***Git*** *is* ***distributed version control system.***

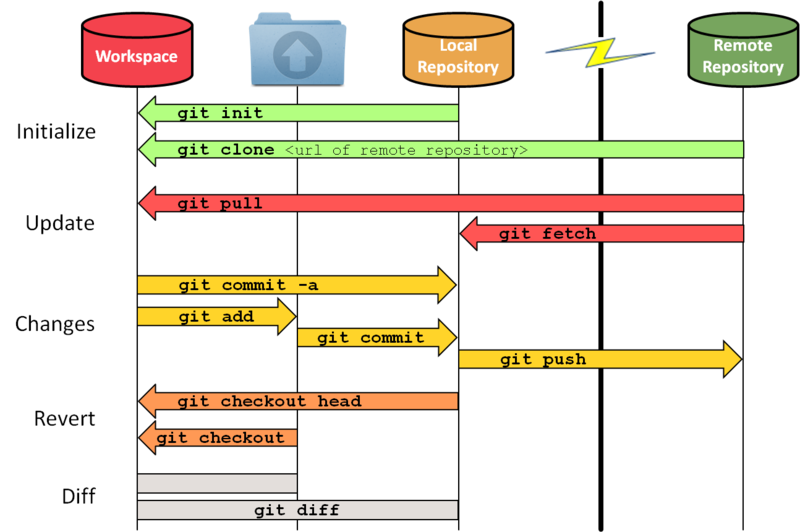
[***GitHub***](https://medium.com/@pamit2897/introduction-to-git-e9d93a4ebd3)*is a***source code hosting service*.***

Created by ***Linus Torvalds for work on the Linux kernel ~ 2005.***

***Version control / revision control / source control*** is all about managing multiple versions of documents, programs, web sites, etc.

* Keep track of changes made to files
* Allows rollbacks
* Merge the contributions of multiple developers
* Facilitates backups
* Increased productivity (vs manual version control)
* Encourages experimentation
* Helps to identify/fix conflicts
* Makes source readily available –Less duplicated effort

Diagram

Description automatically generated with medium confidence 

Git, Mercurial

CVS, Subversion

Tracked files

Untracked files

Staging area

***First step configuration (Only first-time using git in a machine)***

*git config –global user.name “Saranju Bule” [Mandatory]*

*git config –global user.email* [*saranj.bule@gmail.com*](mailto:saranj.bule@gmail.com) *[Mandatory]*

*git config –global core.editor notepad++ [By default vim editor is opened]*

*git config –list (or git config –l) by default global for local insert --local*

|  |  |  |
| --- | --- | --- |
| **Git configuration levels** | |  |
|  | # **local:** (by default) Project configs are only available for the current project and stored in .git/config in the project's directory.  *$ git config user.name "John Doe"* | |
|  | # **global**: Global configs are available for all projects for the current user and stored in ~/.gitconfig.  *$ git config --global user.name "John Doe"* | |
|  | # **system**: System configs are available for all the users/projects and stored in /etc/gitconfig. | |
|  | *$ git config --system user.name "John Doe"* | |

***Second Step to initialize git***

*git init* – to initialize the git

***Third step to add files to staging area***

*Git add <file>* - Stage all changes in <files> for the next commit.

*Git add <directory>* - Stage all changes in <directory> for the next commit.

***Fourth step to commit – log everything to repository***

To commit the changes, we need to provide some message along. - *Git commit -m “”*

***Note*** - if you forget, Git will open a text editor so you can write one

To add and commit with message, use. - *Git commit –am “”* (add + commit)

**To differentiate working version and committed version we can use:**

*Git diff* - Show changes between commits, commit and working tree, etc.

**Color code in git bash**

Red color = represent already committed changes

Green color = represent new changes

|  |  |
| --- | --- |
| git log | git status |
| The Git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. | While Git status lets you inspect the working directory and the staging area |

*git log -n <limit>* = git log -n 3 will display only 3 commits.

*git log –oneline* = Condense each commit to a single line.

*git log --author=“”* = Search for commits by a particular author. The argument can be a plain string or a regular expression.

*git log –all –decorate –online –graph* [shortcut dog] = To crate log history in line graph pattern

**Checkout**

The git checkout command serves three distinct functions: - Checking out files, - Checking out commits, and - Checking out branches.

Checking out a commit makes the entire working directory match that commit.

View an old state without altering your current state in any way.

*git checkout master.*

*git checkout a1e8fb5 – (a1e8fb5 is SHA key for commit).*

**Revert**

The Git revert command undoes a committed snapshot.

When you want to remove an entire commit from your project history.

Git revert undoes a single commit—it does not “revert” back to the previous state of a project by removing all subsequent commits.

Reverting doesn’t change the project history.

Git revert is able to target an individual commit at an arbitrary point in the history

This can be useful, for example, if you’re tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use Git revert to automatically do all of this for you.

**Git revert is a “safe” way to undo changes**

*git revert …*

**Resetting**

**Git reset is the dangerous method.**

There is no way to retrieve the original copy. It is a permanent undo.

It’s one of the only Git commands that has the potential to lose your work.

Git reset can only work backwards from the current commit.

Git reset: Reset the staging area to match the most recent commit but leave the working directory unchanged.

Git reset <file>: Remove the specified file from the staging area but leave the working directory unchanged.

**Rename File**

We can simply use a ‘git mv’ to rename a file where mv stands for move in Linux for renaming.

*git mv filetodelete.txt keepfile.txt.*

**Un-stage**

The way we stage a file to track, the same way we can upstage the file to untrack.

*git reset HEAD filename.txt.*

git push = updating changes on local repository **=>** remote repository

git pull = fetch + merge = bringing changes from remote repository **=>** local repository

git pull (or git fetch + git merge) is how you update that local copy with new commits from the remote repository

git clone is how you get a local copy of an existing repository to work on. It's usually only used once for a given repository, unless you want to have multiple working copies of it around. (Or want to get a clean copy after messing up your local one...)

git fork means you are copying the repository to your Github account.

git remote: used to connect local repository with remote server.

Doesn’t provide real time access to repository.

**SHA - secure hash algorithm**

**>Data Assurance**: Cryptographic integrity of every unit of project. It provides unique commit ID to every commit through SHA algorithm. (40 char long)

> SHA value is same for all the branches because head always points to the branches

***Properties***:

> **Deterministic** (for same input we always get same output, regardless of OS/configuration)

> Output should be fixed length

> **Avalanche effect** ---> for any major/ minor change, entire hash value changes

> **Unique value** --> no 2 distinct inputs have same output

> *git rm --cached <file>* --> remove file from caching/staging/indexing area

> *git restore --staged <file>* ---> Rolling back previous changes inside the file

> *git commit --amend -m "<added chnages>"* --> to update the commit messages

> *git restore –staged* 🡪 Restoring/ Undoing local/ staged changes

**Two types of command**

1. **Porcelain** command (wrapper command)

git status, git branch, git commit, git status, etc..

2. **plumbing** (core commands -> never used commands)

> commit

> tree

> blob --> binary large object (commits stored in these)

gitignore: represent untracked files.

**Pushing code to GitHub**

Create a PAT (Personal Access Token) on GitHub, Path => settings > developer settings > PAT

Then add remote using

git remote add origin <path> (if not added)

git push origin master

Then use username and PAT as password while push operation