**https://git.seveas.net/undoing-all-kinds-of-mistakes.html#undoing-all-kinds-of-mistakes**

Removing unwanted data from git repositories

A very common mistake is to commit sensitive data, build products or otherwise data that should never end up in a git repository. Git has an extremely powerful command to deal with this: git filter-branch. This command is not only extremely powerful, it's also nigh incomprehensible.

# Recovering from a detached head

Like a French revolution, git provides many ways to detach your HEAD.

Git really only stores 2 things: items of data, which it calls objects, and pointers to this data, which it calls refs. Every file, directory and commit is an object, every tag and branch is a ref.

But not all refs are equal. Most refs are a pointer from a name, say 'master', to an object, such as a commit. The exceptions to this rule are symrefs or symbolic refs. These are used very rarely, with one exception: the HEAD ref.

HEAD is a special ref in more ways than one. It by definition always points to the currently checked out commit. Usually not directly though, but as a symref: it points to a branch whose tip commit is currently checked out.

Let's use some git plumbing commands to show this:

$ git rev-parse --symbolic-full-name HEAD

refs/heads/master

$ git rev-parse refs/heads/master

7c3c37ba945276ca872217850ab8ceeb2e7249e5

$ git rev-parse HEAD

7c3c37ba945276ca872217850ab8ceeb2e7249e5

As you can see, HEAD actually points to the 'master' branch, and thus both HEAD and the master branch now point you to commit 7c3c37ba.

If you check out anything that is not a branch (like a tag, or just commit sha1), HEAD will point directly to that commit. It no longer is a symbolic ref to a branch and has become detached. Robespierre would be proud of you.

Let's check out the second-to-last commit, HEAD^ is a nice shorthand to refer to it. For more of these shorthands, see the [gitrevisions](https://git.seveas.net/manpages/gitrevisions.html) manpage.

$ git checkout HEAD^

While checking out a non-branch is the only way to detach your HEAD, you don't always do this checkout yourself, so you may end up in a detached HEAD state without knowing it.

* When bisecting, HEAD is always detached
* During a rebase HEAD is detached
* Submodules are almost always in a detached HEAD state

It's easy to forget you're in the middle of a bisect or rebase, and you may end up adding commits in a place where you don't want them. But don't worry, there's always a way out.

If your commit cannot be found in the reflog, for instance because you have removed the reflogs, there is a reasonable chance the commit has been removed as well. But you can try recovering it with git fsck --lost-found. If the commit still exists, it will show up in .git/lost-found/commit.

# The meaning of refs and refspecs

Git's simple objects-and-refs model allows you to be very creative in naming your things and putting them in groups. For instance, all branches have names starting with refs/heads/, tags all start with refs/tags and so on.

Most refs are a pointer from a name to a commit. There are three exceptions to this rule:

* HEAD (discussed next) is usually a symbolic ref pointing to a local branch
* refs/remotes/remote-name/HEAD are symbolic refs that point to remote-tracking branches
* Tags usually point to tag objects or commits, but it's not unheard of for tags to point to blobs or trees.

Symbolic refs do not point to an object, but to another ref

# HEAD

HEAD is a very special ref: it is the currently checked out commit. It can be either a symbolic ref that points to a branch, or a direct pointer to a commit. Unless you're manually updating the HEAD ref with git update-ref or git symbolic-ref, it can have no other values.

$ git update-ref HEAD HEAD^

Another special thing about HEAD is that it is not prefixed with refs/ like most refs, but its name is just HEAD.

To read or update the value of any ref, you must use the git rev-parse, git update-ref and git symbolic-ref command, not read or write those files directly.

# More HEADs

There are a few more HEAD-like refs that don't live in the refs/ hierarchy. Unlike HEAD, these don't have a reflog, and are mostly used by a single tool only.

* ORIG\_HEAD is sometimes created by tools that update HEAD in a drastic way
* CHERRY\_PICK\_HEAD points to the commit you are currently cherry-picking
* BISECT\_HEAD is used by git-bisect in some cases
* SVN2GIT\_HEAD is created by git-svnimport

And there are two refs that are really special in that they can point to multiple objects:

* FETCH\_HEAD contains a list of all refs you last fetched, with the first one in the list marked as usable for merging.
* MERGE\_HEAD contains all the heads you are currently merging into the current branch, which could be more than one.

The existence of the merge, bisect and cherry-pick heads can be used as an indication that such an operation is in progress. git status does this, as does the git extension for the bash prompt.

# Tags

All tags live in refs/tags, both the ones you created locally and the ones you fetched from others. There are two types of tags: lightweight tags which point directly to a commit, tree or blob, and annotated tags which point to a tag object. A tag object contains a tag message (for example "Version 1.0"), a pointer to a commit, tree or blob, and possibly a GPG signature.

Annotated tags should be used for tags you want to share, such as releases. Lightweight tags can be used for simple local bookmarks. It is of course not mandatory to stick to this, but there are tools that have this rule built in, such as git describe, which by default only looks at annotated tags, or the push.followtags configuration variable which also ignores lightweight tags.

Tags should also never change. While branches are used to show progress, and branch heads show the current state, tags are meant to mark a specific point in history. Once created they should never change. In fact, git fetch by default will not fetch any tags that already exist locally, even if the values differ.

# Local branches

Local branches are the place where you add commits. By default git creates a branch named master when you initialize a repository, and most projects stick to that name for their default branch. This is of course not mandatory, for example perl.git doesn't have a master branch, their main branch is called blead, because that's what that branch was called before they moved to git.

For this, git stores this historical information in the reflog, a special log per ref which is only kept for branches and for the HEAD ref.

As you can see every action that changes where the ref points to is stored. You can use this to recover original commits that you accidentally amended, undo rebases, see resets and whatnot. It's a great forensic tool.

But reflogs are not the only thing that sets branches apart from other refs. To help git pull, git push and git merge decide what you mean when you use them without arguments, branches can be configured to know what they should merge with and where they should push to by default. As a more concrete example, when you clone a repository, the default branch is checked out and configured to merge from origin/remote and push to origin.

This configuration means that git pull will fetch from the remote named origin and merge what its refs/heads/master points to, that git push will push the branch to the origin remote and that master@{upstream} can be used to refer to refs/heads/remotes/origin/master.

When creating a branch based on a remote branch (for example: git checkout -b develop origin/develop), a similar configuration is set up for the new branch. Other ways to configure this for a branch are using git branch -u or git push -u.

And that brings us to the last thing that's different about branches: there are more ways to specify a commit relative to a branch than for other refs. If you look at the [gitrevisions](https://git.seveas.net/manpages/gitrevisions.html) manpage, you'll see there are many ways to specify a commit relative to another one, such as HEAD~2 for the leftmost grandparent of HEAD. For every ref except branches and HEAD, you can only use commit tree walking tricks, such as refs/tags/v2.0~4^2~3 (take the v2.0 tag, walk 4 parents back using the first parents, then take the second parent of that merge commit, and walk 3 more parents back from there). But for branches you can say things like master@{upstream} to refer to the branch it would merge from, or master@{8.hours.ago}, which uses the reflog to tell you where master pointed to 8 hours ago.

# Remote-tracking branches

So far we've only talked about local refs, and technically all refs are local. However, some are less local than others. The refs under refs/remotes are all copied from your remote repositories when you clone, fetch or push. Git even configures your repository in such a way that any update to those refs is accepted from the remote, even updates that rewrite the history of those branches.

There is one exception to this rule, and it sometimes causes confusion: branches deleted on the remote are not automatically deleted locally. And because refs are currently still stored as files, this can cause file/directory conflicts for certain ref updates.

To create a local branch based on a remote-tracking branch, you used to have to do two steps:

$ git branch develop refs/remotes/origin/develop

$ git checkout develop

Which could be shortened to

$ git checkout -b develop origin/develop

But more recent git versions allow you to simply say

$ git checkout develop

And if there is no local branch with that name, and exactly one remote that has a branch by that name, git will interpret that as git checkout -b develop some-remote/develop. Git is built for and by lazy people, which leads us nicely into the next section.

# DWIM (Do What I Mean)

We're all lazy and we don't like typing refs/heads or refs/tags all the time. So git allows you to use only the relevant parts of the ref and tries to guess what you mean. When you use the word 'tortoise' as a ref, git will try to find it in the following locations, in this order and stops at the first found match:

* A file in .git, which is really only useful for the HEAD variants which live there.
* The tag refs/tags/tortoise
* The branch refs/heads/tortoise
* The remote refs/remotes/tortoise, which means that the remote-tracking branch refs/remotes/tortoise/shell can be specified as tortoise/shell
* The remote-tracking symbolic ref refs/remotes/tortoise/HEAD

Any other ref, such as the ones mentioned below, will only be found by its full ref name, such as refs/pull/42/head

# Specialty refs

The refs discussed so far are all pretty common. But there are quite a few more refs that are more special cases.

## Stash

git stash uses the refs/stash ref and its reflog to keep track of your stashes. The (ab)use of the reflog is why you refer to stashes as stash@{1} etc

# Refspecs

Now that we know all about refs, there's one last trick to know: the refspec. With refspecs you tell git what to push/fetch where and how to map local refs to remote refs and vice versa.

When you clone a repository git sets up the default refspec, you can see it in .git/config in the repository:

When you clone a repository git sets up the default refspec, you can see it in .git/config in the repository:

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[remote "origin"]

url = https://git.example.com/example.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

Like refs, git's handling of refspecs is very DWIM-heavy. For instance, git push origin master actually maps to git push origin refs/heads/master:refs/heads/master@{upstream}, first mapping master to refs/heads/master and then looking up in the config what it should be pushed to. And if it cannot be found in the config, then it actualy maps to git push origin refs/heads/master:refs/heads/master.

One last thing to mention about refspecs is that pushing an empty source will cause the destination ref to be deleted, which means that git push origin :test will delete the test branch remotely.

# Getting rid of submodules

Posted on Sat 07 November 2015 in [Repository maintenance](https://git.seveas.net/category/repository-maintenance.html)

I'm not a fan of submodules. They do have their place, and they can be used in a good way. But they're cumbersome to use and they're too often used as a poor substitute for properly managing dependencies.

So here's how to delete a submodule from your repository that you've added for the wrong reason, or want to get rid of for any other reason.

* Delete the submodule from .gitmodules. If there are no more submodules left, remove the file completely.
* git add .gitmodules to tell git about the change
* rm -rf path/to/submodule to get rid of the files
* git rm -f --cached path/to/submodule to tell git to get rid of the files. Having to do this in two steps is one of the things that is cumbersome about submodules.
* git commit

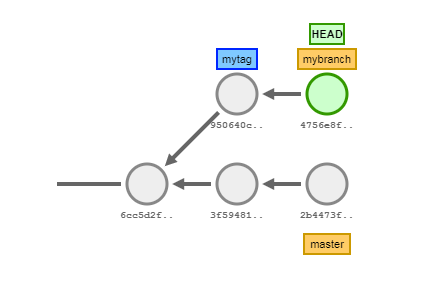
Once you've committed that change, you will need to inspect .git/config and .git/modules for more leftovers that need removing.

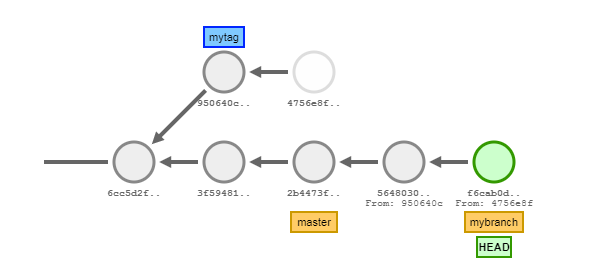
# Rebasing illustrated

## **Other refs**

You also saw that that your rebase only affected the mybranch branch. No other branches or tags were touched, including the master branch, which you rebased onto.

If there are refs that point to commits you are rebasing, they will also not change. Take the following example, that has a tag pointing to a commit on a branch that is about to be rebased.

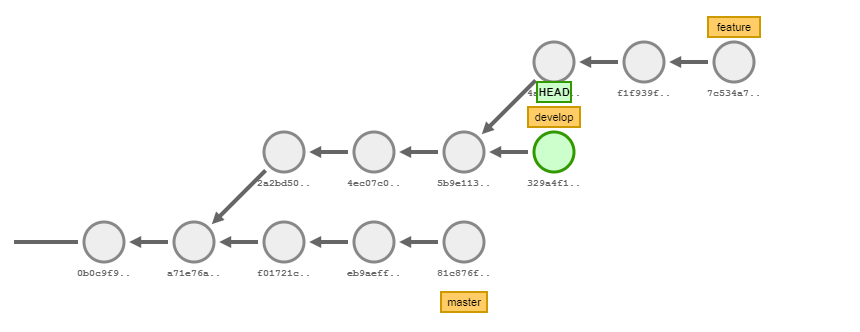


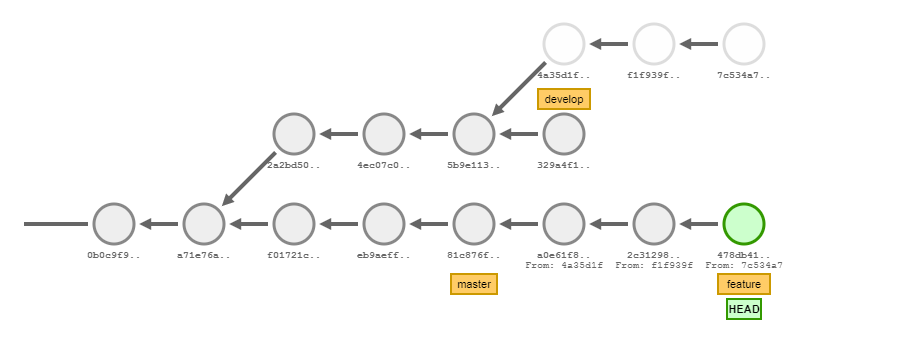


When you git rebase master in this repository, you see that the mytag tag does not move. If you wish the tag to move as well. You will need to do so manually (but beware that moved tags are not automatically fetched by clients).

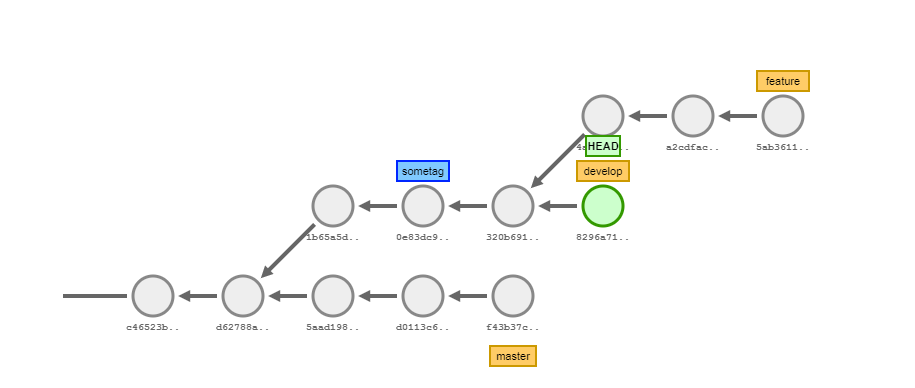
## **Three-argument rebase**

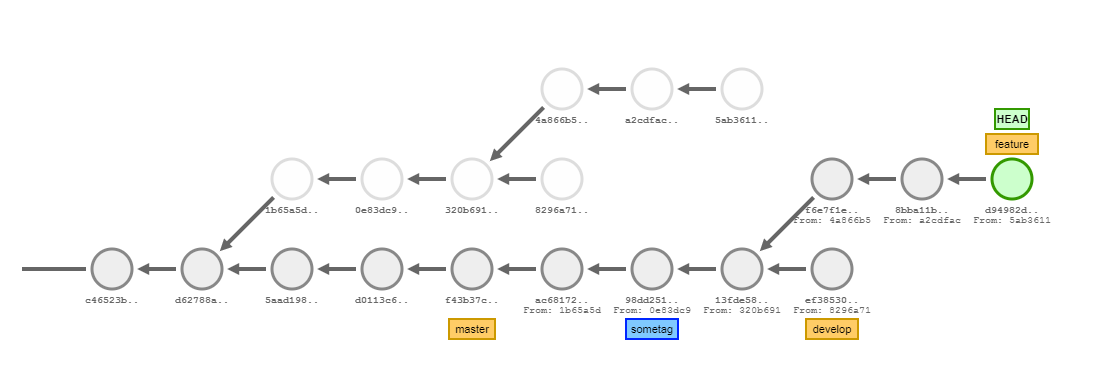
Git rebase can also transplant arbitrary commits to anywhere in the commit tree. In the example below, we have a master branch from which a develop branch has been split. From that develop branch, a feature branch has been split, but that feature branch really should have been based on master. We can tell git rebase to take all commits from the ancestor of develop to feature and transplant them to a new branch on top of master.





## **Rebasing more than one branch**



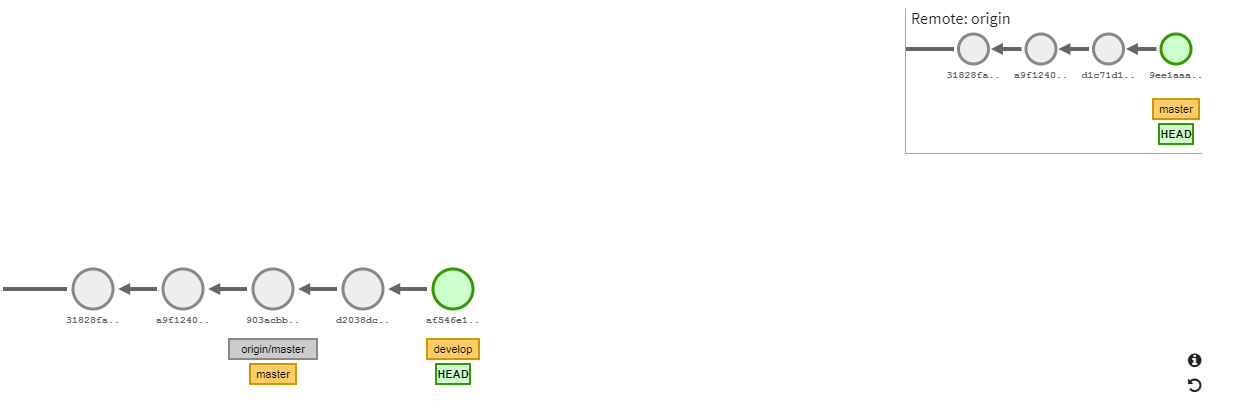


Let's make it a bit more difficult: let's rebase everything onto master, while keeping the layout intact. Let's do the easy one first. We've checked out the develop branch and git rebase master. Now we do git checkout feature, but we cannot just rebase it, as that would duplicate all commits that were common between develop and feature. So we need to carefully rebase just the commits we want, the last three, and we need to attach it to the parent of develop. This turns into git rebase --onto develop~1 HEAD~3.

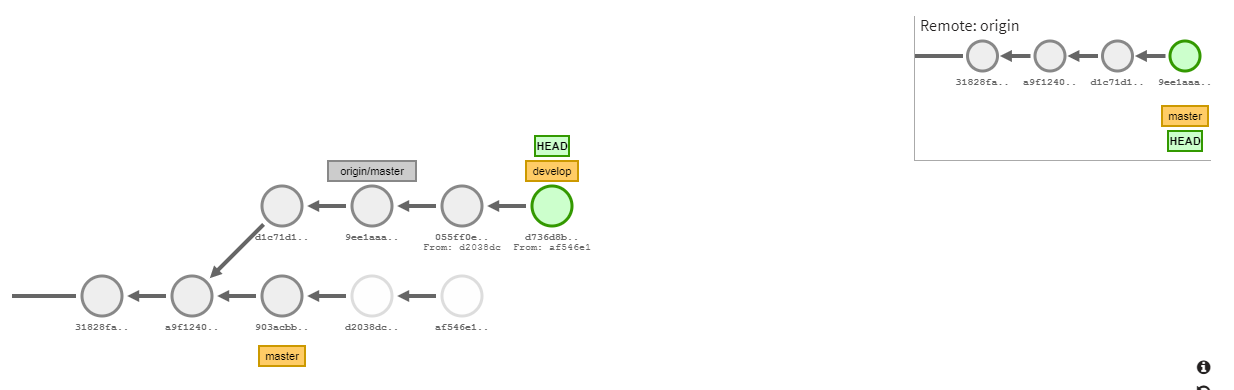
Lastly, we want to move the sometag tag to the grandparent of the new develop branch. This can be done with git tag -f sometag develop~2

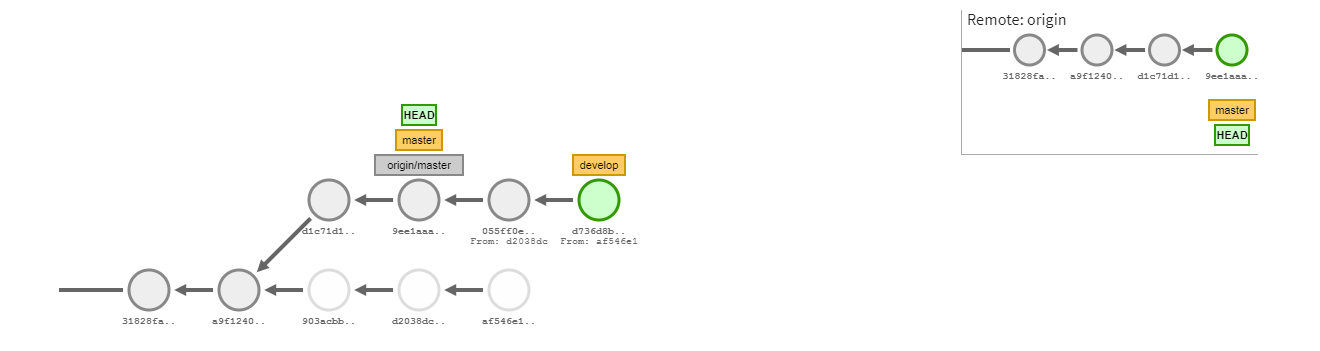
## **Recovering from an upstream rebase**

As I warned above, changing published history causes work for people who based new work on that history. But how much work? Let's take a simple example. As you can see in the graph below, the local origin/master ref points to a commit that no longer exists remotely: remote has changed history, possibly by rebasing.









So let's see what we need to do. First we git fetch the commits into our repository. As you can see, we now have an extra commit (currently pointed to by master) that we should drop. This means we need some surgery on the master and develop branches.

We must fix branches right-to-left, or topological newest to oldest, so we can use a three-argument rebase as above. So first we transplant the develop branch on top of the new origin/master: git rebase --onto origin/master master develop.

And then we git checkout master so we can fix it. Since we didn't have local changes to master, we can just git reset --hard origin/master to move the ref without copying any commits.

That seems simple enough, right? Well... this example is almost trivial (one branch, two commits) and still takes manual inspection and careful use of git to fix. Imagine having a few dozen branches and commits based on work that was rebased, it can be quite a bit of work. Moreover, because any rebase can cause conflicts, the work may not be limited to just moving commits, no some may need to be modified or entirely rewritten. So on behalf of all your collaborators, think twice before rebasing history somebody may have based their work on.

## **Interactive rebase**

All the examples so far show git's default non-interactive rebase. The only time you need to do more than a rebase invocation, is when you have a conflict. This all works fine if all you do is moving commits around, but since you're rewriting history anyway, it is also a good time to groom your history.

Together with git add -p, git rebase -i makes a very clean commit history possible even if your way of working isn't quite as clean. For example, I often work on a few things at the same time, creating many small commits and fix-ups with git add -p as I go. When time comes to publish my work, I use git rebase -i to combine smaller commits into logical units, reorder commits to make sense and maybe even create multiple branches for multiple pull requests.

The example below is one of these cleanup sessions. It's what you can see when you do git rebase -i origin/master to groom all your unpushed commits. In this case I wasn't too messy, but I still wanted to combine the third and fourth commit and fix a typo in commit #5's commit message

pick f351964 Python 3.4 compatibility

# Previewing a merge result

Sometimes you'll want to preview what the result of a merge would be, either to see if it makes sense or to check for merge conflicts. Unfortunately there is not really a way to do this, but the following tricks come close:

git diff other

This will show you the diff between your branch and another one. The downside is that changes made on your branch will also be shown, but in reverse. So it's not very useful.

git diff ...other

This will show the changes on branch *other* since that branch and the one you're on started diverging. The pitfall is that if any changes were also applied to your branch (e.g. with cherry-pick), they'll still show up. Merge conflicts will also not be detected.

Fortunately, git also exposes all its innards to the user. One of these innards is the merge-tree It takes three arguments that should resolve to trees. To preview the effect of merging a branch named develop into the current branch, you could run:

git merge-tree $(git merge-base HEAD develop) HEAD develop

As it's a plumbing command, it does not guess what you mean, you have to be explicit. It also doesn't colorize the output or use your pager, so the full command would be:

git merge-tree $(git merge-base HEAD develop) HEAD develop | colordiff | $(git var GIT\_PAGER)

So really, the best way to preview a merge is to simply try the merge and discard the result. Make sure your working tree is clean and simply merge:

git merge --no-commit other

git diff

This will show exactly what the change will be and will also show all merge conflicts. When you're done inspecting the result, simply discard the changes:

git reset --hard

# Undoing all kinds of mistakes

Let's start with some definitions where git is very precise, but people are not. When asking questions about fixing your repository, it really helps if you use these terms correctly.

#### working directory

The working directory or worktree is the directory that holds both your .git directory and the checked out files.

#### commit

A commit is a full snapshot of all files in your repository. Git does not store diffs or patches, only full snapshots of files. How git manages not to require a very large amount of disk space for your repository is a subject for another article. For now it's just important to remember that all versions of all files ever committed can be retrieved by git.

#### branch, tag and ref

Usually when people refer to a branch, they mean a set of commits that follow each other. This is not wrong per se, but in the world of git, the meaning of branch is more subtle. A branch label, say 'master', is really nothing more than a name that points to a commit (called the tip of that branch). Branch names are not recorded in commits, and once merged they can be deleted without losing commits.

Such a label is called a ref (from reference), and git has [many of them](https://git.seveas.net/the-meaning-of-refs-and-refspecs.html). The most important ones are heads (branches), which are refs that move, and tags, which are refs that don't move.

#### reflog

For refs that move (branches and HEAD), git keeps a log of when they moved and why. So for every commit, reset, merge and all other actions that move heads around, git tracks before and after states. Even when you change history and commits become unreachable, the reflog has your back. And because git's garbage collection also does not delete things that are still in the reflog, you can even undo things that would be very destructive otherwise. Git really doesn't like losing committed data.

#### commit-ish

Many git commands take a commit identifier as argument. While each commit has it's own unique identifier, the sha1, a commit can usually be referred to in many ways: any ref that points to it, the commit tree walking tricks with ^ and ~ etc. 'commit-ish' means any of the ways you can refer to a commit.

#### index

The index or staging area is a feature that is unique to git and is part of what makes git so powerful at commit grooming and refining. The staging area, as its name implies is a staging area where you prepare the next commit. It is in essence a simple list of (filename, sha1) pairs that tell git which data objects should be part of the next commit.

When you git add a file, git actually already adds the file to the object database and adds the sha1 of the file to the index. This is what makes git add -p possible, but also why you have to git add the file after every change.

#### revert

As a noun, it means a commit that is the inverse of another commit, effectively undoing the changes of that commit. As a verb it means to create such a commit. This is the most misused word when talking about undoing changes, so please only use revert if you actually mean either of these two things.

#### reset

Reset can affect the working tree, the index and the commit graph. So it can mean three things, or a combination of two or three of those things. Have I told you yet that git can be confusing?

* When talking about the commit graph, to reset a branch means to point a branch label to another commit, in the context of undoing changes usually an older commit. This makes git forget that commits newer than the commit you reset to have ever been part of that branch.
* Reset can also manipulate the index (reset --hard, reset --mixed, reset -p). This does the inverse of git add, making the index resemble the last commit and not the worktree.
* And finally, reset can undo changes in the worktree (reset --hard).

As a verb, unfortunately it can also mean all of these things. So when talking about a reset, it's vital to say exactly which command you mean.

#### checkout

To check out something means to update the index and working tree with contents from a commit and update the HEAD pointer. The usual invocation of git checkout branchname makes the index and worktree match the tip of the branch and also updates HEAD to point to that tip.

Checkout can also be used to grab only parts of the contents of a commit. In this mode it does not update HEAD. And finally, because git users are lazy, git checkout can also be used to create new branches and check them out at the same time, this is what the -b option does.

#### merge

A merge commit is a commit with more than one parent. Nothing more, nothing less.

To merge means to create a merge commit, merging two or more branches into one. When merging, you will often need to resolve conflicts between these branches.

#### rebase

Rebasing commits copies them to another place in the commit graph. See the [rebasing illustrated](https://git.seveas.net/rebasing-illustrated.html) article for more info on rebase.

# Fixing up uncommitted changes

## Getting rid of all local uncommitted changes

Your friend in this case is git reset --hard which resets the index and the worktree to the state of the last commit.

And if you also want to get rid of untracked files, git clean -di (or its more destructive options, -f and -x) will help you clean up even more.

## Undoing selected local changes

While it's fun to tableflip all your changes away, usually you only want to undo some of your local changes while preserving the rest. If you've already git added the changes, first do a git reset --mixed of the files you want to change to make git forget that you added some changes to the index.

If you want to undo all changes to a certain file, you can simply check the file out again: git checkout -- path/to/file. This also works to 'undelete' a tracked file that you deleted

To only undo some changes to a file, you can still use checkout, but now with the -p flag: git checkout -p -- path/to/file. Like git add -p, it will show each change and ask you what to do with it.

## Undoing staged changes

If you've git added a change, or an entire new file, you can simply git reset filename to undo the adding, without touching any history or your worktree. If you don't want to undo the adding, but want to add more changes to the same file, simply git add them and git will update the index.

## Moving changes to a different branch

Another common issue is finding out you're on the wrong branch and wanting to move your changes to that other branch. If you're lucky, you can simply check out that branch (git checkout branchname if the branch already exists, git checkout -b branchname for a new one). However, if your changes conflict with that branch, you can first git stash your changes, do the checkout and git stash apply, followed by the normal conflict resolution.

I don't like git stash though, so I take a different approach. Which is not actually that different from what git stash does, except with a whole lot less magic and no abuse of the reflog.

First I tag where I'm currently at so I can easily go back. git tag backup. Then I git commit my changes in one or more commits. If there are also changes, I do not want to commit, I'll reset them out of the way. Once that's done, I'll git checkout develop to go to the other branch. I then git cherry-pick backup..master to cherry-pick the new commits onto that branch, solving any merge conflicts that may arise. Then I git checkout master and git reset --hard backup to point master to where it should be. Now we can git tag -d backup and everything is squeaky clean again.

## Recovering uncommitted files after reset

After working with git for a while, most people know that once a file has been committed, git will not easily lose it. What many people do not know is that even just git add is enough to make git remember the version of the file you are adding, even when you make more changes and do another git add. And even when you git reset --hard before committing!

The trick is that git add actually already creates a git object for you and puts its sha1 in the index. When you add again, or when you reset, that blob becomes a so-called dangling blob and git gc will eventually clean it up. But until it has done so, git fsck will find it and tell you the sha1's of all dangling blobs. You can then use git show to recover them, or use git fsck --lost-found to recover them all at once.

# Fixing up committed changes without rewriting history

Once a change has been committed, there are two general ways of undoing the change: rewriting history, making it look like the change never happened. Or creating changes that invert your change. While it's perfectly safe to change history you have never pushed, or to clean up/alter history that has not yet been merged in main branches, things become more complicated when changing for example the master branch of a popular project after pushing it to a central repository, as others may have based new work on it.

If you change published history that other people have based their work on, they also need to alter their histories. Please be aware of this when altering such history. To help those people, we start with fixes that do not require any modification of the commit history.

## Undoing an older commit

To make a commit that inverts all the changes of another commit, you use git revert. For example, to revert the second to last commit in the graph above, you could do git revert HEAD^.

And since a revert is just a simple commit, it can also be reverted, making the changes appear again. This can be useful if you only had to revert changes temporarily while preparing for them to work. In the graph above, git revert HEAD would do the trick.

## Reverting many commits

You can revert many commits in a single command. For example, should you decide that everything between version 0.1 and 0.2 was actually a big mistake, you cangit revert v0.1..v0.2

## Reverting to a specific commit

If you wish to make the next commit look exactly like another commit, you can of course revert until you reach its state. But that may be tricky, or even impossible if that commit is not a direct ancestor of the current HEAD.

But fear not, git is here to help you out. Remember that git does not track changes, instead each commit is a full snapshot of your files. So let's not try to undo changes made, but just git checkout commit-ish -- .. Your tree now looks exactly like the commit you specified, and you can commit it.

There's just one caveat: if there are files in your current commit that are not in the other commit, they will be kept in their current state. So a more complete version of this recipe is: git rm -rf :/ && git checkout commit-ish -- :/

If the commit you want to use as the source of truth is on another branch (and if it isn't, you can simply create a branch) can also trick git merge into doing this. By using the 'ours' merge strategy, it will make a merge commit that has multiple parents, but instead of merging the contents of those commits and their merge base, it simply discards the contents of the other commits and keeps the contents of the current branch.

So if you want to make master look like exactly like develop, that would look as follows:

$ git checkout develop

$ git merge -s ours master

$ git checkout master

$ git merge --ff-only develop

If the commit you want to revert to is not at the tip of a branch, you can simply create a temporary branch:

$ git checkout -b temp-branch 03406c86

$ git merge -s ours master

$ git checkout master

$ git merge --ff-only temp-branch

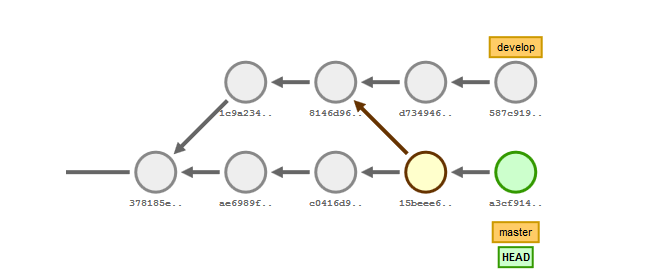
## Reverting a single file

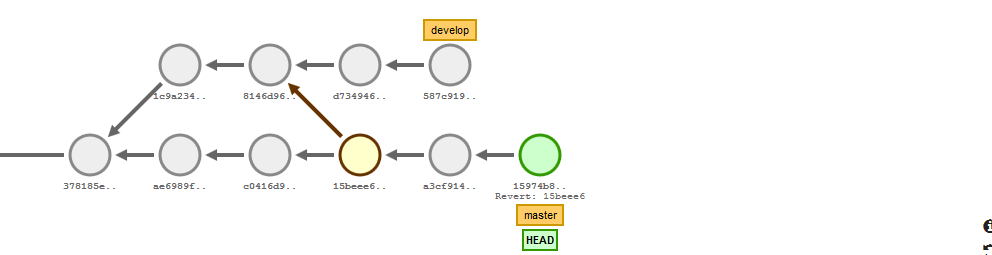
The above recipes are all very useful if you want to revert entire commits. But what if you just want to revert parts of it? To revert the edits to a single file, you can use a combination of diff and apply: git diff commit-ish^..commit-ish -- file | git apply -.

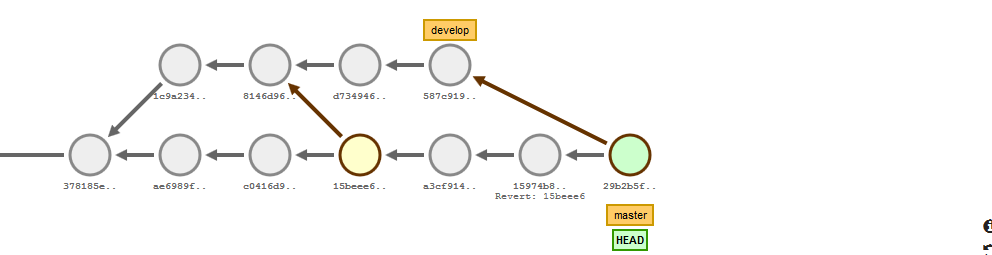
And if you want to make a file look the way it looked in another commit, you can simply check the file out: git checkout commitsh -- file. Use checkout -p to decide hunk-by-hunk whether to retain your current version or use the other version.

## Reverting a merge

Every commit can be reverted, even a merge commit. But reverting a merge commit has one really big downside, which I will illustrate with the graph below. There are 2 branches: master and develop, and develop got merged into master. After this both master and develop have received new commits.







When you git revert HEAD^, git does not undo the merge, but only its effect. So all changes from the develop branch disappear. If you now git merge develop again, they also do not come back, only the changes from the last two commits on the develop branch are applied!

Why is this? Well, when git does a merge, it does a 3-way merge of the content of the current branch, the branch merged in and their common ancestor. For the second merge, the grandparent of the tip of the 'develop' branch is now that common ancestor. So all git sees is that in the current branch a bunch of changes were made, it does not see that these are undoing older commits. It also does not see those older commits, as it does not look further back than the merge base.

So all in all, reverting a merge is not always a good idea. If you still really want to make that merge go away and do not mind rewriting history, there is another recipe for you further below.

## Stop tracking a file

If you want a file to no longer exist, you git rm it. This deletes it from disk and adds the deletion to the index, ready for the next commit. But if you do want to keep it on disk, just not in the repository, you can git rm --cached it, this only stages the deletion but leaves the file untouched.

# Rewriting history to make mistakes disappear

While some people consider it a thoughtcrime to even think about changing history, sometimes you really need to be Winston and make sure things have never happened. Whether you've committed passwords or simply want to clean up before merging, git has you covered.

Before you go all minitrue (ok... that's enough 1984 puns), please do think about the people you are collaborating with in the repository you are manipulating. While it's perfectly safe to alter history you have never pushed, or to clean up/modify pull requests that have not yet been merged, things become more complicated when changing for example the master branch of a popular project after pushing it to a central repository, as others may have based new work on it.

If you change published history that other people have based their work on, they also need to alter their histories. This can be a complicated, error-prone task and you should really avoid forcing others to do so.

## Changing the latest commit

The latest commit is the easiest to change. Just make more changes and git commit --amend. Of course this doesn't actually change the commit, but creates a new one and moves the refs for HEAD and the current branch there.

## Changing an older commit

You can then use the interactive rebase tool to squash these changes into the existing commit. To do this interactive rebase, first use git log to find the sha1 of the commit you wish to change

This is called the worksheet and with it, you can tell rebase exactly what to do. In this case we want to squash the last commit into the first one, so we move the commit and change pick to squash.

Save the worksheet, close your editor and git rebase will do its magic. If you do git log -p, you will see that your commit is now gone, its effects having been moved to the commit where they should have been.

## Making the latest commit or commits disappear

Making commits disappear is easy. git reset --hard HEAD^ makes the last commit go away. git reset --hard HEAD~5 does the same for the last 5 commits. Both also make the changes disappear from your index and worktree. If you do want to keep the changes in your worktree, for instance because you like the changes bit the commits were all messy and you want to redo them using git add -p, don't use hard resets, but git reset --soft.

Hard resets also work really well to undo merges that shouldn't have happened. If you git pull and notice that it does a merge you did not expect, you can do a hard reset to make the merge disappear (and then think about how to actually integrate your changes).

## Making an older commit disappear

As was the case for changing a commit, making older commits go away is slightly trickier, but not much. Again do a git rebase -i to the parent of the commit you want to eradicate. In the instruction sheet, you simply delete the lines corresponding to commits that should go away, and git will make it happen.

## Moving changes to a different branch

As we saw earlier in this article, git doesn't frown upon wanting to move changes to a different branch. We saw how to do this for uncommitted changes, but for committed changes it is really not that much different.

Start with checking out the branch that the changes should have been on. Then cherry-pick the commits that you want to have on this branch. Now go back to the branch they should not have been on and use the recipes above to make the commits disappear. Either a hard reset or an interactive rebase, depending on where the commits are in your history.

## Making (parts of) files disappear from all of history

The recipes above work great for removing or changing single commits, but what if you want to remove a file from all of history? Or committed a password 20 commits ago and want to eradicate it? There are two ways of doing this: git filter-branch, which is black magic on steroids that deserves its own article, or the BFG repo cleaner, which is kinda black magic but much more usable.

The BFG also deserves its own article, and [already has one](https://git.seveas.net/removing-unwanted-data-from-git-repositories.html)! Go read that article for more information about this kind of scrubbing.

# Undoing a rebase, reset or other rewriting

All this rebasing and reseting lets you fix up a lot of things. But what if you mess up while doing so? How do you go back to history that has been deleted? Once again, git has got you covered. As explained early on in this article, git keeps a log of everything you do to refs that change, this includes rewriting the history. So even after a rebase, git reflog knows what you were up to and can help you recover from even more mistakes. As long as a commit is in the reflog, or reachable from a commit in the reflog, git will not delete it during garbage collection and you have yet another safety net in case of mistakes.

# Describing the relationship between commits

As you may have heard by now, git stores its commits and other data in a directed acyclic graph of objects. What this means is that each commit is recorded as a piece of data containing an identifier (a sha1), a pointer to a tree object (another sha1), a log message, author and committer info, and most importantly for this article: information about its parents. A commit can have zero parents (root commit), one parent (regular old commit), or more than one parent (a merge commit).

Using this information about parents, you can describe each commit in relationship to its parents, and using some of git's plumbing information you can use this information for many purposes.

## Commit-ish and git rev-parse

Many git commands, such as git show and git checkout accept a "commit-ish", something that looks like a commit, as argument to specify a commit to act on. A commit-ish can be the sha1 of a commit, a branch, tag or other ref that points to that commit, or some of the things in this article.

When experimenting with these things, the git rev-parse command is incredibly useful, as it can tell you whether you actually have a commit-ish or just some random string:

$ git rev-parse --verify HEAD

## git describe

These exact paths through the commit tree are incredibly useful, but can be a bit unwieldy, sometimes you just need a general indication of how big the 'distance' is between two commits. A prime example of this is in build systems that use git information to create version numbers.

$ git describe

v2.8.0-rc3-12-g047057b

$ git rev-list --count v2.8.0-rc3..

12

$ git rev-parse --short HEAD

047057b

So what does git describe do? It walks the commit history backwards to find the nearest annotated tag, in the case above that would be v2.8.0-rc3. It then appends the number of commits that have been added since that tag and an abbreviation of the exact sha1 of the commit you're looking at. That way you do uniquely identify the commit, but still put it in relation to the latest released version. And you can even feed the output back into git:

$ git rev-parse v2.8.0-rc3-12-g047057b

# Git and configuration files

Git also utilizes this Directed Acyclic Graph structure for content storage. Git is essentially a content-addressable filesystem made up of objects that form a hierarchy which mirrors the content’s filesystem tree. Git has three main primitive types it uses to represent content for a repository: trees, blobs, and commits. All content is essentially stored as either tree or blob objects. A blob is a file stored in the repository and a tree object references either subtrees or blobs. You can think of the blob as the file contents while the trees are like directories. A commit object, on the other hand, has three main attributes. It points to a tree which represents a top-level snapshot of the project at the time of the commit. It also contains references to the commits that came directly before it, a field for author of the commit and, optionally, a commit message.

All of these object primitives are referenced by a 40-digit SHA hash. Two identical objects will have the same hash and different objects will have different hashes. By using the SHA hash as a reference identity, Git can calculate diffing efficiently. In order to safeguard against data corruption, one can recalculate an object’s hash to easily identify corruption or data loss.

Git also uses a DAG to track the history of changes to the content. As stated above, each commit object contains metadata about its ancestors where a commit can have any number of parent commits. Git’s usage of DAGs to store content and keep track of commit and merge histories allows it to maintain full branching capability as the history of a file is linked all the way back up its directory structure to the root directory and a commit object.

Git uses a different merge strategy when the commit of the branch you are on isn’t a direct ancestor of the branch you are merging in, meaning your development history diverged. In this case, Git uses the “recursive” strategy and performs a three-way merge. Git creates a new snapshot of the file state and a new merge commit object that points to the snapshot. This merge commit object now has two parents, pointing to the commit objects at the heads of both of the branches being merged together. Git’s usage of a nonlinear content storage and commit history system allows it to seamlessly merge two branches of a project together.

### Distribution and Initialization

Git handles content and history distribution of projects among collaborators using the distributed model, where users can work offline and make commits on their local repository. Every contributor has a copy of the Git repository where they can work offline, make changes, commit their changes, and (optionally) pull in new changes from a remote repository to stay up to date. When a collaborator is ready to share their changes, they can push them to a publicly accessible repository for other collaborators to access. Once the public repository verifies that the commit can apply to the branch it was pushed to, the same objects that were created and stored on the local repository are created for the public repository and the repository is updated for all collaborators to access.

In order to initialize a local Git repository, you run the command “git init”. This creates a newly initialized repository on your local filesystem, creating a .git directory inside of your current working directory. The .git directory is a subdirectory of the root “working directory” and functions as the actual local repository, containing various config files, the object database, reference pointers for branches, and other scripts that can be run at various points in the projects lifecycle. Another important file is created once you make some changes to files, the Git index, located in .git/index. The Git index file is the staging area between the working directory and the local repository, staging specific changes within one or more files to be committed.

ntroducing the magic controlled by a hidden folder: .git/

In every git repository, you’ll see something like this

$ tree .git/  
.git/  
├── HEAD  
├── config  
├── description  
├── hooks  
│ ├── applypatch-msg.sample  
│ ├── commit-msg.sample  
│ ├── post-update.sample  
│ ├── pre-applypatch.sample  
│ ├── pre-commit.sample  
│ ├── pre-push.sample  
│ ├── pre-rebase.sample  
│ ├── pre-receive.sample  
│ ├── prepare-commit-msg.sample  
│ └── update.sample  
├── info  
│ └── exclude  
├── objects  
│ ├── info  
│ └── pack  
└── refs  
 ├── heads  
 └── tags

8 directories, 14 files

This is how Git controls and manages your entire project. We will go into all the important bits, one by one.

Git consists of 3 parts: the object store, the index and the working directory.

#### The Object Store

This is how Git stores everything internally. For every file in your project that you add, Git generates a hash for the file and stores the file under that hash. For example, if I now create a helloworld file and do git add helloworld (which is telling Git to add the file called helloworld to the git object store), I get something like this:

A new object has been generated! For those interested in going under the hood, Git internally uses the [hash-object](https://git-scm.com/docs/git-hash-object) command like so:

$ git hash-object helloworld

a0423896973644771497bdc03eb99d5281615b51

Yes, it’s the same hash we see under the objects folder. Why the subdirectory with the first two characters of the hash? **It makes searching faster.**

Then, Git creates an object with the name as the above hash, compresses our given file and stores it there. Hence, you can also actually see the contents of the object!

$ git cat-file a0423896973644771497bdc03eb99d5281615b51 -p  
hello world!

This is all under the hood. You’d never use [cat-file](https://git-scm.com/docs/git-cat-file) in day to day adds. You’ll simply add, and let Git handle the rest.

That’s our first Git command, done and dusted.

**git add creates a hash, compresses the file and adds the compressed object to the object store.**

#### The working directory

As the name suggests, this is where you work. All files you create and edit are in the working directory. I created a new file, byeworld and ran git status:

#### The Index

This is the core of Git. Also known as the staging area. The index stores the mapping of files to the objects in the object store. This is where the commits come in. The best way to see this is to test it out!

As you can guess, the first object is the commit metadata: who did what and why, with a tree. The second object, is the actual tree. If you understand unix file system, you’ll know exactly what this is.

The tree in Git corresponds to the Git file system. Everything is either a tree (directory) or a blob (file) and with each commit, Git stores the tree information as well, to tell itself: this is how the working directory should look at this point. Note that the tree points to a specific object of each file it contains (the hash).

It’s time to talk about **branches**! Our first commit added some other stuff to .git/ as well. Our interest is now in .git/refs/heads/master:

$ cat .git/refs/heads/master   
a39b9fdd624c35eee08a36077f411e009da68c2f

Here’s what you need to know about branches:

A branch in Git is a lightweight movable pointer to one of these commits. The default branch name in Git is master.

Git works with objects — compressed versions of files you’re asking Git to track.

Each object has an ID (a hash generated by Git based on contents of the file).

Every time you add a file, Git adds a new object to the object store. **This is exactly why you can’t deal with very large files in Git** — it stores the entire file each time you add changes, not the diff (contrary to popular belief).

Every commit creates 2 objects:

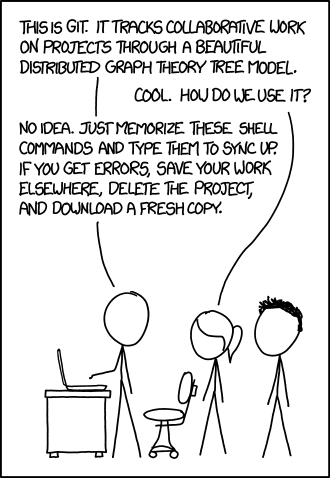
1. **The tree**: An ID for the tree, which acts exactly like a unix directory: it points to other trees (directories) or blobs(files): This builds up the entire directory structure based on the objects present at that time. Blobs are represented by the current objects created by add.
2. **The commit metadata**: An ID for the commit, who made the commit, a tree that represents the commit, commit message and parent commit. Forms a linked list structure linking commits together.

Branches are pointers to commit metadata objects, all stored in .git/refs/heads

That’s all for the understanding behind the scenes! [In the next part](https://medium.freecodecamp.org/now-that-youre-not-afraid-of-git-anymore-here-s-how-to-leverage-what-you-know-11e710c7f37b), we’ll go through some of the Git actions that give people nightmares:

reset, merge, pull, push, fetch and how they modify the internal structure in .git/.





When a merge fails, here’s what git does: It modifies the file with the merge to show you exactly what it can’t decide about.

$ cat helloworld   
hello world!

<<<<<<< HEAD  
Master World  
=======  
Middle World  
>>>>>>> the-middle

The <<<<< HEAD part is ours (the base branch) and the >>>>>> the-middle part is theirs (the branch merging into the base branch).

You can simply edit the file to remove the extra stuff added by git, and choose what should go into helloworld finally. There are some tools and editor integrations to make this easier, but I think knowing how it works underneath the hood gives you more confidence when you don’t have your favourite editor lying around.

### Push and Pull

Push and Pull are actions applied on the remote.

Push pushes your changes to the remote. So, we are sending the Index and corresponding Objects from the object-store!

git push <name of remote> <name of branch>

Pull pulls the code from the remote. Exactly as before, we are copying the Index and corresponding Objects from the object-store!

git pull origin master

origin is the default name of the remote. And since master is the default branch, you can see how the command devolves into the simple name we find everywhere: git pull origin master. Now you know better.

<https://adityasridhar.com/posts/what-is-git-and-how-to-use-it>

Git is an **Open Source Distributed Version Control System**. Now that’s a lot of words to define Git.

Let me break it down and explain the wording:

* **Control System:** This basically means that Git is a content tracker. So Git can be used to store content — it is mostly used to store code due to the other features it provides.
* **Version Control System**: The code which is stored in Git keeps changing as more code is added. Also, many developers can add code in parallel. So Version Control System helps in handling this by maintaining a history of what changes have happened. Also, Git provides features like branches and merges, which I will be covering later.
* **Distributed Version Control System**: In Git, every developer’s computer has a local repo which has the entire copy of the code with the change history. Git does not necessarily need a central server to work with. But in most projects, git is used along with a remote repository provided by github, gitlab, bitbucket and so on. Will explain more about local and remote repositories in this post.

**Git itself is not to be confused with github, gitlab, bitbucket and so on.**

A branch is nothing but a pointer to thelatest commit in the Git repository. So currently our master branch is a pointer to the second commit “demo.txt file is modified”.

<https://adityasridhar.com/posts/how-to-become-a-git-expert>

# I made a mistake in my commit. What should I do?

## Scenario 1

Let’s say that you have committed a bunch of files and realised that the commit message you entered is actually not clear. Now you want to change the commit message. In order to do this you can use git commit --amend

git commit --amend -m "New commit message"

## Scenario 2

Let’s say that you wanted to commit six files but, by mistake, you end up committing only five files. You may think that you can create a new commit and add the 6th file to that commit.

There is nothing wrong with this approach. But, to maintain a neat commit history, wouldn’t it be nicer if you could actually somehow add this file to your previous commit itself? This can be done through git commit --amend as well:

git add file6

git commit --amend --no-edit

--no-edit means that the commit message does not change

## Scenario 3

Whenever you do a commit in Git, the commit has an author name and author email tied to it. Generally, when you set up Git for the first time, you set up the author name and email. You don’t need to worry about the author details for every commit.

That said, it’s possible that for a particular project you want to use a different email ID. You need to configure the email id for that project with the command:

git config user.email "your email id"

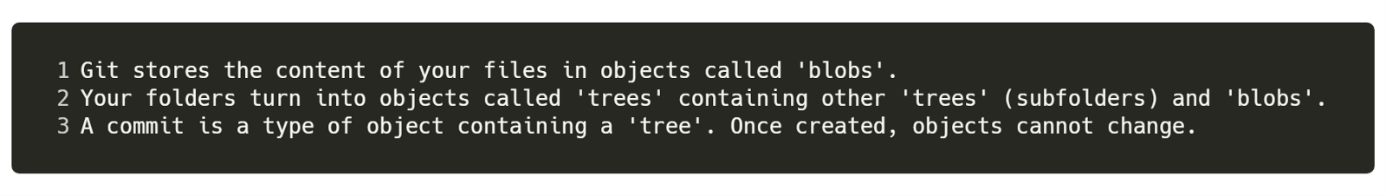
Let’s say that you forgot to configure the email and already did your first commit. Amend can be used to change the author of your previous commit as well. The author of the commit can be changed using the following command:

git commit --amend --author "Author Name <Author Email>

<https://medium.com/@gohberg/the-biggest-misconception-about-git-b2f87d97ed52>

## Git is all about storing differences between files, right?

### The Zen of Git (The 3-Line Version):



That is basically it. 99% of your Git work is just creating those objects and manipulating pointers that reference them

Using this approach, Git doesn’t need to endlessly apply diffs to files to reach some point in your project’s lifetime. **A snapshot of your project can be reconstructed by a simple tree traversal, starting from the commit object. This is why Git is not diff-based, but object based.**

## So Git doesn’t care about diffs at all?

Not exactly. Git tries to be very efficient in storing its objects on disk, since software projects can get bloated very quickly. Git compresses the content of your files (using [zlib](https://en.wikipedia.org/wiki/Zlib)) but that’s not all. What if I change one line in a big file and make a new commit? According to what we learned, that will require creating a new ‘blob’ object, since its contents have changed. That will result in 2 big objects in git’s object database that are very similar.

Git will occasionally look for those incidents and will try to create ‘packfiles’ that contain several objects in one file. In those ‘packfiles’, Git will utilize the difference between two nearly-identical files, storing one version of the file as a whole, and the other as a delta. The version that will be stored intact is the more recent version, because that’s what you’ll most likely be working with. This technique is called ‘delta compression’, and Git tells you about it all the time, especially when you deal with a remote repository. So now when you see this once-cryptic Git message:

DVCS represents history in a Directed Acyclic Graph

Called a “DAG” or simply, a “graph

not possible!

Branches

Exist at the repository level

A branch applies to the entire repository

Unlike most centralized version control tools where branches exist *inside* the repository

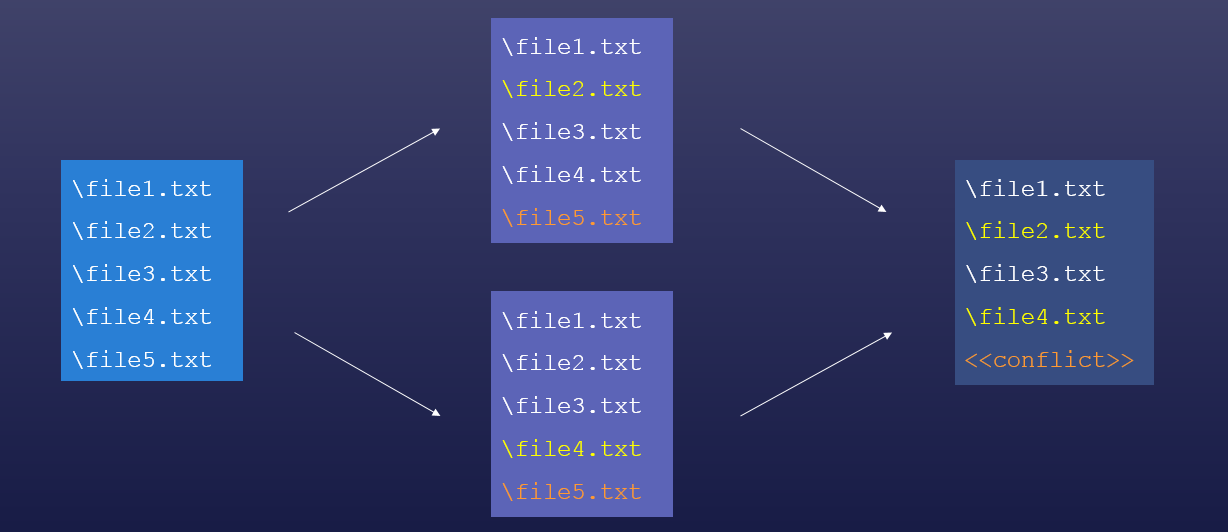
Exceptionally lightweight

Implemented as a pointer to a commit in the graph

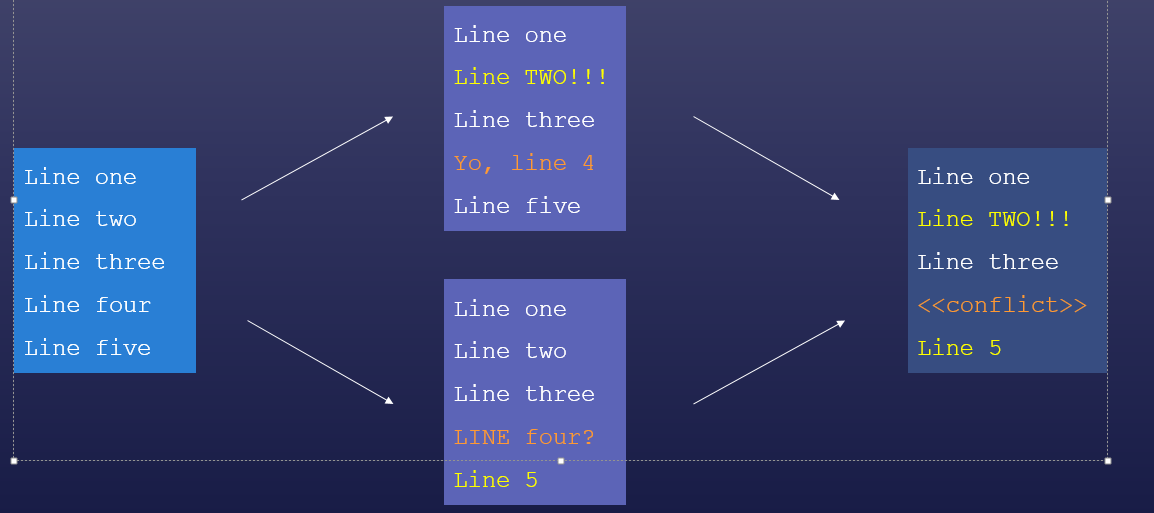
Exist only in the local repository until they’re explicitly shared

Encourages feature branches

Merging two commit trees



Merging two files



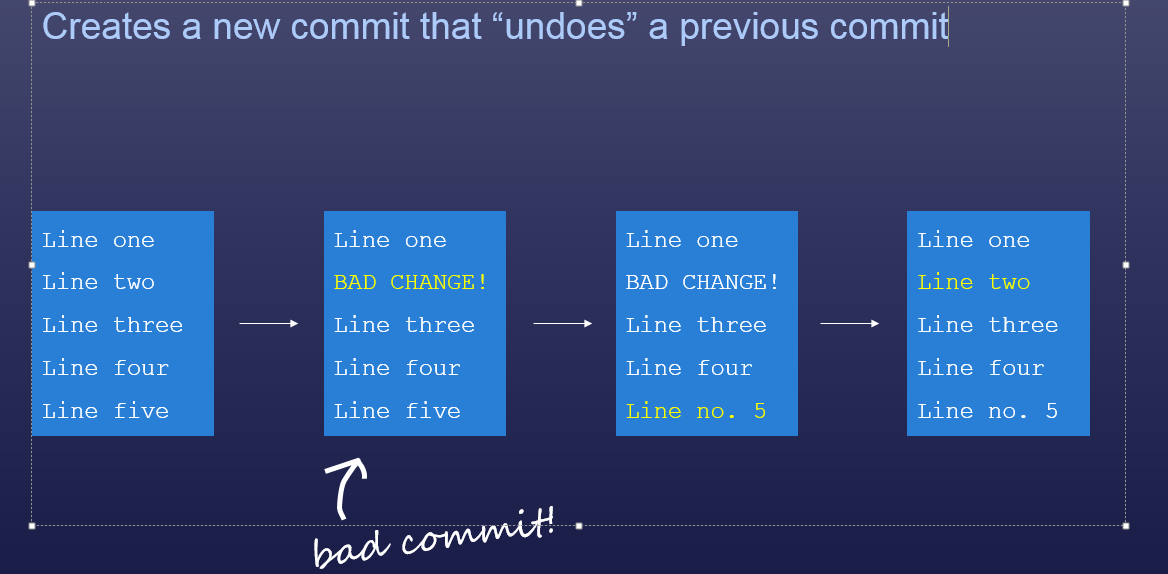
Rebase

Can be used to “clean up “ history before sharing it

Should never be done after sharing

Revert

Creates a new commit that “undoes” a previous commit



Index

Stage

The contents of the repository that will be committed

Add files (git add) to the index to add them to the next commit

Remove files (git rm) from the index to remove them from the next commit

## <https://www.edwardthomson.com/blog/the-git-index-and-recovering-files.html>

## **T**[**he Git index and Recovering Files**](https://www.edwardthomson.com/blog/the-git-index-and-recovering-files.html)

May 9, 2014

When Igit add a file, it will add that file to the object database and place the object information in the staging area. For example, I can create a new file and git add it to my staging area, and examine my staging area using the git ls-files --stage command to see the details (including the object ID) of what's staged:

% echo "new file" > newfile.txt

% git add newfile.txt

% git ls-files --stage

100644 40ee2647744341be918c15f1d0c5e85de4ddc5ed 0 file.txt

100644 3748764a2c3a132adff709b1a6cd75499c11b966 0 newfile.txt

So this file is a normal git blob at this point, and lives inside the git repository, even though I haven't committed these changes yet:

% ls -Fls .git/objects/37

total 1

1 -r--r--r-- 1 ethomson Administ 26 May 9 09:26 48764a2c3a132adff709b1a6cd75499c11b966

If I append some data to this file, it will have both staged changes *and* unstaged changes:

% echo a

Generally, though, I won't know the object ID of the file I've just misplaced, so I would use the git fsck tool, which will do an integrity check of the git repository and show me any objects that are not "reachable", either because they were part of a commit that is not on a branch anymore, or because I git added a file and did not commit it. My newfile.txt is one of these unreachable objects:

% git fsck

Checking object directories: 100% (256/256), done.

dangling blob 3748764a2c3a132adff709b1a6cd75499c11b966

Unfortunately, its filename is not stored in the object database (since identical contents would have the same object regardless of name), so if you have many dangling blobs, you will have to examine each one:

% git show 3748764

new file

Once I determine which dangling blob it is that I want to recover, I can put it back on the filesystem by redirecting git show:

% git show 3748764 > newfile.txt

And the file is recovered!

## [**Introducing git-recover**](https://www.edwardthomson.com/blog/introducing_git_recover.html)

June 15, 2017

Git's index is a "staging area" that will become the next commit. If [you recall from my discussion about how Git works](https://www.youtube.com/watch?v=xWHejdMuIMA), a commit in Git is a snapshot of the entire repository at a single point in time. And the index is also a snapshot: it contains a list of all the files in the repository that will make up the next commit.

You can see this if you look at the index, and Git provides a tool to do just that: git ls-files --stage. When I've just cloned a repository:

% git clone /tmp/foo\_repo .

% git ls-files --stage

100644 6af0abcdfc7822d5f87315af1bb3367484ee3c0c 0 foo.txt

And when I add a new file to this repository, I can inspect the index again, and will see the new file:

% git add bar.txt

% git ls-files --stage

100644 ce013625030ba8dba906f756967f9e9ca394464a 0 bar.txt

100644 6af0abcdfc7822d5f87315af1bb3367484ee3c0c 0 foo.txt

Note that the entry for bar.txt contains the object ID of the file. When you run git add, Git actually adds the file to its object database, and takes the resulting object ID (the SHA-1 hash of the file) and places that in the index.

You can see the file on disk - Git has added it to the repository as a loose object:

So Git has prepared this new file for our commit. But what if we *don't* commit this file? What if, instead, we git rm it? Or if we make some more changes to bar.txt and add those instead?

% echo "different changes" > bar.txt

% git add bar.txt

Now we've overwritten our original changes to bar.txt:

% git ls-files --stage

100644 4a95512212b2f24397fe2df5a2554935bd0a032a 0 bar.txt

100644 6af0abcdfc7822d5f87315af1bb3367484ee3c0c 0 foo.txt

You can see that the object ID for bar.txt had changed - reflecting our new file. But what's happened to the original file we added? Where is object ce01362?

It's still in our object database:

But we never committed it, so this object is not pointed to by any commit in the graph. Nor is it in our index anymore. This unreference blob is "garbage" and - eventually - Git will garbage collect it.

But until it does, we can recover it!

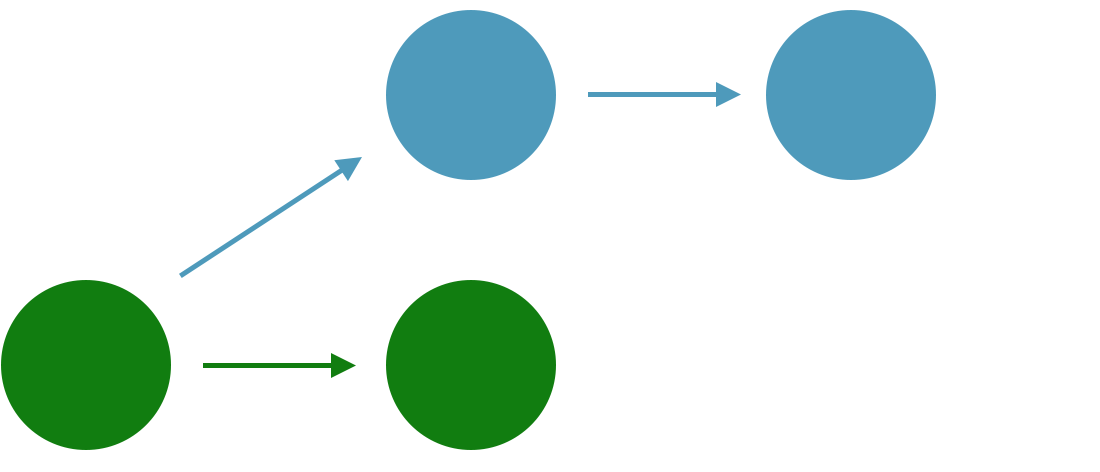
## [**Merge vs Rebase: Do They Produce the Same Result?**](https://www.edwardthomson.com/blog/merge_vs_rebase.html)

https://github.com/ethomson/merge\_vs\_rebase

**Is git-merge guaranteed to produce the same results as git-rebase?**

No!

It's actually not a guarantee; in fact, you can create two branches that merge differently than they rebase. To avoid any spoilers, I've hidden the details in case you want to think about this on your own. 🤔 Click "expand" below to see the details.



Imagine that you have two branches, one is master, and the other is the unimaginatively named branch branch. They're both based off a common ancestor 0d7088f. Further, imagine that your branch has two commits based off that common ancestor:

| **Ancestor 0d7088f** | **branch 3f3ca4f** | **branch 09d3ac4** |
| --- | --- | --- |
| One | One | One |
| Two | 2 | Two |
| Three | Three | Three |
| Four | Four | Four |
| Five | Five | Five |
| Six | Six | Six |
| Seven | Seven | 7 |
| Eight | Eight | Eight |

Finally, imagine that your master branch has a single commit based off the common ancestor:

| **Ancestor 0d7088f** | **master f2e864b** |
| --- | --- |
| One | One |
| Two | 2 |
| Three | Three |
| Four | Four |
| Five | Five |
| Six | Six |
| Seven | Seven |
| Eight | Eight |

What happens when you try to merge or rebase these?

### **Merge**

When Git merges two branches, it only look at the tip commit in each branch, and compares them to their common ancestor. It does not look at any intermediate commits. In the above example, when we merge branch into master, the algorithm looks at the changes made in branch by comparing commit 09d3ac4 to the common ancestor commit 0d7088f. It also looks at the changes made in master by comparing commit f2e864b to the common ancestor commit.

The merge algorithm compares each line[1](https://www.edwardthomson.com/blog/merge_vs_rebase.html#fn:1) in the common ancestor, comparing it to the file in branch and the file in master. If the line is unchanged in all branches, then there's no problem - that line is brought into the merge result. In this example, line 1 in unchanged in both branches, so line 1 of the merge result will be One.

If a line is changed in only one branch, then that change is brought forward into the merge result. In this example, line 7 is changed only in branch. So in the resulting merge, line 7 will have the contents from branch, which is the digit 7. Also, line 2 is changed only in master, so in the merge result it will be the digit 2.

| **Merge Result** |
| --- |
| One |
| 2 |
| Three |
| Four |
| Five |
| Six |
| 7 |
| Eight |

Remember that merge only looks at the tip commits, so comparing the common ancestor to branch, line two appears unchanged, since the ancestor and tip are identical.

### **Rebase**

Rebase works a bit differently - instead of doing a three-way merge between the tip commits on each branch, it tries to replay the commits on one branch onto another. In the above example, if we want to rebase branch onto master, then Git will create a patch for each commit on branch and apply those patches onto master.[2](https://www.edwardthomson.com/blog/merge_vs_rebase.html#fn:2)

When you rebase, Git will switch you to the master branch, checking out f2e864b. Then Git will apply the differences between the common ancestor and the first commit on branch. In this example, the patch between the common ancestor and the branch changes line two from Twoto 2. But that's already the value of the file in master. So there's nothing to do, and the patch for 3f3ca4f applies cleanly.

Then a patch for the second commit on the branch is applied: it changes like two back to the text representation, and changes line seven to a digit. So the rebase result is:

| **Rebase Result** |
| --- |
| One |
| Two |
| Three |
| Four |
| Five |
| Six |
| 7 |
| Eight |

So rebase preserves the changes in the branch while merge preserved the changes in master

### **Conclusion**

Generally these sorts of changes will cause a conflict instead of different results. It was key that in branch we changed the contents of line 2 back to the contents in the common ancestor. That allowed the merge engine to consider that the line in branch was unchanged.

| **Merge Result** | **Rebase Result** |
| --- | --- |
| One | One |
| 2 | Two |
| Three | Three |
| Four | Four |
| Five | Five |
| Six | Six |
| 7 | 7 |
| Eight | Eight |

So… **is this a problem?**

It might seem concerning that this comes up when there was an apparent revert of your changes. Logically, both the branch and the master branches changed line two, but then branch changed it back. So although this seems rather derived, it's not that unlikely.

But whether you prefer a merge workflow or a rebase workflow, you should be careful of your integration and following good development practices:

1. Code review, ideally using [pull requests](https://docs.microsoft.com/en-us/vsts/git/pull-requests), so that your team members have visibility into changes before they're integrated into master.
2. Continuous integration builds and tests, as part of your integration workflow. Ideally, with [build policies](https://docs.microsoft.com/en-us/vsts/git/branch-policies) to ensure that builds succeed and tests pass.

So make sure to do proper code reviews, which keep this an interesting difference instead of an **actual problem** in your workflow.

1. Strictly speaking, the merge engine doesn't actually look at lines, it looks at groups of lines, or "hunks". But it's easier to reason about individual lines for this example. [↩](https://www.edwardthomson.com/blog/merge_vs_rebase.html#fnref:1)
2. By default, rebase will create and then apply patches, but when invoked with git rebase --merge then it will cherry-pick the changes. This uses the merge engine instead of patch application, but in this example, the results are the same.

## [**Git for Windows: Line Endings**](https://www.edwardthomson.com/blog/git_for_windows_line_endings.html)

March 20, 2018

f you’re on a team of Windows developers - or more importantly, on a cross-platform development team - one of the things that comes up *constantly* is line endings. Your line ending settings can be the difference between development productivity and constant frustration.

The key to dealing with line endings is to make sure your configuration is committed to the repository, using .gitattributes. For most people, this is as simple as creating a file named .gitattributes at the root of your repository that contains one line:

\* text=auto

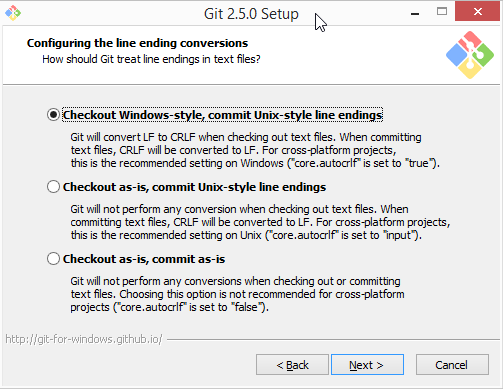
With this set, Windows users will have text files converted from Windows style line endings (\r\n) to Unix style line endings (\n) when they’re added to the repository.

### **Why not core.autocrlf?**

Originally, Git for Windows introduced a different approach for line endings that you may have seen: core.autocrlf. This is a similar approach to the attributes mechanism: the idea is that a Windows user will set a Git configuration option core.autocrlf=true and their line endings will be converted to Unix style line endings when they add files to the repository.

The difference between these two options is subtle, but critical: the .gitattributes is set in the repository, so its shared with everybody. But core.autocrlf is set in the local Git configuration. That means that everybody has to remember to set it, and set it identically.

The first, best option you have to get this right is when you’re installing Git for Windows



You probably want the first option, but you’d be forgiven if you didn’t know that the first time you ran the installer.

The problem with core.autocrlf is that if some people have it set to true and some don’t, you’ll get a mix of line endings in your repository. And that’s not good - because his setting doesn’t *just* tell Git what you want it to do with files going in to your repository. It also tells Git what you’ve already done, and what the line endings look like on the files that are already checked in.

This is why one of the most common symptoms of a line ending configuration problem is seeing “phantom changes”: running git status tells you that you’ve changed a file, but running git diff doesn’t show you any changes. How can that be? Line endings.

### **Phantom Changes**

Imagine that some file is checked in to your repository with Windows-style line endings. For some reason, somebody hadn't set core.autocrlf=true when they added the file. You, on the other hand, being a diligent Git for Windows user, did set that option.

When you run git status, git will look at that file to decide whether you've made any changes to it. When it compares what's on disk to what's in your repository, it will convert the line endings on-disk from Windows-style style to Unix-style in the repository. Since the existing file in the repository had Windows-style line endings, and you expect them to be Unix style, git will determine that the file is different. (It is, byte for byte, different.)

By using .gitattributes, you ensure that these settings exist at the repository level, instead of leaving it up to individual users to understand to configure correctly. This means there’s no opportunity for misconfiguration by an individual user.

Of course, the best time to set this up is at the very moment you create your repository, before you add any files. Doing it after the fact means that you may still have some files added with the wrong configuration.

Over time, these files will be updated as you edit them. You can [try to renormalize files](https://help.github.com/articles/dealing-with-line-endings/), updating the line endings, but doing so will cause annoying merge conflicts for anybody who created a branch before the renormalization.

### **What About Binaries?**

Generally speaking, git is pretty good at detecting whether a file is a binary or not. If it decides that a file is a binary, then it will refuse to convert line endings. But it's still good practice to configure git not to convert line endings for your binary files.

You can remove the text attribute from files that you don't want to have line ending conversions. For example, if you have PNGs in your repository, your .gitattributes might look like this:

\* text=auto

\*.png -text

Of course, there are more advanced settings in your .gitattributes that can be applied. These are especially useful in particular development scenarios. We'll dive deeper into some of those - like using Unity - in the next blog post

## [**Git with Unity**](https://www.edwardthomson.com/blog/git_with_unity.html)

March 22, 2018

## **.gitignore**

The .gitignore file is a metadata file that controls how Git operates on your repository. Files listed in .gitignore will be — like the name implies — ignored. They won't show up in git status and they won't be added to the repository.

It's important to make sure that you .gitignore your build output directories, any cache data and temporary files or directories that your tools make.

https://github.com/github/gitignore

## [Advent Day 24: The Reflog](https://www.edwardthomson.com/blog/advent_day_24_the_reflog.html)

One of the great things about git is that it's very hard to get into a state where you've lost data. Even when you rebase changes away, those changes still exist, and the reflog will always show the changes that you've made throughout history.

## **.gitattributes**

The other metadata file that Git uses is called .gitattributes. You might be familiar with .gitattributes because you use it to control how Git handles line endings for your file.

[You do use .gitattributes to configure your line endings, don't you?](https://www.edwardthomson.com/blog/git_for_windows_line_endings.html)

But .gitattributes is more than just line endings - you can also configure how files are merged when two people change the same file in two different branches.

This means that you can set up Unity's ["Smart Merge"](https://docs.unity3d.com/Manual/SmartMerge.html) functionality. By default, Git is totally unaware of the type of content that you're checking in. If a file is changed in two different branches, it will try to merge the file just by looking at the lines, without understanding them.

But Unity includes a semantic merge tool that understands the actual contents of the scene files, so it can help deal with merging them. You just need to configure .gitattributes to use it.

You can add these lines to your .gitattributes:

\*.anim merge=unityyamlmerge eol=lf

\*.asset merge=unityyamlmerge eol=lf

\*.controller merge=unityyamlmerge eol=lf

\*.mat merge=unityyamlmerge eol=lf

\*.meta merge=unityyamlmerge eol=lf

\*.physicsMaterial merge=unityyamlmerge eol=lf

\*.physicsMaterial2D merge=unityyamlmerge eol=lf

\*.prefab merge=unityyamlmerge eol=lf

\*.unity merge=unityyamlmerge eol=lf

## **Git LFS**

One of the great things about Git is that it's a distributed version control system. That means that you get an entire copy of the repository from the server. That means not just all of the files in the current version of the branch that you're interested in, but all the branches, and all the history that you've ever checked in.

This means that you can work completely disconnected from your server: you can run git log or git blame to analyze the changes that have been made, even if you're on an airplane[1](https://www.edwardthomson.com/blog/git_with_unity.html#fn:1).

But it's problematic when you're checking in large files. If you have assets like images, audio or movies, Git starts to choke. And it's not even the size of the assets themselves as much as the history that's problematic.

If you have a 100K PNG, then that's not so bad. The problem is that you've changed that 100K ping a dozen times. Now you've got 1.2 MB in history that you have to download every time you run git clone. And that's just one file. So it adds up very quicky.

Git LFS helps here: it's the Large File Storage extension to Git.

Instead of storing every copy of these assets in the repository directly, Git LFS stores this data in a separate location, the large file storage area. In the repository, it just checks in a little stub file, the "git-lfs pointer file", that lets Git LFS know where it can get the data when it needs it.

So when you clone the repository, you don't download all those assets, just the tiny (128 byte) git-lfs pointer files. When git needs the files, to write them to your working directory, Git LFS will download them from the server and put them on disk. It's a nice hybrid system between a totally distributed version control system, and a centralized system.

You can [download git-lfs](https://git-lfs.github.com/) - or, if you use [Git for Windows](https://gitforwindows.org/), it's already included. It's easy to set up, you just add some more lines to your .gitattributes file to make sure that your textures and artwork are handled by LFS:

\*.jpg filter=lfs diff=lfs merge=lfs -text

\*.gif filter=lfs diff=lfs merge=lfs -text

\*.png filter=lfs diff=lfs merge=lfs -text

\*.wav filter=lfs diff=lfs merge=lfs -text

\*.ogg filter=lfs diff=lfs merge=lfs -text

\*.mp3 filter=lfs diff=lfs merge=lfs -text

\*.mp4 filter=lfs diff=lfs merge=lfs -text

\*.mov filter=lfs diff=lfs merge=lfs -text

\*.fbx filter=lfs diff=lfs merge=lfs -text

\*.blend filter=lfs diff=lfs merge=lfs -text

\*.obj filter=lfs diff=lfs merge=lfs -text

## [**Advent Day 1: gitattributes for Text Files**](https://www.edwardthomson.com/blog/advent_day_1_gitattributes_for_text_files.html)

A friend of mine asked to have a quick chat about a git problem that he was having and I knew before he even asked what it was going to be about: line endings.

It's always about line endings.

He'd gotten advice to set core.autocrlf; [I've said it before](https://www.edwardthomson.com/blog/git_for_windows_line_endings.html), but it's so important that I want to reiterate: this is always the wrong advice.

The problem with the core.autocrlf configuration option is that it's set on your local repository. This setting isn't checked in anywhere, so you have to rely on everybody who works in your repository to set it, and set it to the same value. If somebody sets core.autocrlf=true and someone else sets core.autocrlf=false then you'll constantly be battling each other, flipping the line endings back and forth from Unix to Windows style each time one of you edits the file.

Instead, you should set the text option in the .gitattributes file at the base of your repository. Don't have one? Create it now. I recommend:

\* text=auto

This will turn on automatic text translation for the files in your repository (at least the ones that aren't binaries); when files are checked in to the repository, they'll be normalized to Unix-stye line endings (\n). When files are checked out to the working directory on a Windows machine, they'll be converted to the native Windows-style line endings (\r\n).

If you want to force a particular type of line endings - perhaps you have Makefiles or shscripts on Windows and these need to have Unix-style line endings to be used. In that case, you can force the files to have Unix style LF endings on all platforms:

Makefile text eol=lf

\*.sh text eol=lf

If you're old school and have been using version control systems before git, you might remember that this concept of having your version control system be opinionated about line endings is rather new. And you may not like it. You may also think it's inefficient to perform this translation (it is). Or you may just like Windows-style line endings.

These are all perfectly reasonable opinions, but you still need to enforce these opinions with your .gitattributes so that everybody in your repository has the same opinions. If you want to not perform line ending translation, set up your .gitattributes with:

\* -text

This will disable all text translations on your text files.

Although git is somewhat opinionated about keeping files in the repository itself with Unix-style line endings, you can configure how this is saved in your working directory, or even disable this behavior altogether. But whatever you choose to do, please do enforce that behavior with a .gitattributes.