

Git It Right: A Complete Cheatsheet

Source Control

Source control is the practice of tracking and managing changes to code. There are two types:

1. In **centralized source control**, a centralized server acts as the ultimate source of truth for a collection of versioned files.
 - *Implications.* An internet connection to the central server is required for most basic operations.
 - *Examples.* **Subversion**, **CVS**
2. **Distributed or decentralized source control** doesn't require a central source of truth and allows for most operations to be local.
 - *Implications.* You can work independently of an internet connection.
 - *Examples.* **Git**, **Mercurial (Hg)**

History of Git: **Git** was developed by **Linus Torvalds**, the creator of **Linux**, to handle the requirements of the **Linux Kernel Project**. It is often used, because...

1. Due to its distributed nature, **Git** can **scale massively**.
2. **Git** is **very fast** to execute, since most of its operations are local.
3. Due to its history, it has a **very active community**.

Git Theory

A **repository** is a collection of version controlled files that are kept together. This includes (a) all the files related to a specific project/application, (b) the history of changes, and (c) any special configurations.

Git states: **Git** has three **local** states.

1. The **working directory state** holds all the project or application files. These files may or may not be managed by **Git**, but **Git** is aware of them.
2. The **staging area state** or **Git index state** is holding area for the queue of changes to be included in your next commit.
3. The **local Git repository state** is a hidden folder called **.git**, which contains your entire local commit history.

Git also has a **remote (repository) state**, which is just another repository with its own three internal states. A specific **Git** command is used to move files between these states. i.e.,

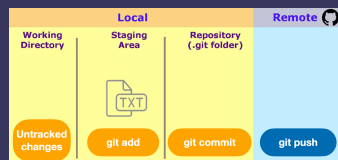


Figure 1: Git states and associated commands. [Source](#)

Git Theory (2)

Tracking: A **tracked file** is any file that **Git** is aware of and is actively tracking. In other words, any files that aren't new.

Git Command	Description
git ls-files	Returns the list of files tracked by Git
git commit -am "commit message"	Simultaneously add and commit a tracked file
git add .	Recursively add untracked files from current filepath location

Quick Start

Installation: Depends on your OS.

- **MacOS.** Normally, it's pre-installed. Otherwise, use command line developer tools to install it.
- **WindowsOS.** Install the open source **Git for Windows Project**. Make sure to execute your **Git** commands from the **Git Bash**.

Configure: You need to tell **Git** who you are and where the remote repository hosting platform is.

Git Command	Description
git config --global user.name "bellanich"	Set your global username
git config --global user.email "youremail@gmail.com"	Set your global email
git config --global --list	Check your global configs

Initialize your project: Here's how to start a new **Git** project.

Git Command	Description
git init my-project-name	Initialize a new empty Git repository
git init	Convert an existing dir into a Git project
git clone git-project-url	Initialize from your code hosting platform of choice
git remote add origin git-project-url	Add a remote reference to your local repository
git push origin main	Force push to main branch (only do for initialization)

Commit History

Introduction: Your **Git commit history** is the chronological record of all commits ever made within your **Git** repository, where each commit is a snapshot of your project at a specific point in time.

- *Implications.* The commit messages you make matter. We recommend writing **conventional commits**.

A project's **Git** history is typically **represented as a directed acyclic graph (DAG)** data structure. Use the idea of a perfect commit to scope your commits.

Commit History (2)

How does **Git** commit history work? **Git** doesn't copying entire files in each commit. Rather, it uses a **system of blobs and pointers**.

- **Git** uses **SHA-1 hashing** to create unique identifiers (hashes) for file content and stores
- The hashes are stored as **blobs** (**binary large objects**) in a database
- **Git** uses pointers to reference the changes made to these blobs in its commit history

Search: Here is how you can search through your **Git** history logs.

Git Command	Description
git log --oneline --graph --decorate	View your entire Git history in a user friendly way
git log --since="3 days ago"	Search for all Git commits made in the last 3 days
git log -- your_filename	Get the history for only a specific file
git log --follow -- your_filename	Include filename changes in your Git history search
git show git_commit_hash	See a given commit's entire content, including: commit message, that commit's git diff results, author, and date

Compare: You can what you have and haven't committed in **Git**. You can also compare different points in your commit history.

Git Command	Description
git diff	Compare staged and unstaged changes in your working directory to your last commit.
git diff -- your_filename	Only preview comparisons for a specific file
git diff --staged HEAD	Review the (staged) changes about to be committed
git diff commit_hash1 commit_hash2	Compare two commits
git diff local_branchname origin/remote_branchname	Compare local and remote branches

Repo restructuring: Here's how to get **Git** to track your repo structuring.

Git Command	Description
git mv current_filepath new_filepath	Rename your file and get Git to track it in one-go
git rm your_filename	Simultaneously delete and stop tracking a file in one-go
git add -A	Stages any file renaming or deletions done via your IDE
git checkout -- deleted_filename	Undo tracked file deletion

Undo Unwanted Changes

Undo unwanted changes: Once you've committed something, it's in Git forever. However, there are some things that can be undone.

Git Command	Description
<code>git reset HEAD your_filename</code>	Undoes an unwanted <code>git add</code> .
<code>git reflog</code>	View records of time travel, i.e. <code>HEAD</code> resets
<code>git checkout -- your_filename</code>	Reverts a file to its last commit version
<code>git commit --amend -m "new commit message"</code>	Update your Git commit message

Note. The `git reflog` holds records for the last 60 days. After that, it autopurges.

Resets: There are 3 different types of resets in Git.

1. **Soft Reset.** Moves the `HEAD` and branch pointer to a different commit, but **leaves changes in the staging area**. Undoes a `git commit` while keeping changes made.
2. **Mixed Reset.** Moves the `HEAD` and branch pointer to a different commit and **clears the staging area**. Default in Git.
3. **Hard Reset.** Same as a mixed reset, except it **also discards all changes in the working dir**. Hence, a destructive operation.

Stashing

Introduction: The `Git stash` command allows you to temporarily save and store changes in your working directory that are not ready to be committed. It stores these changes as a **stack data structure**.

- *When to use?* Whenever you want to save but not commit incomplete work. e.g., switching branches or addressing urgent tasks
- *Best practices.* Save your stash with a specific stash message, especially when working with multiple stashes at once

Git Command	Description
<code>git stash list</code>	Get the list of all stashed changes in your repo
<code>git stash</code>	Stash all tracked changes
<code>git stash -u</code>	Include untracked changes in your stash
<code>git stash show stash@{01}</code>	Shows aggregate file changes for 1st index in Git stash stack
<code>git stash push -m "stash message"</code>	Stash your work with a clear message

Move stashes: Eventually, you'll either **(a)** want to delete your stashed work to declutter the stash stack or **(b)** move your stashed work back into the working directory.

Git Command	Description
<code>git stash apply</code>	Applies most recent stash to working directory without deleting it from stash stack
<code>git stash drop</code>	Manually delete most recent stash from stack

Stashing (2)

Commands to move stashes: (continued)

Git Command	Description
<code>git stash pop</code>	Shorthand combination of <code>git stash apply</code> and <code>git stash drop</code>
<code>git stash apply stash@{1}</code>	Apply a specify specific stash to working dir
<code>git stash drop stash@{1}</code>	Delete a specify specific stash from stack
<code>git stash clear</code>	Deletes all stashes

Stash into another branch: You can use `git stash` to move changes from one branch into another one. Here are the 2 steps:

1. Stash the changes that you want move
2. Use the command `git stash branch new_branch_name` to automatically create a new branch, where your stash is applied. Your stash will also be removed from the stack.

Tagging

Introduction: Git tags are just labels that we can link to a specific commit in our history. There is a 1-to-1 mapping between git tags and commits.

- *Why?* They let us mark significant events or milestones in our repository. e.g., project releases
- *Note.* Tags function like commits. You make them locally and then push to the remote repository.

There are 2 different types of Git tags:

1. **Lightweight tags** are markers on a commit
2. **Annotated tags** have more information than a lightweight tag does. They also contain the tag message, the tag author, and tagging date.

Basic usage: Here are the basic Git tag commands.

Git Command	Description
<code>git tag tag_name</code>	Create a local lightweight tag for most recent commit
<code>git tag tag_name "your tag message"</code>	Create a local annotated tag for most recent commit
<code>git tag tag_name commit_hash "your tag message"</code>	Create a local annotated tag for an older commit
<code>git tag --list</code>	Shows the names of all Git tags within your repo
<code>git show tag_name</code>	Preview the contents of the tagged commit
<code>git tag --delete tag_name</code>	Delete a local tag
<code>git diff tag_name1 tag_name2</code>	Compare the differences between 2 different tagged commits

Syncing tags: When you push a tag to your remote repository, you also push its associated commit.

Tagging (2)

Commands to sync your Git tags:

Git Command	Description
<code>git push origin tag_name</code>	Push a specific Git tag to remote
<code>git push origin remote_branch_name --tags</code>	Push all Git tags to a specific remote branch
<code>git push origin :tag_name</code>	Delete a tag that you've accidentally pushed to remote

Undoing tags: Let's suppose that you put your git tag on the wrong commit. There are 2 ways to fix it:

1. **Delete and recreate** your Git tag on the correct commit.
2. **Force a git tag change** by using the command `git tag -a tag_name -f commit_hash`. This forces the git tag to move from its current commit to a new one.

Branches

Introduction: It's a best practice to create all your changes on feature (or topic) branches, make sure these changes stable, and then merge into the main branch.

Git Command	Description
<code>git branch</code>	Lists all current local branches
<code>git branch --all</code>	Lists all current local and remote branches
<code>git branch -m old_branch_name new_branch_name</code>	Rename your Git branch
<code>git branch --delete branch_name</code>	Deletes a local branch

Merging: Insert the changes from one branch into another as a new commit. Main types of merges are:

- A **fast forward merge** happens when one branch is ahead of another. i.e.,

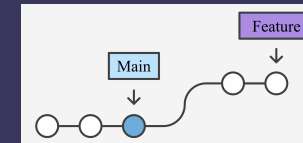


Figure 2: Before fast-forward merge. Source



Figure 3: After fast-forward merge. Source

- A **3-way merge** happens when histories between two branches diverge. This can result in a Git conflict. i.e.,

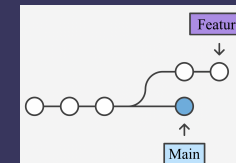


Figure 4: Before 3-way merge. Source

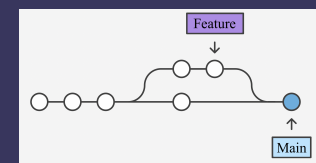


Figure 5: After 3-way merge. Source

Branches (2)

Either way, you can **squash** multiple commits into one as you merge. This ensures that your commit history size remains manageable. Of course, **less common merge types** do exist for very specific situations.

Rebasing: rewrite the commit history by "rewinding" the changes from one branch onto another. This means moving commits around.

- **Why?** This lets you maintain a linear project history when your feature and master branches start to diverge. It also eliminates the unnecessary merge commits required by merging.
- **Pull with a rebase.** You can pull changes from **main** into your feature branch as a rebase. This keeps your Git history flat. Use `git pull --rebase origin master`.

The Golden Rule of Rebasing is do **not** rebase on **public branches**.

- **Why?** Branch histories can diverge. This can only be resolved with a merge, which would pollute the commit history.
- **More info.** Check out [this explanation](#).

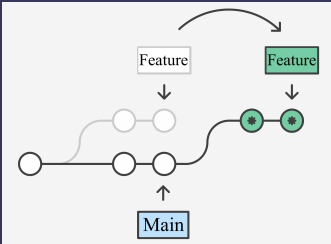


Figure 6: The effects of a git rebase. [Source](#)

Branching strategies: the strategy chosen by a software development teams when writing, merging and deploying code. Here are some noteworthy ones:

1. **GitFlow** allows for parallel development to protect the production code. **Many variants exist**. It has the following types of branches:

Branch	Description
master	The code that's in production
develop	Where developers merge their features into. Branches off of master .
feature	One new feature per branch. Branches off of develop . Merge back once stable.
release	Prepare for a new production release, needs to be merged back into master and develop .
hotfix	Fix a bug that has been discovered and must be resolved (usually from production)

2. In **trunk-based development**, developers merge small, frequent updates to a core "trunk" or main branch. You **push directly** into **master** and use **release** branches.
3. **Ship / Show / Ask** has three categories for merges:

Category	Description
Ship	Make a change directly into your mainline. Great for updated docs, simple bug fixes; etc.

Branches (3)

3. **Ship / Show / Ask** has 3 categories for merges: (continued)

Category	Description
Show	Open a Pull Request with your change but merge without waiting for anyone.
Ask	Open a Pull Request and wait for approval before merging

Aliases

Introduction: Rather than adding Git aliases to a `~/ .bashrc` or `~/ .zshrc` file, you can declare them directly in Git.

Git Command	Description
<code>git config --global alias.alias_name "your git command"</code>	Creates a new Git alias. Do not include git in double quotations, since Git automatically adds this.
<code>git config --global --unset alias.alias_name</code>	Globally delete your Git alias

Your alias definition will be saved in your machine's `~/ .gitconfig` file. You can also edit or delete your aliases from here.

Favorite aliases:

Git Alias	Definition
<code>git history</code>	<pre>git config --global alias.history "log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%ad%C(reset) %C(bold green)(%ar)%C(reset) %C(bold cyan)(committed: %cd)%C(reset) %C(auto)%d%C(reset)%n'' %C(white)%s%C (reset)%n' %C(dim white)- %an <%ae> %C(reset) %C(dim white)(committer: %cn <%ce>)%C(reset)'"</pre>

Git Conventions

Repository name conventions: what to keep in mind when naming projects and referencing remote

- **Project names** are expected to be **(a)** a short and descriptive name, **(b)** in all lowercase letters, and **(c)** only use dashes as separators.
- **Remote repository** is referred to as **origin** in Git.

Commits: There are some shorthands to refer to the most recently made commits.

Git Command	Description
<code>HEAD</code>	Shorthand in Git for the last commit
<code>HEAD~</code>	Shorthand for 2nd to last commit.
<code>HEAD~3</code>	Shorthand for 3 commits before current HEAD.

Default branch name: The Git community is slowly migrating away from the default branch name being **master** and towards it being **main**.

Git Command	Description
<code>git branch -m main</code>	Change the default branch name from master → main for a specific repo
<code>git config --global init.defaultBranch main</code>	Change the default branch across all repositories

Git Conventions (2)

Naming new branches: **One convention for branch naming** is to have a prefix that states the branch's general purpose. i.e.,

Branch Type	Convention
feature	Branch name is either feat/my-description or feature/my-description
bugfix	Branch name is either fix/my-description or bugfix/my-description
release	Branch name is release/version-number
hotfix	Branch name is hotfix/my-description