GIT

# BEST PRACTICE

* **Atomic Commits** – Keep the scale of each **Commit** as small as possible.
  + Helps with managing work flow in case of complications (one example is data loss).

# GIT COMMANDS

## Git Config:

* **git config --global user.name “**user-name” – creates the username that will be used with git (globally).
* **git config --global user.name** – Displays the username in use with Git, globally.
* **git config user.name** or **git config --local user.name–** Displays the username in use with Git, locally (per repo).
* **git config --global user.email “**user-email” – creates the user email id that will be used with the git updates, globally.
* **git config --global user.email** – Displays the user email in use with Git, globally.
* **git config user.email** or **git config --local user.email** – Displays the user email in use with Git, locally (per repo).
* **git config --global color.ui auto** – Git command to enable some colorization of Git output.
* **git status** - to check the status of the git repository (returns fatal error, when not in a git repo).

## Getting & Creating Projects:

* **git init** - to initialize a git repository.
* **git clone <URL>** - Creates a new directory at the PWD where it creates the Git repo (already initialized).

For **git init** and **git clone <URL>**, make sure that you are not already in a repository.

Once a repo is cloned, it knows about the cloud repo (remote) and is automatically connected to that repo and doesn’t have to be manually reconnected (no remote setup needed).

* The .gitignore file should be present (created) at the root of the Working Tree. Its purpose is to track and make Git ignore (not track) the files specified in it.

## Working on Existing Repo

* **git add .** - adds all files in the Working Tree to the **Staging Area**.
* **git add <file1> <file2>** ... <filen> - A way to add file(s) by specifying filenames (space separated), against adding all files at one go (using the previous command).
* **git rm --cached <file>** - To remove (unstage) file from the Staging area and to bringing it/them (plural) back to the **Working Tree**.
* **git rm --cached .** – To unstage all files those are staged currently.
* **git rm --cached <file1> <file2> <filen>** - Unstaging multiple files by specifying them.
* **git commit** - triggers the default git editor with the commit message window active (*Useful for a descriptive commit or multiline commit*).
* **git commit -m** "This message will be committed as a one-line description only."
* **git commit -a -m "The message** (one line)**."** - adds and commits in 1 command (**-a** option for adding).
* **git commit -am “Message goes here.”**  – Adds files added at least once with **git add** command (previously in their existence and not this time) to the Staging Area from the Working Area and Commits them along with the message in quotes.
* **git commit --amend** – Opens the default editor at the latest commit. Files in this commit and the Commit messages can be amended.
* **git log** - displays currently active branch’s commit logs.
* This consists of commit hash, Author’s Name (email in parentheses), Date, Time and Time zone difference from Greenwich Mean Time.
* **git log --oneline** - displays one-line log messages (just displays the first line) for each commit logged in the branch.

## Branching

* **git branch** - displays the branches that are available in a repo. The star/asterisk (\*) sign signifies HEAD position.
* **git branch -v –** (Option **-v** means verbose) Displaysall branches along with their respective reference (a short 7-character wide Commit hash of respective branch’s reference, along with the respective Commit Message to that reference).
* **git branch branch-name** - creates a branch with given name (use **kebab-case** naming convention).
* **git switch branch-name** / **git checkout branch-name** - switches to the mentioned branch’s Branch Reference. [But, **git checkout** has *several more functionalities*].
* **git switch -c branch-name** **/ git checkout -b branch-name** - creates a new branch at the current HEAD location and switches to the newly created branch (The newly created branch is also sharing the same Commit Hash, and currently **HEAD** is pointing here. But the active branch is the new branch).
* **git branch -d branch-name** - deletes the branch name specified. [Currently active branch and the branch to be deleted should not be the same].
* **git branch -D branch-name** - *force deletes* the branch!
* **git branch -m branch-name** – renames the *active branch*.
* **git branch -M branch-name** – Forcefully renames the *active branch*.

## Merging

* **git merge branch-name –** Combines the specified branch’s history into the Active branch.
* Conflict Type 1 - In **Fast Forward Merge** cases merging happens automatically without any conflict.
* **git merge --no-ff --no-commit <branch>** - **--no-ff** - Prevents fast forward merging**.**

**--no-commit** – Prevents automatic commits, giving opportunity to edit the commit message.

* **git merge --edit <branch>** or **git merge -e <branch>** - Invokes an editor before committing, thus allowing entry of custom Commit Message in place of the auto generated one.
* Conflict Type 2 - **Merge Commit** – If, *ort strategy* is used. It just commits, if there is no conflict.
* Conflict Type 3 - **Merge Conflict –**
* **Delete markers** in the editor. **Delete duplicates** and unnecessary data. **Close editor**.
* This **adds changes** and makes a **commit**.

## Comparing Changes

* It’s all about comparing differences due to changes (Commits) in the Git repo. Difference comparison can be performed between *individual Commits*, *Branches*, *Files*, the Working Tree*, the Staging Area etc.*
* **git diff** – (option needed) shows the differences between Commits, commit and *Working Tree*. (Staging is referred to as Index). [According to the book **Pro Git**, the Staging Area is just a list, or index, that says which files will be committed when a git commit is done. Now the name **Index** is more commonly known as the "**Staging Area**”.]
* **git diff HEAD** – Shows changes between HEAD and both the Working Tree (un-staged) and Staging Area (Index).
* **git diff HEAD [filename or path]** – Similar to git diff HEAD but only displays changes in the mentioned filename. [multiple file names or path separated by space can be given to view changes across the specified files.]
* **git diff --staged** / **git diff --cached –** Only shows changes between the **Staging Area (Index)** and **HEAD**.
* **git diff --staged [filename or path]** – Similar to git diff --staged but only displays changes in the mentioned files. [multiple file names or path separated by space can be given to view changes across the multiple files specified.]
* **git diff branch1..branch2** or **git diff branch1 branch2** – Displays changes between the two branches (order matters).
* **git diff commit1..commit2 or git diff commit1 commit2 -** Displays changes between the two commits (order matters).
* **HEAD** – parent. **HEAD~1 …. HEAD~n**– parent of HEAD. And so on.

## Saving Work for Later

* ***Stashing***: Uncommitted changes in the Working Tree cause no issue as long as there is no conflict with Active branch. In case of probable issue with the branch to be switched; Git will warn us and won’t allow switching.
* **git stash** (or) **git stash save** – Stashes the changes in the Working Tree and in the Staging Area (Index) to the Stash list and allows their retrieval at a later time. The changes can be reapplied to the same branch or a different branch.
* **git stash pop** – Removes the item from the Stash List to the Staging Area. [In presence of multiple stashes, ***git stash*** *pop* pops out the oldest stash].
* **git stash apply** – This can be used to apply the stashed changes to the Working Tree without removing them from the Stash List.

This in turn allows for reuse of the stashed changes to be applied across multiple branches (this is in contrast to the use it and lose it approach of ***git stash pop****)*.

* Multiple stashes: *Until a stashed item is popped, it is retained in the Stash List*.

**git stash list** – Lists the stash entries that you currently have. Each stash entry is listed with its name (e.g. stash@{0} is the latest entry, stash@{1} is the one before, etc.), the name of the branch that was current when the entry was made, and a short description of the commit the entry was based on.

**git stash apply stash@{n}** – Applies the stashed items at the mentioned stash name (number used in Git).

* Dropping Stashes:
  + **git stash drop stash@{n}** – drops the stash with the specified stash-name from the various stashes among the Stash List.
  + **git stash clear** – To drop all the stashed items from the Stash List.

## Working At Different Screenshot

* **git checkout** is a versatile command and it in combination with other options performs several functions.
* **git checkout <commit-hash>** [full hash value or short hash (7 characters wide)]– You are in 'detached HEAD' state (HEAD usually refers to a branch reference (a branch reference usually points to the latest commit in that branch). But in a detached head state, the HEAD points to the specified commit). You can look around, make experimental changes and commit them and exit Detached Head state by switch back to a branch.

If you want to create a new branch to retain commits you create, you may do so.

* Or undo this operation with: **git switch –** - The HEAD goes back to its old position (i.e., position before the ***git checkout <commit-hash>*** *command was executed*).
  + **git switch <branch-name>** – git HEAD goes to the branch specified.
* **git checkout HEAD~1** – means checkout 1 commit before the current commit (parent of the current (from where the command was executed) commit).
* **git checkout HEAD~2** – means checkout 2 commits before the current commit (grand-parent). And so on.

## Undoing changes:

* **Restoring Change**
  + **git checkout HEAD <file>** or **git checkout HEAD <file1> <file2>…<filen>** – Git forgets the changes in the specified file(s) ((space separated, in case of plural files) that are @ the Working Tree) and makes them identical to the respective files at the commit-hash HEAD is currently pointing to.
    - **git checkout -- <file1> <file2>…<file n>** – Does the same as above.
    - **git checkout HEAD .** or **git checkout -- .** – All files that are present in the Working Tree are compared with the respective files at the commit-hash HEAD is currently pointing to and are made identical.
  + **Git Restore** isa new git command that does the same as above! It restores the specified file or files so that the respective specified contents in the Working Tree match the contents in the head.
    - **git restore <file> <file n>** or **git restore .** – Restores specified files @ the Working Tree by making them devoid of changes that differentiates them from the respective files based on the commit-hash HEAD is referring to.
    - **git restore --source <commit-hash> <file>** - Restores the specified file @ the Working Tree based on the file @ the commit hash.
* **Un-staging changes**
  + **git restore --staged <file>… -** This removes the specified file(s) from the staging area leaving the rest intact if all had not been mentioned.
  + **git restore --staged .** – Unstages all the staged files and places them @ the Working Tree.
* **Git RESET** –
  + **git reset <commit-hash> -** Git removes the commits created after the specified commit hash. But, retains the changes (deleted commit files) @ the Working Tree. User has the option to create a new branch or, not (user choice).
  + **git reset --hard <commit-hash>** - Git removes the commits created after the specified commit hash and the changes are also removed from the Working Tree.
* **git revert <commit-hash>** - Achieves undoing changes at the specified commit by creating a new commit (revert(ed) commit) and allows making the necessary changes at that commit (This is good for collaboration as the commit history is not lost).

Git preservers history of the reverted commit but removes the unwanted commit files. Also opens up the default git editor to enter a new git revert-commit-message (a default message is already provided and it can be edited). [Reverts the changes @ the specified commit and creates a new commit with the reverted changes. The old commit (specified commit) still exists in order to preserve collaboration history].

## Looking Into Files

* **cat <file>** or **cat path/<file>** - Displays the contents of the file (either located in the PWD or at the path specified).

**cat <file in .git either HEAD or branch>** – Shows the contents of the specified file. HEAD contains the commit-hash it currently points to. But branch contains the latest screenshot of that branch (branch reference).

**cat .git/<HEAD>** or **git path/<file>** – reveals the contents of HEAD (the hash value of the git screenshot HEAD is currently pointing to).

## Remote Repository Commands:

* **git remote** – Displays list of remote(s) associated with the local repository.
  + **git remote -v** (verbose for more info) – Displays remote name and the associated URL.
  + **git remote add <name-label> <URL>** - A remote is really two things: a URL and a label (user specified). To add a new remote, these have to be provided to Git (Origin is the remote name-label that is widely used).
  + **git remote rename <old> <new>** - renames the remote label.
  + **git remote remove <name> -** removes the remote.
* **git push <remote> <local-branch>** - Pushes the specified local branch to the remote repo’s branch that has an identical to the local repo’s branch. If no such branch exists then a repo branch of that name is created.

One doesn’t have to be on the respective branch while pushing to remote repo. And while *pushing*, an entire branch is pushed, not just one commit.

* + - **git push --set-upstream <remote> <local-branch>** or **git push -u <remote> <local-branch>** - The **-u** flag (means upstream) or **--set-upstream** adds a tracking reference between the specified local branch and remote repo’s respective branch. This lets usage of **git push** and **git pull** without specifying any more arguments.

For example, once a **git push -u <remote> <local-branch>** is executed,

* + - * **git push** – pushes the local branch with the upstream set up to the remote-repo’s respective branch.
      * **git pull** – as per the set up upstream Git pulls the remote-branch that matches with the local-branch’s branch name.
      * The upstream link setup is specific and relates only to the respective local branch and the remote branch.
      * If **git pull** or **git push** is used from other branches where there is no upstream link set beforehand, then the attempt to push or pull between the remote and the local repo would fail.
        + In that case, the whole command (**git push/pull <remote> <local-branch>**) would have to be typed in, or a new upstream link (**git push/pull <remote> <local-branch>)** would have to be setup for that specific branch.
* **git push <remote> <local-branch>:<remote-branch>** - Git pushes the specified local repo’s branch to the remote repo’s specified branch (local-branch and the remote’s branch are colon separated), which is the destination branch of the local-branch being pushed into the remote repo.
  + **Upstream link** can be set here too, but not recommended; when linking local and remote repos of different branch aliases.
* **Remote Tracking of Repositories:** 
  + **git branch -r** – Displays the various branches at the remote repo.
  + **git branch -r -v** or **git branch -rv** – Displays the various branches at remote along with 7-character wide commit hashes and one line commit message.
* **git checkout <remote>/<remote-branch>** - We go into detached head state and checkout the specified branch of the remote repo (Head re-attaches via **git switch – (**HEAD goes to its previous position**)**or **git switch <branch-name> (**HEAD goes to the specified branch’s branch-reference**)**).
* By default, only the main/master branch is cloned. No other branches are displayed locally, nor there exists any copy of the other remote-branches locally. But various remote-branches can be viewed using **git branch -r** (or similar) command.
  + **git switch <remote-branch>** or **git checkout --track <remote>/<remote-branch>** when used on a *non-existent* local branch, it usually returns error. But in case of the branch name matching one of the remote repo’s branch names, a copy of the branch is locally created.
* **git fetch <remote>** or **git fetch** (if not mentioned, defaults to origin)- Takes changes (new commits) from the remote-branch and brings it to the local repo (as commits). But the changes are displayed as a fork (Origin->Main). Our branch structure (HEAD->Main) is preserved. Changes are available in our computer and not just the cloud.
  + **git fetch <remote> <remote-branch>** - Fetches a specific branch from among the remote - branches.

**git fetch** – Git fetches the latest changes from the remote-branch (this gives the user on the local-repo access to the latest files from *remote-repo* but without altering the *Working Tree*.

While using **git branch -r**, always **git fetch** to be sure that local and remote repos are up to date. As the local repos always tend to display that they are up to date with the remote repos!

* **git pull** or **git pull remote <local-branch>** - Takes the changes from the remote-branch and brings it to the local repo’s branch (both commits and Working Tree contents get changed).
  + **git pull** = **git fetch** (Updates the remote tracking branch with latest changes from the remote repo, i.e., commits) **+** **git merge** (Working Tree is updated with changes from remote).
  + **git pull <remote> <remote-branch>** (if upstream remote link is not set up! Or to be specific). OR,
  + **git pull** (from relevant branch at the local repo!). This only works as long as there is only one remote for this branch and that remote is ‘origin’ (most widely used remote name) and then, it will default to the one branch on the remote with similar name to the branch on local-repo (Note: Tracking is automatically set between remote and local repos while using **git switch** **<local-branch>**).

This can be configured otherwise, but generally not recommended.

* + It matters where the command is run. As changes will take effect at the active branch at the local repo.
  + May result in merge conflicts (can be handled normally in case of a conflict)!
  + Good Practice – **git fetch** before **git pull**!
* **git fetch** 
  + Gets changes from remote branch(es).
  + Updates the remote-tracking branches (upstream set up branches in local repo) with the new changes.
  + Does not alter the Working Tree.
  + Safe to do anytime.
* **git pull = git fetch + git merge**
  + Gets changes from remote branch(es).
  + Updates the current branch with the new changes, merges them.
  + Can result in merge conflicts.
  + Not recommended if you have uncommitted changes! As it alters the Working Tree.

## Git Rebase

* Helps in combining changes from two branches while helping cleanup git history cluttered because of Merge Commits.
  + - It is useful and quick when applied wisely!
    - An *alternative to Merging* committing.
    - *Cleans up commit history* (rewrites history).
* In order to clean up merge commits at the branch history because of lot of activity at the main branch.
  + To rebase, **git switch <feature-branch>**
  + **git rebase <default-branch>** - Rebases the commit history of the repo.
    - This places all the <feature-branch> commits serially and chronologically from the beginning, at the tip of the <default-branch>.
    - As a result, the Merge Commits are gone.
      * Original metadata and data are preserved, but these are newly created commits (a different hash value for the commits).
    - With every rebase the entire commit history is rewritten.
    - The branch at which **git rebase <default-branch>** is called gets rebased on to the **<**default-branch>!
      * This results with the feature-branch having all the default-branch commit history at the time the **git rebase <default-branch>** command was called upon.
      * So, while pushing our feature branch, this entire history (default branch commits (at the time of rebase) + feature-branch commits) gets pushed onto the remote repository (not good, very bad).
    - When **Not to Rebase**?

Never rebase commits that have been shared with others (through GitHub or similar tools).

* + - * + If you have commits pushed to the GitHub repo and used by others; *don’t rebase*.

This results in changing the commit history for everyone!

* + - * Only prefer rebasing while working on your own project.
      * Avoid Rebasing while collaborating with others.
* Interactive Rebase –
* Running git rebase with the -i option will enter the interactive mode, which allows us to edit commits, add files, drop commits, etc. Note that we need to specify how far back we want to rewrite commits.

**git rebase -i HEAD~4** – Interactively, rebase up to 4 commits from the current HEAD position.

Also, notice that we are not rebasing onto another branch. Instead, we are rebasing a series of commits onto the HEAD they currently are based on (alters only the branch the user is on).

* This changes the commit hash of the current commit as wells as the commit hashes of prior commits, up to the commit specified!
* P, pick <commit-hash> – Tells Git to use the marked commit, but Git will remake it (so, a new commit hash gets generated at the log).
* r, reword <commit-hash> – When selected at the editor, this allows us to reword the commit message (after the current editor closes and a new editor with the respective commit message appears, for us to edit.
* e, edit <commit-hash> – Use the commit, but stop for amending.
* s, squash <commit-hash> – Use the commit, but meld it into the previous commit.
* f, fixup <commit-hash> – Use the contents of the commit, but meld it into previous commit (adjacent commit that is chronologically newer) and discard the commit message.
* x, exec <commit-hash> - Run command (the rest of the line) using shell.
* b, break – Stop here (continue rebase later with ‘**git rebase --continue'**
* d, drop – Remove commit (both files and commit messages are gone).
* Move Up
* Move Down
* Search

## Git Tags

* Understanding Git Tags –
  + Used for labeling/naming a particular commit. Mostly developers use this for marking release points (*Versioning of releases*)*.*
* Versioning –
  + [Semantic Versioning](https://semver.org/) specification outlines a standardized versioning system for software releases by providing a consistent way for developers to give meaning to software releases through version designation split across three numbers that are period separated.

[**Major Release].[Minor Release].[Patch Release] => 5.2.1**

* Major Release signifies significant changes that are no longer backwards compatible. Features may be removed or changed substantially.
* Every new Major release resets the respective Minor and Patch identifiers to 0.
* Minor release signifies new features or functionalities that have been added, but the project is still backwards compatible with no breaking changes. The new functionality is optional and should not force users to rewrite their own code.
* Every new Minor release resets the Patch identifier to 0.
* Patch releases typically signify bug fixes and other changes that do not impact how code is used. They do not contain new features or significant changes.
* Viewing Tags:
  + **git tag** – will print a list of all the tags in the current repository.
  + Search for tags that match a particular pattern by using **git tag -l** and then passing in a *wildcard pattern*.
    - **git tag -l “\*beta\*”** – The ‘-l’ will display a list of all the tags that contain ‘beta’ in their tags.
      * The location of \* specifies the possibility of presence of characters (any character/s if any, based on their position).
  + **git show <tag-name>** - This displays the additional Meta Data related to the specified tag.
    - Tagger name, email id.
    - Tag creation datetime, time zone.
    - The commit hash, commit author, commit datetime, time zone info, diff.
* Diffing Tags (Comparing Tags)
  + A tag always points to a commit. So, when a tag is being checked out, we are in Detached head.
  + To checkout a tag: **git checkout <tag-identifier>**
  + **git diff <tag-identifier> <another tag-identifier>** - Just like diffing two different commits (which is what this is, exactly!).
* Two types of Tags
  + Lightweight Tags are just a name/label that points to a particular commit.
    - **git tag <tag-name>** - Git will create a light weight tag with the specified name at the commit HEAD is referring to.
  + Annotated Tags store extra meta data; including the Tag author’s name, email, date and a tagging message (akin to a commit message, but not the same).
    - **git tag -a <tag-name>** - Git will open the default Git editor and prompt for additional information (this point to the commit HEAD referring currently).
    - **git tag -a <tag-name> -m”**message, one line**” -** Similar to **git commit**, we can also pass use the **-m** option to pass a message directly and forgo the opening of the text editor.
* For tagging previous commits
  + **git tag <tag-name> <commit-hash>** - For tagging previous commits (by directly specifying the commit, without entering detached HEAD state!).
  + **git tag -a <tag-name> <commit-hash>**
  + **git tag -a <tag-name> <commit-hash> -m**
* Moving Tags
  + Tag names/labels are unique. So, a tag label can be only moved, not duplicated.
  + **git tag -f <tag-name> <commit-hash>** - Forcefully moves the tag label to the specified commit hash (the **-f** can be placed anywhere in the command).
  + Deleting Tags: **git tag -d <tag-name>** - Git deletes the specified tag associated with some commit, but not the commit.
* Pushing Tags
  + By default, the **git push** command doesn’t transfer tags to the remote while pushing commits (**git push**).
  + **git push <remote> <tag-name>** - Pushes the specified tag to the specified remote (only one tag at a time).
  + **git push <remote> --tags –** Git *pushes tags* from the local machine that are not already on the respective (in case of **-u** upstream setup, <remote> name can be omitted) remote (multiple tags at once).

## Git Reflog

* Reflog is shortform for reference logs. They are logs (transcript or record(s)) that Git maintains for us, whenever references are updated.
* Git keeps record (@ **.git/log/** of individual files respectively for HEAD (**<HEAD>**, each <local-branch(s)> (**heads/main**, **heads/<local-branch>**) and each <remote-branch(s)> (**remotes** (per remote) **/<remote-branch>**)) of when the tips of branches and other references were updated in the repo.
* **Limitations:**
  + Git only keeps reflogs about local activity (not remote repos).
  + They are not shared with collaborators.
  + Reflogs also expire. Git cleans out old entries after around 90 days, through this can be configured.
* **git reflog** – The git reflog command accepts subcommands **show, expire, delete** and **exists.** Show is the only commonly used variant, and it is the default subcommand.

**git reflog show** will show the log of a specific reference (it defaults to HEAD).

For example, to view the logs for the tip of the **main** branch we could run **git reflog show main**.

* Reflog References: Specific Git reference pointers can be accessed through the **name@{qualifier}** syntax (name represents a reference).
  + **git reflog show HEAD@{10}** – This will show the contents of reflogs for HEAD from the beginning till 10 moves ago from the current position of HEAD.
  + This can also be passed to other commands including **git checkout**, **git reset** and **git merge**.
    - **git checkout HEAD@{2}** may or may not be equal to **git checkout HEAD~2**.
      * **HEAD@{qualifier}** – specifies Position of HEAD two moves ago in the HEAD reflogs.
      * Whereas, **HEAD~2** or **HEAD~{qualifier}** means go to position of HEAD 2/qualifier commit positions ago.
  + **Timed References:**
    - Every single reflog entry has a commit-hash, the user.name that created it and the timestamp (Unix-timestamp) at which it got created.

Reflog entries can be filtered through **time/date** qualifiers like,

* + - * **1.day.ago**
      * **3.minutes.ago**
      * **Yesterday**
      * **Fri, 12 Feb 2021 14:06:21 -0800**.
      * For example,
        + **git reflog main@{one.week.ago}** or **git reflog main@{one.minute.ago}**
        + **git checkout bugfix@{2.days.ago}** or **git checkout bugfix@{two.days.ago}**
        + **git diff main@{0} main@{yesterday}**
* **Reflog Rescue:**
  + Reflog entries can sometimes be used to access commits that seem lost and are not appearing in **git log**.
  + Even after a hard reset (**git reset --hard <commit-hash>**)files may be retrieved and saved (**if, it wasn’t too late!** It’s because git cleans up disconnected/dangling (no parent or child) commits periodically).
    - When reset (hard) the commits before the specified commit are deleted.
    - But in the reflogs the commits are still retained. So, the commit has can be retrieved from the reference logs and files accessed again via detached head state and saved, if required.
    - Or a **git reset --hard master@{1}** i.e. **git reset --hard name@{qualifier}** (based on the reflog) to restore the commit.
    - **Reflogs only work with local changes**!
  + Undoing a **git rebase** through **git reflog.**
    - Same concept as above.
    - But, in case of commit hash **git reset --hard <commit-hash** only found @ reflog**>** - know that each commit except the initial commit has its parent commit and children commit(s) (if any)!
    - So, based on the position of the commit-hash, one or more commits will be successfully restored.

## Git Aliases

* Config File:
  + Local Config File – Applies to the local repo in which it is configured.
    - The local config file per repo is present in the file path is **.git/config**.
    - **git config --local user.name “<user-name>” –** Sets local user name.
    - **git config --local user.email “<user-email>” –** Sets local user email.
    - **git config --local user.name** – Displays user.name that is set locally.
    - **git config --local user.email –** Displays user.email that is set locally.
  + Global Config File – Applies globally across the account.
    - Git looks for the global config file either at **~/.gitconfig** or **~/.config/git/config**. Any configuration variables that get changed will be applied across all Git repos.
    - Altering config files from the command line
      * **git config --global user.name “<user-name>” –** Sets global user name.
      * **git config --global user.email “<user-email>” –** Sets global user email.
      * **git config --global user.name** – Displays user.name that is set globally.
      * **git config --global user.email –** Displays user.email that is set globally.
  + System Config File – Applies System wide. Across all accounts in the system.
* Writing Basic Aliases
  + Git Aliases can be set up to make the Git experience a bit simpler and faster. They are just custom command that user can set up to correspond with actual Git command.
  + For example, **“**git ci**”** can be set up as an alias for **git commit**.
  + Similarly **“**git lg**”** can be set up as an alias for a formatted commit log.
  + In the global config file under section,

**[alias]**

s = status; **git s** – for git status.

l = log; **git l** – for git log.

**<custom-command>** = **<known-git-command>**

cm = **commit -m** – for git commit -m. The argument(s) passed are appended to the end of the command by git. (**Aliases with arguments**)

* From the command line,

**git config --global alias.<**custom-command**> <git-command-to-alias>**. As **global** command is used, this is set in the *global config file*. If **local** command is used, this will be set in the *local config file*.

* **git ls** – One line git log with the name of the user who committed it (Added from online).

When **!** is used in quotes along with a git command, then it’s a shell script.

# Git-Bash (WINDOWS SPECIFIC) COMMANDS

* **touch <file-name.ext>** - creates the file with the specified extension at the PWD.
* **touch <file-1.ext> <file-2.ext>** – makes new files with the specified filenames that are space separated.
* **mkdir <directory-name>** – creates a directory with the specified filename.
* **mkdir <dir-name1> <dir-name2>** - makes new directories with the specified names that are space separated.
* **cd directory/path** - changes to that directory
* **cd ..** - takes a step back
* **PWD** - displays the full path of the current directory or present Working Tree.
* **mv <file-name> <directory-name>** – moves the specified file to the specified directory.
* **mv <file-name1> <file-name2> directory-name** – moves the specified files that are space separated to the specified directory.
* **cat <file>** or **cat path/<file>** - reveals the contents of the file.
* **start .** - opens the current directory in File Explorer.
* **code . <file.ext>** - Opens the specified file with VS Code.
* **ls** - to display list of files in the PWD if path is not provided (if path is provided then contents of that directory are displayed).
* **ls -a** - same as the ls command but displays hidden files too.
* Hidden files start with a period. For example: **.<filename>** or **.<directory-name>/**
* Pressing tab cycles through the available files in the directory.
* After partially typing a file name we can press tab to auto complete it.
* **echo ‘hello’ | git hash-object --stdin** – Piping (|) ‘hello’ to git hash-object --stdin. [The objects/ directory has info/ and pack/ directories in them at initiation.]
* **echo ‘phrase’** – The echo command simply repeats whatever is passed to it within quotes. Here, it echoes the sha-1 output of the phrase passed in quotes.
* **echo “phrase” > <file.ext>** - The word within quotes (phrase) gets written in the file specified.
* **echo “phew phew” > phew.txt –** writes the contents “phew phew” into the file phew.txt. at the PWD.

# Git, behind the scenes!

## What is in .git?

There are a lot of contents. But the ones mentioned below are the most important.

* objects/ directory contains all the repo files.
  + This is where Git stores everything;
    - the backups of files,
    - the commits in a repo and more.
  + The files are all compressed and encrypted so they won’t make sense!
  + Objects/ contain subdirectories that are dual digit hexadecimal names and inside them are encrypted files. [Note: Git stores full snapshots].
  + 4 different types of basic Git objects. They are commit, tree, blob, annotated tag.
* Additional, on how Git works (more @ Git Behind The Scenes),

1. Git is a key-value data store.
   * + We can **insert any kind of content** (give Git any type of content) into a Git Repository and Git will hand us back a unique key that we can later use to retrieve that content.
     + These **key**s that we get back are SHA-1 checksums.
     + [Note: Git uses SHA-1 in several places. Not just commits and key-value data store].
2. Git uses SHA-1 hashing function. It outputs Hexadecimal (Base 16) characters exactly 40 characters wide (no matter what the size of the input is!) called SHA-1 checksums.

Hashing functions are functions that map input data of some arbitrary size to fixed-size output values (i.e., they take inputs of variable size and give out fixed size outputs).

1. Characteristics of Cryptographic Hash Functions:
   * It’s one way encryption function which is infeasible to invert (not reversable or reverse engineered).
   * Even a small change in input yields a large change in output.
   * Deterministic – same input yields same output.
   * Unlikely to find two outputs with same value (mathematical probability of such occurrences is very low, though not null, but this is negligible in real life).
2. Hashing in Git (~ Manually storing data in Git at low level!):
   * **git hash-object –stdin** – The **git hash-object** returns a unique SHA-1 hash for the content provided; this hash refers to that data object! Here, **--stdin** specifies that the item in the standard input is to be used instead of a file (which is provided).
     + Typically, **git hash-object <file>** - returns a SHA-1 hash value for the provided data (a file here) at the objects/ directory and this refers to the data (file here) provided to Git.
     + **git hash-object --stdin -w** – The **-w** (write option) tells the **git hash-object** command to write to the **objects/** directory.
     + When a file is generated, it is located at the **objects/** directory.
     + The Sha-1 hash function outputs a 40 character hexadecimal (Base-16) hash.
     + The first two characters are used to name a directory within which a file is generated and is named with the rest of the 38-character sha-1 hash checksum.
   * **git cat-file -p <object-hash>** - the **git cat-file** command with the **-p** (pretty print option) prints the decrypted 40-character SHA-1 hash that was provided.
   * **git cat-file -p <object-hash> > <file>** - Prints the decrypted content as well as writes the contents into the provided file (Git creates file (because of this option >), which matches the label provided while running the command).
   * **Blobs** (Git object) (the 2 (directory name) + 38 character hash-object output)– Git blobs (binary large object) are the object type Git uses to store the **contents of files** in a given repository. Blobs don’t even include the filenames of each file or any other data. They just store the contents of a file (Git stores names separately)!
   * **Trees** are Git objects used to store the contents of a directory. Each tree contains pointers that can refer to blobs and to other trees.

Each entry in a tree contains the SHA-1 hash of a blob or tree, as well as the mode, type and filename.

* + - Trees (tree object) represent directories.
    - The tree object refers to blobs or other nested tree objects.
    - The tree object refers to blobs or other nested trees via hashes and these hashes are mapped to the respective file-names or other tree object labels internally.
    - The Blob (blob object) represents the contents of the file.
    - **git cat-file -p <branch-name>^{tree}** – git shows the tree structure of the branch reference of the specified branch.
    - **git cat-file -t <blob-hash> -** Git returns the type of git-object the hash is.
  + **Commits** – *Commit objects* (Git object) combine a tree object along with information about the context that led to the current tree.

Commits store references to its **parent commit(s)**, **the author**, **the committer** and of course the **commit message**!

* <config> – Used to alter the configuration on global or local (per repo) scales.
  + Global changes are performed at the system’s .git\ folder (. Signifies hidden file).
  + Local changes are performed at the respective repo’s .git\ folder.
  + **git config --global user.name “**value”
  + **git config --global user.email “**value”
  + **git config --local user.name** “value”
  + **git config --local user.email** “value”
  + Lot of options can be altered. Docs available.
* HEAD is a text file that keeps track of where HEAD points to, in the repo!
  + It is a plain text file in the .git/ folder.
  + It either contains a commit-hash (when in detached head state! It directly points to a commit-hash.)
  + Or it refers to a branch reference informally, branch pointer (reference to a branch’s latest commit (where the branch-head usually points at), (e.g. a HEAD file contains the text, ref: refs/heads/main (reference to the main branch’s branch reference (contains latest commit-hash of the respective branch, here main).
* Index – Staging area.
* refs/ - This is a folder that contains three other folders called heads\, remotes\, tags\.
  + The refs/**heads** folder contains files (pointers, here) that contain the names of the **branches** that are present in the repo.
    - Git makes one file per branch and each file’s name represents a branch and the contents of the file is a **<commit-hash>** which is actually where the branch reference is pointing to.
  + In the refs/**tags** folder there are files which represent the tags in the repo.
    - Git makes one file per tag (each tag is unique) and each file’s name represents a tag’s name (label) in the repo its content is a unique **<commit-hash>** to which the respective tag is pointing to.
  + The refs/**remotes** folder contains a folder per <remote> that is setup.
    - The files inside represent HEAD and other <remote-branches>.
    - The contents of each file is a <remote-commit-hash> (a pointer reference).
      * There are files for HEAD and <remote-branches>.
        + The file named HEAD represents the HEAD at the respective <remote> repo and its content is the commit-hash to which the HEAD at the <remote> is currently pointing to.
        + For the branches the <commit-hash> is the <remote-branch-reference-commit-hash>.

# GITHUB

* **ls -al ~/.ssh** – Checks for existing SSH (Secure Shell) keys.
* SSH Keys– use GitHub SSH keys to work from individual machines. Instead of directly using the username and password.
* **Existing Repo:**
  + Create a new Repo on GitHub.
  + Connect local repo (add a remote).
  + Push up changes to GitHub.
* **Start From Scratch:**
  + Create a brand-new repo on GitHub.
  + Clone it down to the local machine (Once a repo is cloned, it knows about the cloud repo and doesn’t have to be reconnected to push changes to that repo).
  + Work locally.
  + Push up changes to GitHub.
* **Types of Repositories** based on visibility. Can be changed anytime (with ramifications).
  + Public
    - Anyone can see.
    - People without collaborator privileges can only view but not push changes.
    - Can be cloned by public, make changes to their repo. But cannot push changes without permission (collaboration privilege).
  + Private
    - Only accessible to the owner and people who have been granted access to, by the owner.
      * In case of enterprise level GitHub account, private repositories are visible to anyone within the organization.
* **Collaboration / Contribution** – collaborators or contributors
  + One has to be invited.
  + The invitee has to accept the collaboration invitation to begin collaborating on the respective repository.
  + A collaborator/contributors can view repo, push changes etc. But has no access to the settings. It resides with the owner.
    - Collaborator cants:
      * Can’t delete the repo!
      * Can’t invite other people or remove their privileges.
  + But organization and enterprise tiers have options for different owners and permissions level for collaborators.
* **README.md** - .md means markdown. In this file details about the project, guidelines, license etc. can be entered. It will be displayed in the landing page of the GitHub repo, for all to see. It’s a good introductory page for the project to anyone (especially the uninitiated).
* **GitHub Gist**
  + It can be a markdown file, but it can be file with any extension.
  + Its purpose is like a one-page repo (I guess). Without placing the same burden on the person like a GitHub repo.
  + It can show diffs/revision history etc.
* **GitHub Pages** – Static sites are preferred. Low traffic is recommended.
  + GitHub auto detects gh-pages branch name!
* **Workflow:**
  + **Centralized Workflow:**
    - Everyone works on the Main/Master or Some single branch.
    - While trying to push some work, if respective branch has commits that is not In local repo; Git wouldn’t allow that.
      * In that case, work has to pulled and updated at local repo and then local work has to be pushed to remote repo.
    - Bad Idea, though manageable for very small teams!
  + **Feature Branch Workflow:**
    - No one works on Master/Main.
    - People always work on separate Feature Branches.
      * And Merge back in when appropriate or abandoned if not required.
      * This keeps the main codebase safe.
    - Very common and is the best practice.
* **Pull Requests** or **PR:**
  + To compare changes across two branches before merging them; instead of just merging them without forethought.
  + Standard practice, recommended.
  + Feature Branch owner generates a Pull Request and sends to the Person Responsible for the Protected Branch.
  + Person Responsible for the Protected branch,
    - Compares, approves and Merges the Pull Request (one way of closing the Pull request).
    - Compares and discusses changes that are required and communicates through comment.
      * May be invite more people to the discussion, etc.
      * Either approve or reject the changes.
  + As always there is potential for conflicts!
    - Resolvable at GitHub or locally.
    - Locally, fetch the <remote>
      * Switch to the Feature branch.
      * From here, merge Main branch (best to prevent Fast Forward Merging to preserver History).
      * Resolve conflicts!
* **Fork-And-Clone Workflow:**
  + **Forking: GitHub** or similar tools allow users to make copies of other people’s repositories.
    - Access is essential for Forking to succeed.
      * Public – accessible.
      * Private – has to be a collaborator!
    - Such a copy is called a ‘**Fork**’ of the original and is a perfect clone of the forked parent repo (**Pull requests and Forking** are GitHub feature!).
    - It’s perfect to make changes and Push changes to huge Open-Source Projects, via GitHub (GitHub keeps track of the changes between these repos)!
      * Forks can be cloned on to local machine and worked on like any other Git repo!
      * From there changes can be pushed (via Pull Request) to Original project via GitHub through the Forked remote repo.
    - Instead of one centralized GitHub repository, every Developer has their own GitHub repo (very common in large open-source projects with thousands of contributors but with only a few maintainers).
    - Best practice,
      * Set up a remote to the original repo at the local machine (**git remote add <label> <URL>**)!
      * git remote add upstream URL – typically named ‘upstream’.
* **Branch Protection Rules:**
  + There can be only one **Default** branch.
  + Different Rules can be applied to protect the Default branch from forced or accidental commits and merging!
  + Reviews requirements and Approval requirements can be set for each Pull Request!