https://medium.com/@porteneuve

**Getting solid at Git rebase vs. merge**

### A clean, usable history that makes sense

One of the most important skills of a Git user lies in their ability to maintain a clean, semantic public history of commits. In order to achieve this, they rely on four main tools:

* git commit --amend
* git merge, with or without --no-ff
* git rebase, especially git rebase -i and git rebase -p
* git cherry-pick (which is functionally inseparable from rebase)

These two commands actually have hardly anything in common. They have entirely separate purposes and, indeed, are not supposed to be used for the same reasons at all.

### When should I use merge?

As its name implies, merge performs a merge, a fusion. We want to move the current branch ahead so it incorporates the work of another branch

### Quick summary: core workflow principles

The following principles embody reflexes you should acquire; in the remainder of this article, we’ll dive into the details of the Git commands to achieve these effortlessly.

* **When I merge a temporary local branch…** I make sure it doesn’t show in my history graph by ensuring a fast-forward merge for it, which may require a prior rebase.
* **When I merge a well-known local branch…** I make sure it shows in my history graph, from beginning to end, by ensuring a true merge.
* **When I’m about to push my local work…** I clean up my local history first so I can push something clean and usable.
* **When my push is denied** because of extra work that got pushed in the meantime, **I rebase on the updated remote branch** to avoid polluting the graph with lots of ill-advised micro-merges.

### Merging a branch, the smart way

You should merge a branch only to incorporate the entire feature set it provides. As discussed earlier, the core question you must ask yourself then is **“should this branch remain visible in the graph?”**

When it represents a **well-known body of work** (a task in the project management system, a bugfix linked to an issue or ticket, a story or use case in your agile methodology or project documents, etc.), then it is desirable for it to **remain visible in the long run**, even when the branch name gets deleted.

Otherwise, the branch was just a technical entity and has no reason to keep “existing visually” in the history graph. We will then make sure we use a fast-forward merge for it, which may require a prior rebase of it.

#### Remaining identifiable thanks to a true merge

Let’s assume we have a *feature branch* called *oauth-signin*, and a receiving branch that is *master*.

If *master* has moved on since *oauth-signin* sprouted from it, we’re good. This might be due to other branches getting merged in *master*; or direct commits on it; or someone *cherry-picked* commits in it. At any rate, there is now a divergence between *master* and *oauth-signin*. Git will automatically go for a *true merge* then.

This is what we want, with no particular tweaks to get it.

However, if *master* hasn’t moved since *oauth-signin* sprouted from it, the latter is a *direct descendant* of *master*. Which means that Git will, by default, react to a *merge* by doing a *fast-forward*: it will not create a merge commit, but simply move the *master* branch label to the same commit *oauth-signin* points to. The *oauth-signin* branch becomes transparent: the graph does not isolate its starting point anymore, and once its branch name gets deleted, there won’t be any trace left of it in the graph.

This is not what we want, so we’ll force a *true merge* by using the *--no-ff* option (which obviously stands for *no fast-forward*, not *no Firefox*).

#### Merging transparently by ensuring a fast-forward

This is the opposite situation: our branch should not remain visible in the graph, as it bears no semantic value. We must then ensure the merge will end up doing a *fast-forward*.

Let’s assume we have a comfort, just-for-safety local branch named *quick-fixes*, and *master* is the receiving branch.

If *master* hasn’t moved on since *quick-fixes* sprouted from it, we’re in the clear: by default, Git will perform a *fast-forward*.

On the other hand, if master did move ahead since quick-fixes started, we would get a true merge and our branch would pollute the graph, which we obviously don’t want. Adding the --ff option wouldn’t change anything: this is already the default behavior, and produces no miracles. As for --ff-only, it only refuses true merges, so it will block our merge attempt.

What we need is to tweak quick-fixes so it becomes a direct descendant of master again, making the fast-forward possible. The perfect command for this is indeed rebase. This is exactly what we’re trying to do here: we want to change the base commit of our quick-fixes branch so it is not the old tip of master but its current tip. This will rewrite the history of our quick-fixes branch, but as it is strictly local so far, that doesn’t matter a bit.

Pay special attention to how this scenario plays out:

1. We have a diverging branch to merge transparently, so…
2. We rebase it on our up-to-date receiving branch,
3. We then get back to the receiving end, as rebase changed the current branch,
4. Finally we merge it, the default *fast-forward* being available now.

And *voilà!* Depending on the nature of our branch, we are now assured to always obtain the graph we want

### Rebasing an old branch

Sometimes you start work on a feature branch then don’t have time for it anymore for a long time. When you get back to it, it lacks many fixes and cool new stuff from it base branch, that evolved a lot in the meantime. That bothers you. In such cases, and assuming nobody is working on that branch just now except you, it is perfectly acceptable to rebase it over an up-to-date base branch:

(master) $ git rebase master better-stats

Beware though: if that branch had been pushed to a remote (for backup purposes, for instance), you’ll need to force the next push of it with the -f option, as you just replaced its commit history with a fresh one.

### Cleaning up your local history before pushing

When using Git correctly, we do frequent atomic commits. We also are mindful not to fall into the “subversionian” reflex of commit+push, which reinstates one of the graver faults of centralized source control: every commit is immediately sent to the server.

Indeed, that would deprive us of the flexibility of decentralized source control, which lets us be flexible as long as we haven’t pushed. All our local commits are for now ours alone, so we have complete **freedom to clean them up, rewrite them, cancel them**, right up until the moment we share our work through the remote. Why deny ourselves that flexibility and confort by pushing too often, too soon?

This all yields a rather messy history, difficult to read, understand or leverage by others

But this is no cause for alarm; Git provides a nifty way for you to effortlessly clean up your local history using whatever small touches are necessary:

Rebase provides an interactive mode that will drive it all in a rather sweet, more user-friendly way.

Interactive rebasing is just like regular rebasing, except that instead of following a simple, foreseeable script (“I’ll cherry-pick every commit one by one, just skipping those that end up being duplicates on the new base”), it lets you edit the script beforehand.

In our current situation, **the rebase will not, actually, change the base. It will only rewrite the history since that commit**. In an everyday situation, that branch already exists on your remote, and you wish to clean up the local commits you made since your last sync (usually your last pull).

Let’s say you’re working on an experiment branch. Your command line would then be, typically:

(experiment) $ git rebase -i origin/experiment

Our editor opens up with the following script:

As per usual, Git is nice enough to throw an ad-hoc bit of documentation our way (considering your average developer would rather die than actually browse the doc…). The script at the beginning describes what rebase will eventually do.

By default, it’s a classic rebase: cherry-picking in sequence for every commit in the list. Note this list is chronological (unlike git log, which by default starts from the most recent and works backwards in time).

Like any editor-based Git operation, leaving only blank or commented-out lines will cancel the operation

#### The tricky case of a rebasing pull over a local merge

By default, **a rebase will inline merges**. As we now make sure our merges have clear semantics in our history graph, this inlining is real bummer:

We can avoid this by telling rebase we want to preserve merges: all we need to do is invoke it with --preserve-merges (or the shorthand -p).

1. *git fetch*
2. *git rebase -p origin/feature*

*or* We can interactively git pull --rebase=preserve

# 30 Git CLI options you should know about

### Partial (un)staging with -p

The git add -p command is actually a refinement of git add -i: it pre-selects the **interactive add patch mode**.

git add is **not** about putting a file under version control, but to **stage** an edit, that is, to confirm that edit as a part of the next commit.

### Properly account for renames using -A

To deal with this, we use git add -A, or its longer form git add --all. **This takes everything into account**. When the index is then aware of both changes, it can “realize” it’s a rename (even if some of the content has changed in the file, too), which later allows git log to follow the file across renames, for instance.

**Starting with Git 2.0**, this is the default behavior of git add if you provide a path to it (e.g. git add **.**). By the way, another important change in 2.0: before it, when you did git add -A with no path, it would only work on the current directory and its subfolders, but from 2.0 on, it will work on the entire repository, wherever you are in it.

### Get inside untracked directories for status

when you add a folder to a repo, git status only lists the folder itself as untracked, not its contents. I find that annoying. We can ask status to get inside using -u:

I find this so useful that I set the appropriate configuration variable in my global configuration, so it’s always on:

git config --global status.showUntrackedFiles all

### Produce more useful diffs

The diffs produced by git diff, git log and git show, to name only these, are nice but definitely have room for improvement. Here are three tweaks that are near and dear to my heart:

git diff -w

Or its longer form, git diff --ignore-all-space

You can start by asking diff to only display the line once, using word delimiters, thanks to the --word-diff option. The definition of “word” here is based on whitespace. And by the way, if you dislike these +/- brackets, you can use --word-diff=color to spruce this up. Actually, there’s a shorter form called --color-words (ain’t it cute…).

Anyway, this leaves us with a problem when diff’ing code, as whitespace is seldom the only useful delimiter. Just look at this:



To fix this=>we could result in quite verbose command lines: --word-diff=color --word-diff-regex=. or so. We’ll go for a shorter version:

git diff --color-words=.



If you want to make such an approach systematical (I often do), you can configure diff.wordRegex to the proper value (e.g. .), so any type of word diff (e.g. --color-words) with no argument will use it (an explicit regex in the CLI will of course have priority).

### Fix the latest commit with --amend

An easy way to fix this, as long as you haven’t pushed that commit to your friends, is to put yourself back in the proper situation (e.g. perform the necessary git add, git reset or git rm --cached on the problematic file, perhaps combined with a judicious addition to .gitignore…) then do this:

git commit --amend

This option is actually nothing more than a git reset --soft HEAD^ before the requested commit, but most people do not master reset, so this helps.

Also note that most of the time, the initial commit message was fine. In order to avoid having to re-type that message, or simply having the editor pop up, you can do this:

git commit --amend --no-edit

In your log, only the latest version of the commit is visible. The former version(s) of the commit of course remain in your [reflog](https://www.atlassian.com/en/git/tutorial/rewriting-git-history#!reflog), as the general Git principle holds: “if it’s been committed, it’s virtually impossible to lose.”

### Smart filtering of logs with -S and -G

The git log command is packed with options (100+!), many of which it shares with its close cousin git diff.

A number of these options are there to **filter the log** even before displaying it (which is vastly faster and useful than grep’ing it afterwards): filtering based on dates, paths, branches, authors and committers, commit messages… but also **diff contents**. Specifically, active diff lines.

Diff filtering is extremely useful to **hunt down the origin of some code, especially of a bug**. Too many people think they should use git blame for this,

On the contrary, if you filter **diff contents**, you’ll indeed know **which commit introduced the change you’re interested in**.

If we’re only interested in the presence of a given text in the diff’s active lines (the +/- lines, not the context lines), regardless of why and how it got there, we’ll usually go with -G (this is regex-based, so do remember to escape regex-special characters):

git log -G 'Secure\_?Random' -2 -- path/to/problematic\_file

On the other hand, if we’re specifically looking for diffs that **removed or added the text**, we’ll go with -S, which only returns diffs that **changed the number of occurrences** of the text. By default, -S takes a fixed string, but if you want it to be a regex, just add --pickaxe-regex:

git log -S 'Secure\_?Random' --pickaxe-regex -2 -- path/to/file

If you need your texts, or regexes, to be case-insensitive, add -i. Regexes are always processed as extended-syntax (ERE). Finally, if you want to display diffs on the fly (which can make for heavy display, be warned), add the usual -p (all the more reason to filter on the specific file you’re inspecting).

### Faster branch handling with -b, -v, -vv

git checkout -b creates your new branch on the fly

git checkout -b ticket-12

Of course, nothing stops you from using the 2nd argument to specify the base for the new branch (which defaults to HEAD, as is often the case with the Git CLI).

Extra tip: checkout is smart about one case where -b becomes superfluous: when you want to start working on a remote branch super-feature and don’t have yet a local tracking branch. You can just go:

git checkout super-feature

Git will realize there’s no such local branch, but there is such a branch on the default remote, and will **automatically** do the equivalent of what follows (assuming here your default remote is called origin, which is common):

git checkout -b -t super-feature origin/super-feature

You’re probably used to listing your local branches with a simple git branch:

you can get much more info (SHA, spread with any upstream, first line of the commit message) with –v

You could even go so far as to look up the tracked upstreams with –vv

By default upstreams appear as dark blue, but this sucks on my black background, so I setup color.branch.upstream to cyan…

### Easier help with -w

**Git publishes all its docs** not only in the man format, but also **as HTML**, a type far easier to use and known to all, links included. To use this format, just add the -w option to git help:

git help -w reset

These HTML files are stored locally (installed by Git), so you **don’t even need an Internet access**.

### Better stashing with save and -u

Most people just firing up a git stash first, then a simple git stash apply later on.

Such a message is completely useless, as it **doesn’t say anything about what the work in progress (WIP) actually is**, making it difficult later on to identify what the stash was about.

To fix these to issues, all we need to do is go with the **save subcommand and its -u option** (which includes untracked files), and provide our custom message. For instance:

git stash save -u 'Beginning of Bootstrap 3 refactoring'

Another gotcha is that apply and pop, by default, do not restore the stage. t is indeed saved individually by save, still by default, so why not auto-restore it, as it is an important piece of information?

so **I always explicitly ask it to restore the stage**. So I always use the --index option:

git stash pop --index

### Previous active branch: -

You probably now that in most shells, cd - takes you back to the directory you were in just before the current one (so using this multiple times toggles you between two directories).

As Git versions marched on, various commands have learned this trick: checkout, merge, cherry-pick and lately rebase. Here’s a classic sequence:

(topic) $ git checkout master  
(master) $ git merge -

As Git versions marched on, various commands have learned this trick: checkout, merge, cherry-pick and lately rebase. Here’s a classic sequence:

(topic) $ git checkout master  
(master) $ git merge -

### Cancel the current merge yet preserve previous local edits

**Git doesn’t really need a clean tree** to allow a merge to go ahead: it just needs its working directory to be [in good order](http://git-scm.com/docs/git-merge#_pre_merge_checks), which basically means that files to be changed by the merge should not have local edits, and that you shouldn’t have an ongoing stage (to avoid a multi-topic commit eventually).

So when you find yourself wading through merge conflicts, you may have in your working directory both merge conflicts and local edits that were there before you started the merge.

If you decide to cancel the merge for whatever reasons, it is tempting to go with good ol’ git reset --hard. This would actually be **dangerous**, as it would destroy all local edits **that were there before the merge**, too.

This is why we have git reset --merge (or its more recent syntax: git merge --abort, which is more in line with its rebase cousin): it resets **only changes brought on by the merge**.

(master \*) $ git merge cool-feature  
Auto-merging index.html  
CONFLICT (content): Merge conflict in index.html  
Automatic merge failed; fix conflicts and then commit the result.  
(master \*+) $ git merge --abort  
(master \*) $

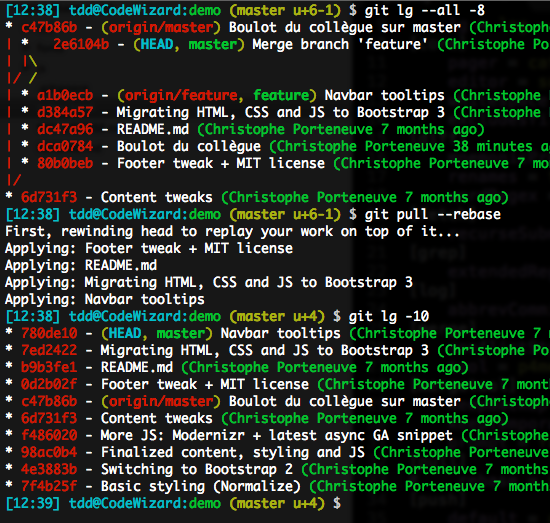
You can even do this **after a successful merge**!

(master \*) $ git merge cool-feature  
Auto-merging index.html  
Merge made by the `recursive` strategy.  
[afbd564] Merged `cool-feature` branch

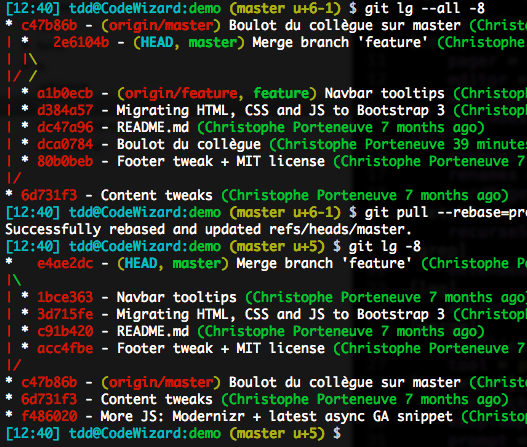
(master \*) $ git reset —merge ORIG\_HEAD  
[ac3489b] Original master tip  
(master \*) $

### Avoid killing a merge when rebasing it

Rebasing is definitely a wonderful [Swiss-army knife](https://medium.com/@porteneuve/getting-solid-at-git-rebase-vs-merge-4fa1a48c53aa), with just one little potential risk: by default, **when rebasing a merge commit, it inlines the merge**. In a nutshell, you run the following risk



To avoid this painful scenario, we can ask rebase to **preserve merges**, using its -p option, or the longer form --preserve-merges. The result will be similar to what follows, although it uses a different CLI context:



### Be a rebase ninja with -i

Talking about interactive rebase, this is indeed where rebase **really shines**, a multi-daily use case being the best-practice reflex of [cleaning up your local history before pushing it](https://medium.com/@porteneuve/getting-solid-at-git-rebase-vs-merge-4fa1a48c53aa#541c), which usually goes git rebase -i @{u}.

### Safely cleaning up with -i and -n

The git clean command is very useful, but **potentially destructive**: it does impact the working directory (WD), so it could **destroy local edits you never committed**, so if you slip up, Git won’t be able to recover your work!

This is probably why, by default, git clean is a no-op, as clean.requireForce defaults to true. You thus would have to git clean -f to start pumping; even then, it would leave directories alone (unless -d) and ignored files too (unless -x).

The good news is, you can **see what your clean would do without any risk**, with the traditional -n (or --dry-run) option that many Git commands feature: it will list files and folders to be removed, but will stop at listing.

And when you do go ahead, you can **gain some confidence** by using -i (the traditional --interactive), that will launch a sort of shell listing candidates for removal, and letting you filter them, confirm each, etc. No more anguish!

### Set the upstream on the fly with -u

So you’re pushing a branch for the first time? You’ll always need to explictly state what the remote is (even if you only have one defined), and what branch you’re pushing (even if it’s the current one), for instance git push origin topic.

However, this simple push **does not set up tracking**: your local configuration does not remember the matching between your local topic branch and its upstream, here the topic branch on the origin remote.

To remedy that, you can at any time re-push with an added -u (or --set-upstream), which will persist that configuration for you, in addition to the push proper. This way you don’t have to specify anything for future pushes and pulls.

git push -u origin topic

Internally, this relies on git branch --set-upstream-to=origin/topic topic, so if you just want to **set this up without pushing just yet**, do that.

As a side reminder, you don’t have to track an homonymous upstream: if names need to differ, you’ll just need to use **the full push syntax**, for instance, to connect a remote christophe-topic branch to your local topic branch:

git push -u origin topic:christophe-topic

This is why the **remote branch deletion syntax** is as follows:

git push origin :old-remote-branch

You’re essentially saying “replace the remote branch old-remote-branch with nothing at all” so… delete it.