# Git Data transport



# Ref

## Q) What is ref

Git is all about commits. The most direct way to reference a commit is via its SHA-1 hash. This acts as the unique ID for each commit. You can find the hash of all your commits in the git log output.

A **ref** is an indirect way of referring to a commit. You can think of it as a user-friendly alias for a commit hash. This is Git’s internal mechanism of representing branches and tags.

Refs are stored as normal text files in the .git/refs directory.

The heads directory defines all of the local branches in your repository. Each filename matches the name of the corresponding branch, and inside the file you’ll find a commit hash. This commit hash is the location of the tip of the branch.

The tags directory works the exact same way, but it contains tags instead of branches.

The remotes directory lists all remote repositories that you created with git remote as separate subdirectories. Inside each one, you’ll find all the remote branches that have been fetched into your repository.

When passing a ref to a Git command, you can either define the full name of the ref, or use a short name and let Git search for a matching ref.



The some-feature argument in the above command is actually a short name for the branch. Git resolves this to refs/heads/some-feature before using it. You can also specify the full ref on the command line, like so:



This avoids any ambiguity regarding the location of the ref. This is necessary, for instance, if you had both a tag and a branch called some-feature. However, if you’re using proper naming conventions, ambiguity between tags and branches shouldn’t generally be a problem.

## Q) Packed Refs

For large repositories, Git will periodically perform a garbage collection to remove unnecessary objects and compress refs into a single file for more efficient performance. You can force this compression with the garbage collection command:



This moves all of the individual branch and tag files in the refsfolder into a single file called packed-refs located in the top of the .git directory. On the outside, normal Git functionality won’t be affected in any way. But, if you’re wondering why your .git/refs folder is empty, this is where the refs went.

## Q) Special Refs

In addition to the refs directory, there are a few special refs that reside in the top-level .git directory. They are listed below:

* HEAD – The currently checked-out commit/branch.
* FETCH\_HEAD – The most recently fetched branch from a remote repo.
* ORIG\_HEAD – A backup reference to HEAD before drastic changes to it.
* MERGE\_HEAD – The commit(s) that you’re merging into the current branch with git merge.
* CHERRY\_PICK\_HEAD – The commit that you’re cherry-picking.

The HEAD ref can contain either a **symbolic ref**, which is simply a reference to another ref instead of a commit hash, or a commit hash.



This will output ref: refs/heads/master, which means that HEAD points to the refs/heads/master ref. This is how Git knows that the master branch is currently checked out. If you were to switch to another branch, the contents of HEAD would be updated to reflect the new branch. But, if you were to check out a commit instead of a branch, HEAD would contain a commit hash instead of a symbolic ref. This is how Git knows that it’s in a detached HEAD state.

## Q) Refspecs

A refspec maps a branch in the local repository to a branch in a remote repository.

A refspec is specified as [+]<src>:<dst>. The <src> parameter is the source branch in the local repository, and the <dst>parameter is the destination branch in the remote repository. The optional + sign is for forcing the remote repository to perform a non-fast-forward update.

For example, the following command pushes the master branch to the origin remote repo like an ordinary git push, but it uses qa-master as the name for the branch in the origin repo. This is useful for QA teams that need to push their own branches to a remote repo.



You can delete them by pushing a refspec that has an empty <src> parameter, like so:





## Q) Relative Refs

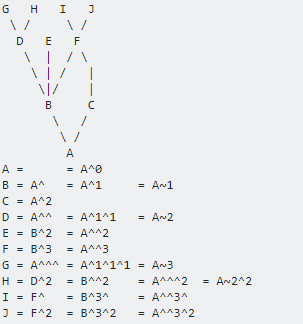
The ~character lets you reach parent commits. HEAD~2 means the grandparent of the tip of the master branch, favouring the first parent in cases of ambiguity.



You can also refer to commits relative to another commit. The ~character lets you reach parent commits.

The ~ character will always follow the *first* parent of a merge commit. If you want to follow a different parent, you need to specify which one with the ^ character.

HEAD^ means the first parent of the tip of the current branch. Remember that git commits can have more than one parent. HEAD^ is short for HEAD^1, and you can also address HEAD^2 and so on as appropriate.



## Q) Ref log

To view the reflog, run the git reflog command

# Saving changes

## Git stash

By default, running git stash will stash:

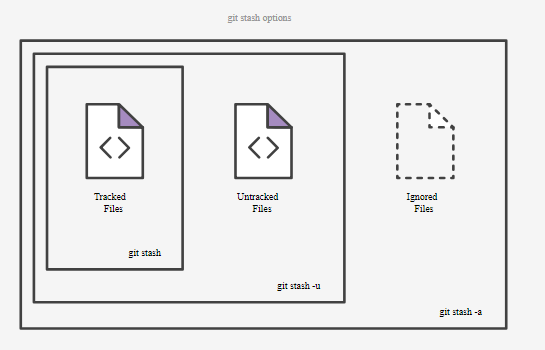
* changes that have been added to your index (staged changes)
* changes made to files that are currently tracked by Git (unstaged changes)

But it will not stash:

* new files in your working copy that have not yet been staged
* files that have been [ignored](https://www.atlassian.com/git/tutorials/gitignore)

Adding the -u option (or --include-untracked) tells git stashto also stash your untracked files.

You can include changes to [ignored](https://www.atlassian.com/git/tutorials/gitignore) files as well by passing the -a option (or --all) when running git stash.



$ git stash list

$ git stash save "add style to our site"

$ git stash pop stash@{2}

$ git stash show

$ git stash show –p

$ git stash drop stash@{1}

$ git stash clear

$ git stash -p

**How stash works**

Stashes are actually encoded in your repository as commit objects. The special ref at .git/refs/stash points to your most recently created stash, and previously created stashes are referenced by the stash ref's reflog. This is why you refer to stashes by stash@{n}: you're actually referring to the nth reflog entry for the stash ref. Since a stash is just a commit, you can inspect it with git log:

$ git log --oneline --graph stash@{0}

\*-. 953ddde WIP on master: 5002d47 our new homepage

|\ \

| | \* 24b35a1 untracked files on master: 5002d47 our new homepage

| \* 7023dd4 index on master: 5002d47 our new homepage

|/

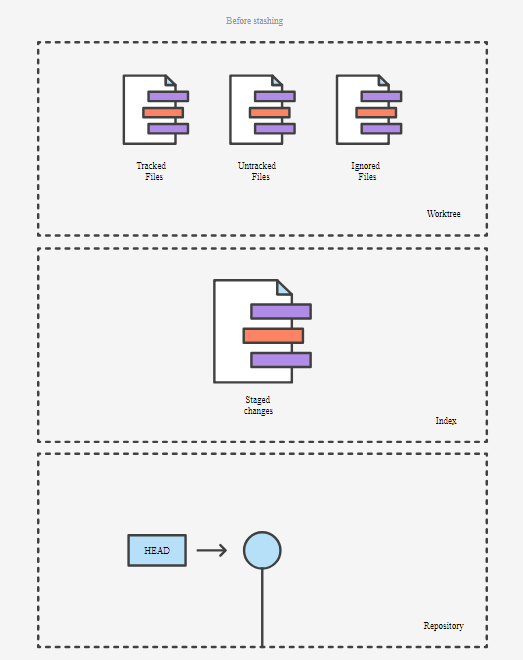
\* 5002d47 our new homepage

Depending on what you stashed, a single git stash operation creates either two or three new commits. The commits in the diagram above are:

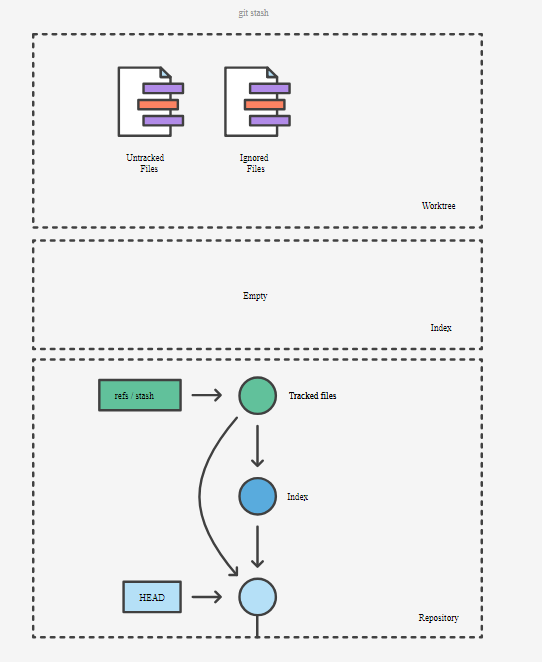
* stash@{0}, a new commit to store the tracked files that were in your working copy when you ran git stash
* stash@{0}'s first parent, the pre-existing commit that was at HEAD when you ran git stash
* stash@{0}'s second parent, a new commit representing the index when you ran git stash
* stash@{0}'s third parent, a new commit representing untracked files that were in your working copy when you ran git stash. This third parent only created if:
  + your working copy actually contained untracked files; and
  + you specified the --include-untracked or --all option when invoked git stash.

How git stash encodes your worktree and index as commits:

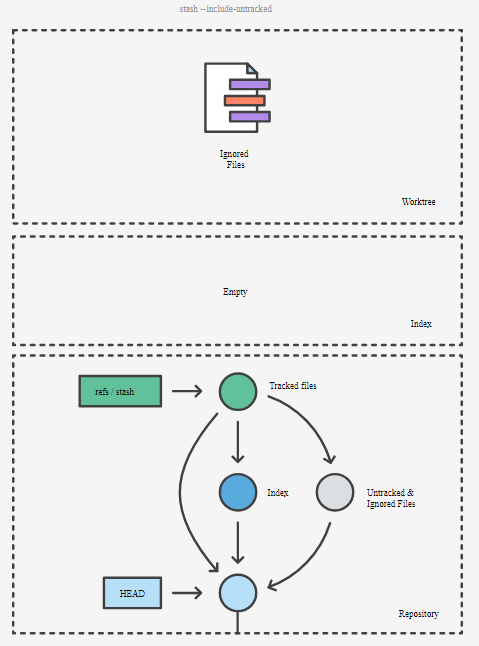
Before stashing, your worktree may contain changes to tracked files, untracked files, and ignored files. Some of these changes may also be staged in the index.



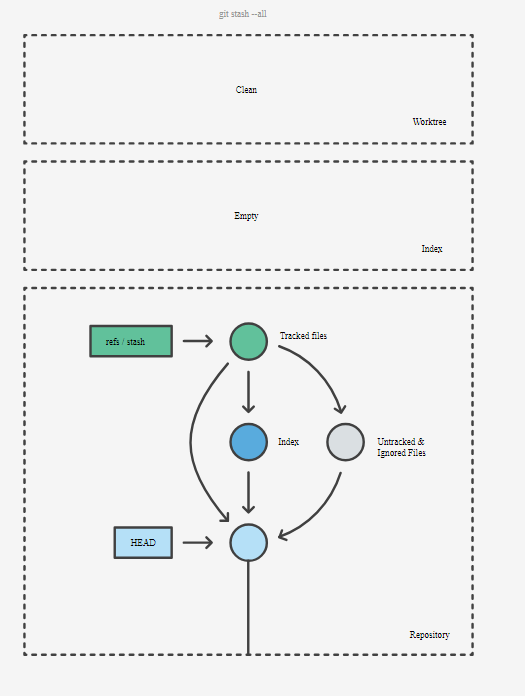
Invoking git stash encodes any changes to tracked files as two new commits in your DAG: one for unstaged changes, and one for changes staged in the index. The special refs/stash ref is updated to point to them.



Using the --include-untracked option also encodes any changes to untracked files as an additional commit.



Using the --all option includes changes to any ignored files alongside changes to untracked files in the same commit.



When you run git stash pop, the changes from the commits above are used to update your working copy and index, and the stash reflog is shuffled to remove the popped commit. Note that the popped commits aren't immediately deleted, but do become candidates for future garbage collection.

# Collaborating

## Git Remote

The git remote command is essentially an interface for managing a list of remote entries that are stored in the repository's ./.git/config file.



Running the command above adds a section to your repository’s .git/config file, specifying the name of the remote (origin), the URL of the remote repository, and the *refspec* to be used for fetching:



The format of the refspec is, first, an optional +, followed by <src>:<dst>, where <src> is the pattern for references on the remote side and <dst> is where those references will be tracked locally. The + tells Git to update the reference even if it isn’t a fast-forward.

In the default case that is automatically written by a git remote add command, Git fetches all the references under refs/heads/ on the server and writes them to refs/remotes/origin/ locally.

## Git Fetch

You can do a git fetch at any time to update your remote-tracking branches under refs/remotes/<remote>/.

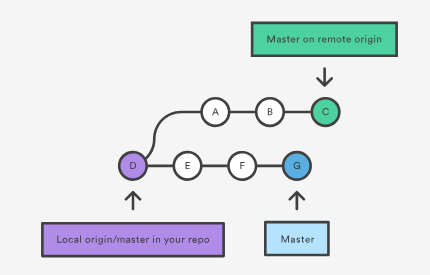
This operation never changes any of your own local branches under refs/heads, and is safe to do without changing your working copy.

## Git Pull

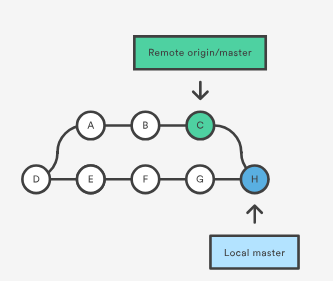
**Git fetch - merge**

In the simplest terms, git pull does a git fetch followed by a git merge.

enter image description here

****

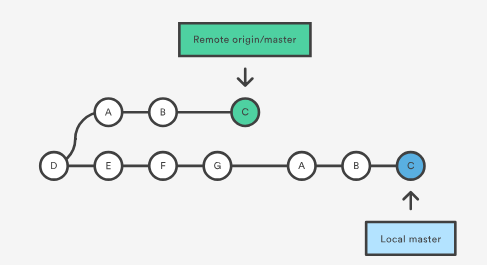
In this scenario, git pull will download all the changes from the point where the local and master diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.

****

In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message.

**Git fetch - rebase**

git pull --rebase



We can now see that a rebase pull does not create the new H commit. Instead, the rebase has copied the remote commits A--B--C and appended them to the local origin/master commit history.

## Git Push



To prevent you from overwriting commits, Git won’t let you push when it results in a non-fast-forward merge in the destination repository.



The --force flag overrides this behavior and makes the remote repository’s branch match your local one, deleting any upstream changes that may have occurred since you last pulled.

## Git Merge

**merge steps**

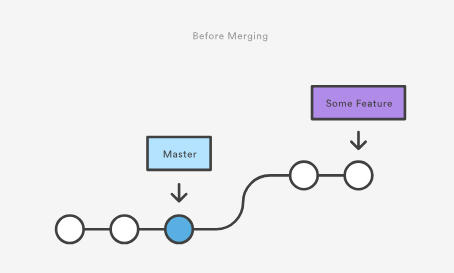
Execute git status to ensure that HEAD is pointing to the correct merge-receiving branch. If needed, execute git checkout <receiving> to switch to the receiving branch. In our case we will execute git checkout master.

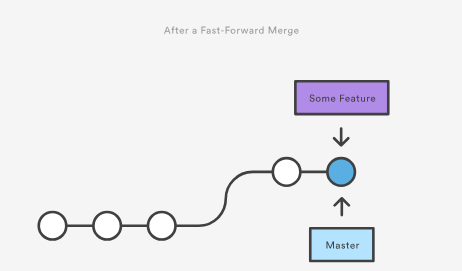
Make sure the receiving branch and the merging branch are up-to-date with the latest remote changes. Execute git fetch to pull the latest remote commits.

"preparing to merge" steps have been taken a merge can be initiated by executing git merge <branch name> where <branch name> is the name of the branch that will be merged into the receiving branch.

**fast-forward merge**

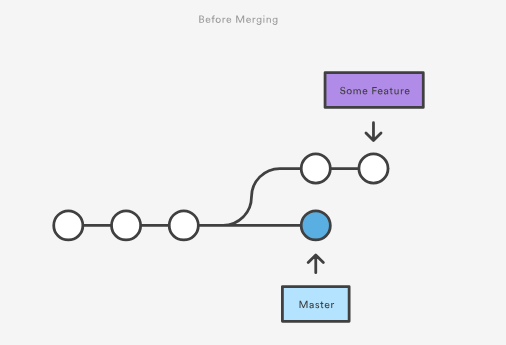
A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch.

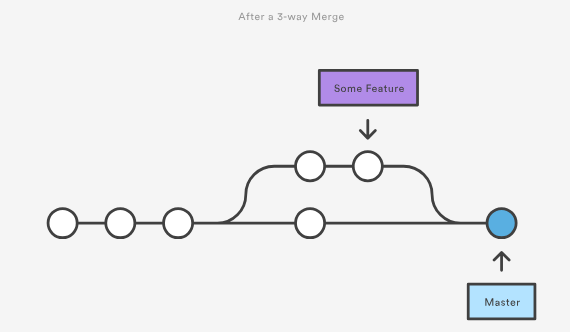




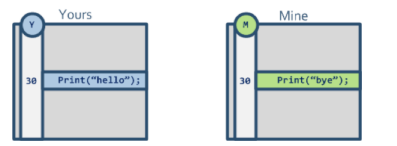
**3-way merge**

However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.





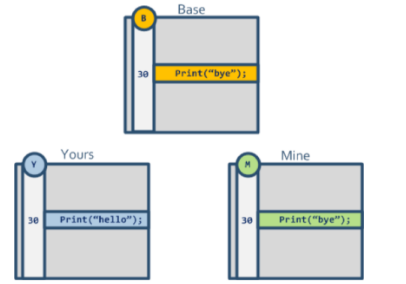
If you were doing a two-way merge (in other words, a diff), the tool could compare the two files, and see that the first and last lines are different.



This third person looking at the files sees there's a difference on line 30 but:

* How can he tell whether you modified line 30 or if I modified it?
* What if we both modified the line? How can he tell?

With a three-way merge, it can compare the two files, but it can also compare each of them against the original copy (before either of you changed it). So it can see that you removed the first line, and that your friend added the last line. And it can use that information to produce the merged version.



## Git Rebase

**What is Rebase?**

Rebasing is the process of moving or combining a sequence of commits to a new base commit.

From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base.

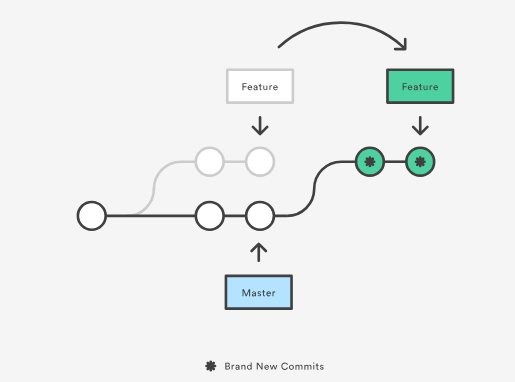
Often, you’ll do this to make sure your commits apply cleanly on a remote branch — perhaps in a project to which you’re trying to contribute but that you don’t maintain. In this case, you’d do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn’t have to do any integration work — just a fast-forward or a clean apply.

**Benefits of rebase**

The primary reason for rebasing is to maintain a linear project history. You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history.

**Rebase on top of master**



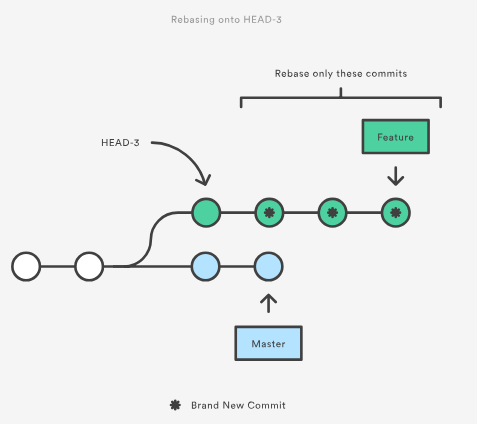


At this point, you can go back to the master branch and do a fast-forward merge.



**Rewrite the history**



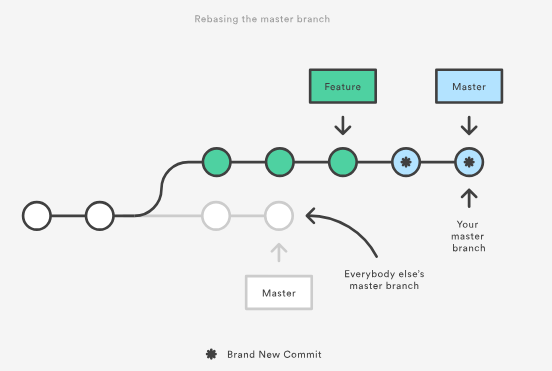


Eliminating insignificant commits like this makes your feature’s history much easier to understand.

**Don't rebase public history**

The golden rule of git rebase is to never use it on *public* branches.

For example, think about what would happen if you rebased master onto your feature branch:



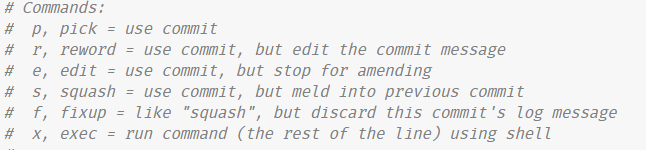
The rebase moves all of the commits in master onto the tip of feature. The problem is that this only happened in your repository. All of the other developers are still working with the original master. Since rebasing results in brand new commits, Git will think that your master branch’s history has diverged from everybody else’s.

The only way to synchronize the two master branches is to merge them back together, resulting in an extra merge commit and two sets of commits that contain the same changes (the original ones, and the ones from your rebased branch). Needless to say, this is a very confusing situation.

**Interactive rebase**

Running git rebase with the -i flag begins an interactive rebasing session. Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits.





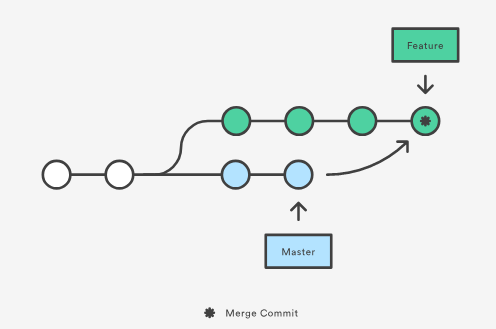
**Force pushing**

Q) Merge

Merge the master branch into the feature branch using something like the following:







Merging is nice because it’s a *non-destructive* operation. The existing branches are not changed in any way. On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit.

## Git Cheery pick

Let’s say you are working in a project where you are making changes in a branch called new-features. You have already made a few commits but want to move just one of them into the master branch.

From new-features branch run git log --oneline  to get a better log of your commits history. Note that the commit hash is what we need to start the cherry picking.



1. Checkout the branch where you want to cherry pick the specific commits. In this case master branch:

git checkout master

1. Now we can cherry pick from new-features branch:

git cherry-pick d467740

1. This will cherry pick the commit with hash d467740 and add it as a new commit on the master branch. Note: it will have a new (and different) commit ID in the master branch.

If you want to cherry pick more than one commit in one go, you can add their commit IDs separated by a space:

git cherry-pick d467740 de906d4

1. If the cherry picking gets halted because of conflicts, resolve them and

git cherry-pick –continue

1. If you want to bail of this step out altogether, just type:

git cherry-pick –abort

1. After all this is done, you can simply push the new commits to the upstream repo (e.g origin) and get on with your day.

# Git log

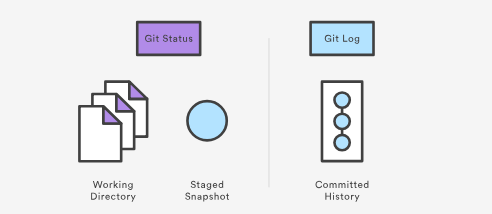
## Git status

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git.

For the untracked file, if those files have been put into .gitignore, then those files won’t display in the git status.

## Git log

The git log command displays committed snapshots.





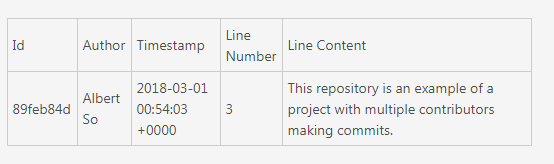
This will display a full diff of all the changes John Smith has made to the file hello.py.



 The displays a brief overview of all the commits that are in some-feature that are not in master.

## Git blame

The git blame command is used to examine the contents of a file line by line and see when each line was last modified and who the author of the modifications was.



# Git Undo

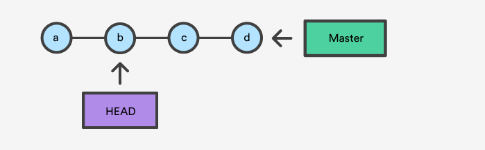
## Q) Checkout

### 1) Viewing an old revision



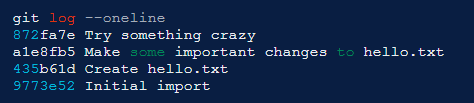
This makes your working directory match the exact state of the a1e8fb5 commit.

Checking out a specific commit will put the repo in a "detached HEAD" state. This means you are no longer working on any branch. In a detached state, any new commits you make will be orphaned when you change branches back to an established branch. Orphaned commits are up for deletion by Git's garbage collector. The garbage collector runs on a configured interval and permanently destroys orphaned commits. To prevent orphaned commits from being garbage collected, we need to ensure we are on a branch.



With git checkout, the master ref is still pointing to d. The HEAD ref has been moved, and now points at commit b. The repo is now in a 'detached HEAD' state.

### 2) Undo uncommitted snapshot



First step:



Second Step:

git checkout -b new\_branch\_without\_crazy\_commit.

This will create a new branch named new\_branch\_without\_crazy\_commitand switch to that state. The repo is now on a new history timeline in which the 872fa7e commit no longer exists. At this point, we can continue work on this new branch in which the 872fa7e commit no longer exists and consider it 'undone'.

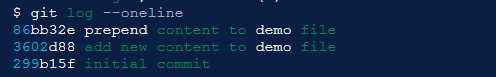
### 3) Undo uncommitted tracked files

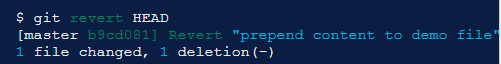
git checkout --.

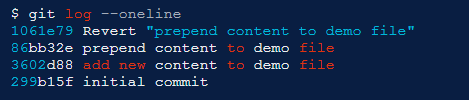
git checkout --<file path>

## Q) Revert

Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.







Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 2nd and 4th commits represent the exact same code base and the 3rd commit is still in our history just in case we want to go back to it down the road.

## Q) Reset

### 1) Three trees of git

Git's internal state management systems. Sometimes these mechanisms are called Git's "three trees". Trees may be a misnomer, as they are not strictly traditional tree data-structures. They are, however, node and pointer-based data structures that Git uses to track a timeline of edits.

**The working directory**

The first tree we will examine is "The Working Directory". This tree is in sync with the local file system and is representative of the immediate changes made to content in files and directories

Invoking git status shows that Git is aware of the changes to the file. These changes are currently a part of the first tree, "The Working Directory". Git status can be used to show changes to the Working Directory. They will be displayed in the red with a 'modified' prefix.

**Staging Index**

'Staging Index' tree is tracking Working Directory changes that have been promoted with git add, to be stored in the next commit. This tree is a complex internal caching mechanism. Git generally tries to hide the implementation details of the Staging Index from the user.

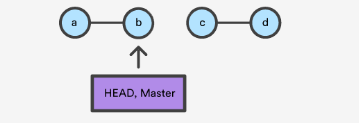
To accurately view the state of the Staging Index we must utilize a lesser known Git command git ls-files. The git ls-files command is essentially a debug utility for inspecting the state of the Staging Index tree.

**Commit History**

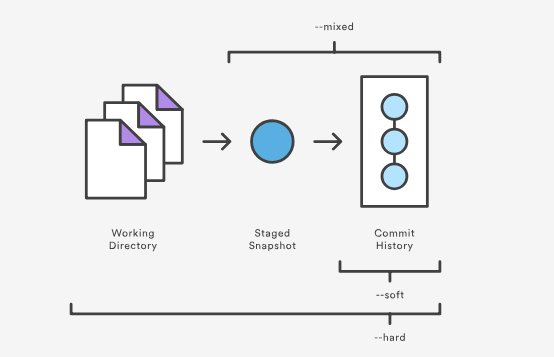
Commit History. The git commit command adds changes to a permanent snapshot that lives in the Commit History. This snapshot also includes the state of the Staging Index at the time of commit.

### 2) Reset options

git reset, moves both the HEAD and branch refs to the specified commit.



In addition to updating the commit ref pointers, git reset will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments --soft, --mixed, and --hard direct how to modify the Staging Index, and Working Directory trees.



--hard

The Commit History ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory are reset to match that of the specified commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

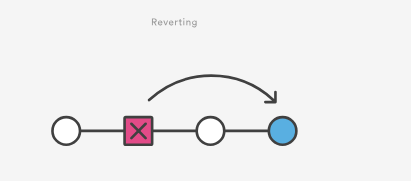
--mixed

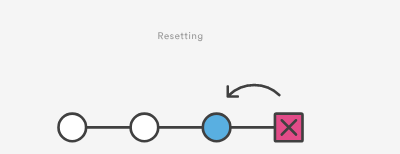
This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory.

--soft

When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched.

## Q) Revert VS Reset





Reverting has two important advantages over resetting. First, it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository.

Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backward from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution.

## Q) clean

The git clean command operates on untracked files.

When there is tracked and untracked changes exist, executing the default git clean command may produce a fatal error.

Step 1 is to show what will be deleted by using the -n option:

git clean -n

Step 2 - **beware: this will delete files**:

git clean -f

* To remove directories, run git clean -f -d or git clean -fd
* To remove ignored files, run git clean -f -X or git clean -fX
* To remove ignored and non-ignored files, run git clean -f -x or git clean -fx

## Q) rm

The primary function of git rm is to remove tracked files from the Git index. Additionally, git rm can be used to remove files from both the staging index and the working directory.

The git rm command operates on the current branch only. The file removal is not persisted to the repository history until a new commit is created. This means that the changes here can be "undone" using common Git commands.





# Git Rewrite history

## Q) Git commit --amend

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit.



The --no-edit flag will allow you to make the amendment to your commit without changing its commit message.

**Don’t amend public commits**

Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

## Q) Git rebase

To modify older or multiple commits, you can use git rebase to combine a sequence of commits into a new base commit.

Since your new commits will be replacing the old, it's important to not use git rebase on commits that have been pushed public, or it will appear that your project history disappeared.