branch; -a: show all incl. remote).
\$ git show [SHA]	Show any object in Git in human-readable format
\$ git show \$id:\$file	A specific file from a specific \$ID
\$ git blame \$file	Who changed what and when in a file
\$ git log [-n count]	List commit history of current branch
\$ git log --oneline --graph --decorate	Overview with reference labels and history graph
\$ git log refA..refB	Show commits on between branchA and branchB (ref can be branch, tag, etc)
\$ git log --follow [file]	Show the commits that changed file, even across renames
\$ git log --author= "<pattern>"	Search for commits by a particular author.
\$ git log --grep= "<pattern>"	Search for commits with a commit message that matches
\$ git log -p \$file \$dir/ec/tory/	History of changes for file with diffs
\$ git log --stat -M	Show all commit logs with indication of any paths that moved
\$ git reflog	List operations (e.g. checkouts or commits) made on local repo. --relative-date to show date info, --all to show all refs

To view the merge conflicts

\$ git diff --base \$file	(against base file)
\$ git diff --ours \$file	(against your changes)
\$ git diff --theirs \$file	(against other changes)

To discard conflicting patch

\$ git reset --hard
\$ git rebase --skip

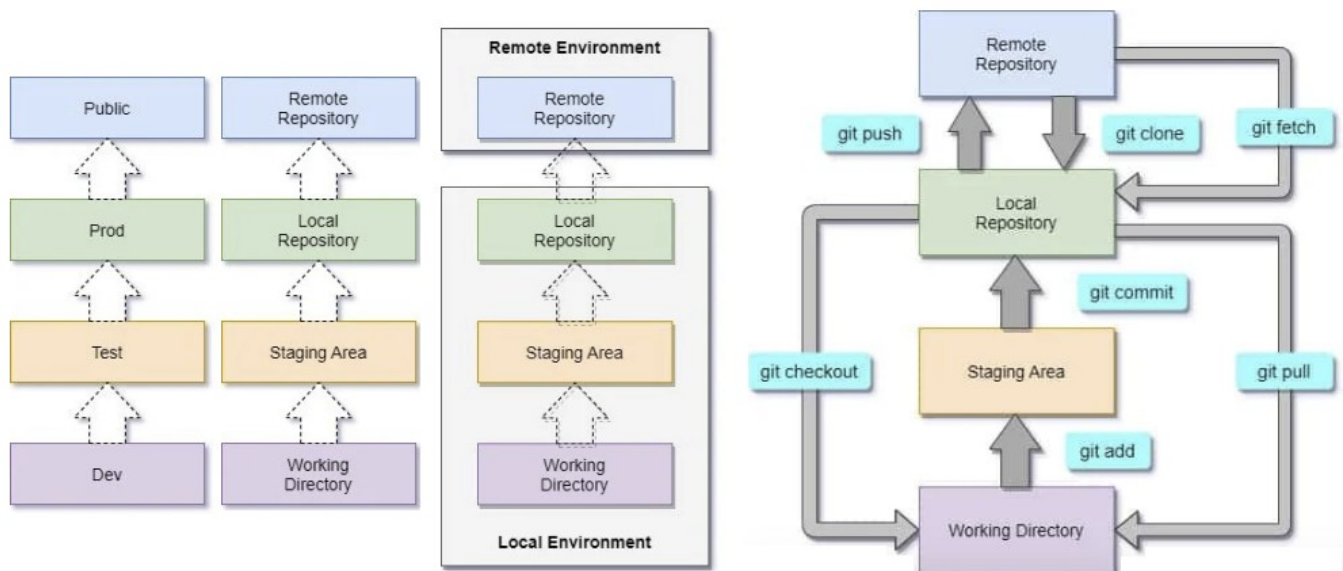
After resolving conflicts, merge with:

\$ git add \$conflicting_file Do for all resolved files
\$ git rebase --continue

\$ git fsck	Check for errors
\$ git gc --prune	Cleanup repository
\$ git grep "foo()"	Search working directory for foo()

Finding regressions

\$ git bisect start	To start
\$ git bisect good \$id	\$id is the last working version
\$ git bisect bad \$id	\$id is a broken version
\$ git bisect bad good	To mark it as bad or good
\$ git bisect visualize	Once you're done
\$ git bisect reset	To launch gitk and mark it



MASTER = default development branch; ORIGIN = default upstream repository; HEAD = current branch; HEAD^ = parent of HEAD; HEAD~4 = the great-great grandparent of HEAD

Git Aliases

Git Please

```
$ git config --global alias.please 'push --force-with-lease'
```

Team leads warn about not force pushing to a shared branch. Rebasing, amending, and squashing can rewrite some shared history and spill duplicate commits all over the repo. Force stomps the upstream branch with your local version, and any changes that you hadn't already fetched are erased from history. Git's `--force-with-lease` checks that your local copy of the ref that you're overwriting is up-to-date first; that you've at least fetched the changes you're about to stomp. Here you only have to type "git please"

Git Commend

```
$ git config --global alias.commend 'commit --amend --no-edit'
```

Commit and then realize you'd forgotten to stage a file? Quietly amend any staged files onto the last commit you created, re-using your existing commit message. So as long as you haven't pushed yet, no-one will be the wiser. Don't amend published commits.

Git It

```
$ git config --global alias.it '!git init && git commit -m "root" --allow-empty'
```

The first commit of a repo can not be rebased like regular commits, so it's good practice to create an empty commit as your repo root. "git it" both initializes and creates an empty root commit in one quick step.

```
$ cd shiny-new-thing
$ git it
Initialized empty Git repo in /shiny-new-thing/.git/
```

Git Staaash

```
$ git config --global alias.stsh 'stash --keep-index'
$ git config --global alias.staash 'stash --include-untracked'
$ git config --global alias.staaash 'stash --all'
```

Takes any changes to tracked files in your work tree and stashes them away for later use, leaving you with a clean work tree to start hacking on something else. However if you've created any new files and haven't yet staged them, git stash won't touch them by default, leaving you with a dirty work tree, use staash. Similarly, the contents of untracked or ignored files are not stashed by default; use staaash. If in doubt, the long one (git staaash) will always restore your worktree to what looks like a fresh clone of your repo.

```
git stsh      # stash only unstaged changes to tracked files
git stash     # stash any changes to tracked files
git staash    # stash untracked and tracked files
git staaash   # stash ignored, untracked, and tracked files
```

Git Shortstat

```
$ git config --global alias.shortstat 'status --short --branch'
```

Git status inline help has improved, but the output is overly verbose for those more familiar with Git. For example, git status emits 18 lines to tell me that I have a couple of staged, unstaged, and untracked changes. Git shortstat tells me the same thing in three lines:

```
$ git shortstat
## master
AM test
?? .gitignore
```

Git Merc

```
$ git config --global alias.merc 'merge --no-ff'
```

On non-rebasing branching workflows running a git merge to combine feature branches with the master is not ideal. With no options, git merge uses the `--ff` merge, which creates a merge commit only if no new changes are on the master branch, otherwise it "fast forwards" your master branch to point at the latest commit on your feature branch. Without a merge commit it's tricky to tell which code was developed on which branches in the git history. The `--no-ff` strategy, to always create a merge commit.

Git Grog (or "graphical log")

```
$ git config --global alias.grog 'log --graph --abbrev-commit --decorate --all --format=format:"%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(dim white) - %an%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n %C(white)%s%C(reset)'"
```