**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 \*) $

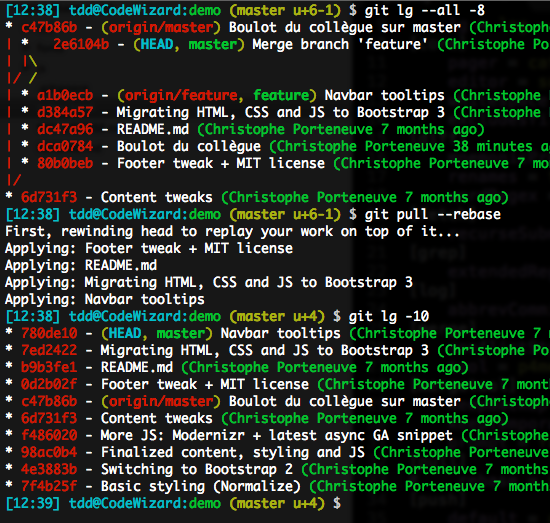
You can even do this **after a successful merge**!

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

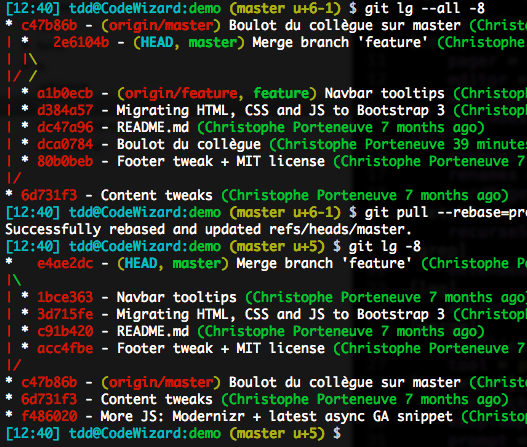
(master \*) $ git reset —merge ORIG\_HEAD  
[ac3489b] Original master tip  
(master \*) $

### 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.

# Mastering Git submodules

Submodules, like subtrees, aim to **reuse code from another repo somewhere inside your own repo’s tree**.

If you have a solid requirement to embed reused code right inside the container code, then you are left with a choice between submodules and subtrees.

In general, [**subtrees are better**](https://medium.com/p/mastering-git-subtrees-943d29a798ec). Most people go with submodules for a few common reasons. Submodules have been around for a good long while, have their own Git command (git submodule), detailed docs. Adding a submodule is very simple (a quick git submodule add), especially compared to adding a subtree. Only later do all the pitfalls and traps come and bite everyone, every day.

### Submodule fundamentals

Submodules rely on **nesting repos**: you have repos within… repos. The module has its own repo, somewhere inside the working directory of its container repo.

In practice, since Git 1.7.8, submodules use a simple .git file with a single gitdir: line mentioning a relative path to the actual repo folder, now located inside the container’s .git/modules.

Be that as it may, the container and the submodule truly act as independent repos: they each have their own history (log), status, diff, etc. **Therefore be mindful of your current directory when reading your prompt or typing commands**: depending on whether you’re inside the submodule or outside of it, the context and impact of your commands differ drastically!

The container and the submodule truly act as independent repos. Finally, **the submodule commit referenced by the container is stored using its SHA1**, not a volatile reference (such as a branch name). Because of this, **a submodule does not automatically upgrade**

Because of this, most of the time a submodule is in **detached head** state inside its containers, as it’s updated by checking out a SHA1 (regardless of whether that commit is the branch tip at that time).

### A plethora of traps

whereas a simple git submodule add suffices to set one up, all contributors to the repo will from then on have to be especially watchful to avoid issues.

we’ll detail what the pitfalls are

* Every time you add a submodule, change its remote’s URL, or change the referenced commit for it, you demand **a manual update by every collaborator**
* Forgetting this explicit update can result in **silent regressions** of the submodule’s referenced commit.
* Commands such as status and diff display **precious little info** about submodules by default.
* Because lifecycles are separate, updating a submodule inside its container project requires **two commits and two pushes**.
* Submodule heads are generally detached, so any local update requires various **preparatory actions** to avoid creating a lost commit.
* Removing a submodule requires several commands and tweaks, some of which are **manual and unassisted**.

#### Adding a submodule

use the git submodule add command. It takes the remote’s URL and a subdirectory in which to “instantiate” the submodule

main (master u=) $ git submodule add ../plugin vendor/plugins/demo

This added some settings in our local configuration:

main (master + u=) $ cat .git/config  
…  
[submodule "vendor/plugins/demo"]  
 url = ../remotes/plugin

And this also staged **two** files:

..new file: .gitmodules  
 new file: vendor/plugins/demo

Huh?! What’s this .gitmodules file? Let’s look at it:

main (master + u=) $ cat .gitmodules  
[submodule "vendor/plugins/demo"]  
 path = vendor/plugins/demo  
 url = ../plugin

This furiously resembles our local config… So why the duplication? Well, precisely because our local config is… local. Our collaborators won’t see it (which is perfectly normal), so they need a mechanism to **get the definitions of all submodules they need to set up in their own repos**. This is what .gitmodules is for; it will be read later by the git submodule init command, as we’ll see in a moment.

Status, like logs and diffs, is limited to the **active repo** (right now, the container), not to submodules, which are nested repos. This is often problematic (it’s super easy to miss a regression when limited to this view), so I recommend you **set up a submodule-aware status once and for all**:

git config --global status.submoduleSummary true

we deal with **two separate repos.** in the submodule’s directory, a .git exists indeed, a single **file** too, not a directory.

Again, since Git 1.7.8, Git does not leave repo directories inside the container’s working directory, but centralizes these in the container’s .git directory (inside .git/modules), and uses a gitdir reference in submodules.

#### Grabbing a repo that uses submodules

$ git clone https://devsource.eua.bvcorp.corp/DV/Mars source

The first thing to notice is that **our submodule is missing** from the working directory; only its base directory is here (BVUtils)

The new repo (source) **is not aware of our submodule yet**: the information for it is nowhere in its local configuration (check its .git/config if you don’t believe me). We’ll need to fill that in, based on what .gitmodules has to say, which is precisely what git submodule init does

Our .git/config is now aware of our submodule. However, we **still haven’t fetched it** from its remote, to say nothing of having it present in our working directory. And yet, our status shows up as clean!.

See, we need to grab the relevant commits manually.

$ git submodule update

In practice, when dealing with submodule-using repos, we usually **group** the two commands (init and update) in one:

$ git submodule update --init

**Git does provide a CLI option** for clone to automatically git submodule update --init recursively right after cloning: the rather aptly-named --recursive option

$ **git clone --recursive remotes/main colleague**

#### Getting an update from the submodule’s remote

On a side note, **I would not recommend using pull** for this kind of update. To properly get the updates in the working directory, this command requires that you’re on the proper active branch, which you usually aren’t (you’re on a detached head most of the time). You’d have to start with a checkout of that branch.

Therefore, I recommend **splitting the process** manually: first git fetch to get all new data from the remote in local cache, then log to verify what you have and checkout on the desired SHA1. In addition to **finer-grained control**, this approach has the added benefit of working regardless of your current state (active branch or detached head).

$ git fetch

$ git log --oneline origin/master -10

$ git checkout 0e90143

Now that our submodule is updated, we can see the result in the **container repo’s status**:

In the “classical” part of the status, we see a new commits change type, which means **the referenced commit changed**. Another possibility (which could be compounded to this one) is new contents, which would mean we made local changes to the submodule’s working directory.

The lower part, enabled by our status.submoduleSummary = true setting earlier on, explicitly states the **introduced commits** (as they use a right-pointing angle bracket >) since our last container commit that had touched the submodule.

In the “terrible default behaviors” family, git diff leaves a lot to be desired. here’s a CLI option that lets us see something more useful

$ git diff **--submodule=log**

Fortunately, there is a matching **configuration setting**:

**git config --global diff.submodule log**

We now only need to perform **the container commit that finalizes our submodule’s update**.

$ git commit -am "Setting submodule on PC2"  
$ git push

### Pulling a submodule-using repo (Mettre à jour un dépôt exploitant des submodules

#### )

This behavior became the default with Git 1.7.5, with the configuration setting fetch.recurseSubmodules now defaulting to on-demand: if a container project gets updates to referenced submodule commits, these submodules get fetched automatically

Still, and this is critical: **Git auto-fetches, but does not auto-update**. Your local cache is up-to-date with the submodule’s remote, but the submodule’s **working directory stuck to its former contents**.

**This is the massive danger: if you don’t explicitly update the submodule’s working directory, your next container commit will regress the submodule**.

Is is therefore **mandatory** that you finalize the update:

$ git submodule update

As long as we’re trying to form generic good habits, the preferred command here would be a git submodule update --init --recursive, in order to auto-init any new submodule, and to recursively update these if need be.

There is another edge case: if the submodule’s remote URL changed since last used (perhaps one of the collaborators changed it in the .gitmodules), you have to manually update your local config to match this. In such a situation, before the git submodule update, you’d need to run a git submodule sync.

**there’s no local configuration setting, or even CLI option for that matter, that can auto-update on pull**. To automate such things, you’d need to use either aliases

git config --global alias.spull '!git pull && git submodule sync --recursive && git submodule update --init --recursive'

#### Updating a submodule in-place in the container

Let’s start by syncing our local state on the remote’s:

$ git checkout master

$ git pull --rebase

Another way to go about this would be, from the container repo, to explicitly sync the submodule’s local branch over its tracked remote branch

colleague (master u=) $ git submodule update --remote --rebase -- vendor/plugins/demo

We can now edit the code, make it work, test it, etc. Once we’re all set, we can then perform **the two commits and the two necessary pushes**

At this point, **the major danger is forgetting to push the submodule**.

You can plainly see how important it is to **remember pushing the submodule too**, ideally before pushing the container.

#### Removing a submodule

#### Temporarily removing a submodule

The first situation is easily handled by git submodule deinit

The submodule is not locally known anymore (it’s gone from .git/config), so its absence from the working directory goes unnoticed.

Any later subcommand of git submodule will **blissfully ignore this submodule** until you init it again, as the submodule won’t even be in local config. Such commands include update, foreach and sync.

On the other hand, **the submodule remains defined** in .gitmodules: an init followed by an update (or a single update --init) will restore it as new:

#### Permanently removing a submodule

a regular git rm will do, just like for any other part of the working directory. In addition to stripping the submodule from the working directory, the command will **update the .gitmodules file** so it does not reference the submodule anymore.

What’s odd though, is that the local config retains submodule information, unlike what happens when you deinit. So, for a comprehensive removal, I recommend you do both, in sequence, so as to end up properly cleaned up (it wouldn’t work after our previous command, because it cleared .gitmodules already):

git submodule deinit path/to/module # ensure local config cleanupgit rm path/to/module # clean WD and .gitmodules

Regardless of your approach, the submodule’s repo remains present in .git/modules/vendor/plugins/demo, but you’re free to kill that whenever you want.

### Best practice recap (TL;DR)

#### Configuration settings

* diff.submodule = log (so you get clearer container diffs when referenced submodule commits changed).
* status.submoduleSummary = true (so git status gets useful again when a referenced submodule commit changed).

#### Adding or cloning

* Initial add: git submodule add <url> <path>
* Initial container clone: git clone --recursive <url> [<path>]

#### Grabbing updates inside a submodule

1. cd path/to/module
2. git fetch
3. git checkout -q <commit-sha1>
4. cd -
5. git commit -am “Updated submodule X to: blah blah

#### Grabbing container updates

1. git pull
2. git submodule sync --recursive
3. git submodule update --init --recursive

#### Updating a submodule inside container code

1. git submodule update --remote --rebase -- path/to/module
2. cd path/to/module
3. Local work, testing, eventually staging
4. git commit -am “Update to central submodule: blah blah”
5. git push
6. cd -
7. git commit -am “Updated submodule X to: blah blah”

#### Permanently removing a submodule ()

1. git submodule deinit path/to/module
2. git rm path/to/module
3. git commit -am “Removed submodule X”

 *git submodule status* is a specific status display for submodules, recursive on request. It tells us what the referenced commits are, whether working directories stray from that, whether submodules are initialized yet or not, and even merge conflicts, if any. Faster than manually checking through your working directories.

 *git submodule summary* lists history ranges between the latest referenced commits and the ones currently checked out. This is what *git status* and *git log* display when submodule logs are enabled.

# Mastering Git subtrees

### Subtree fundamentals

With subtrees, **there are no nested repos**: there’s only one repo, the container, just like a regular codebase. That means just one lifecycle, and **no special tricks to keep in mind for commands and workflows**

### Two approaches:

#### The manual way

Git **does not provide a native subtree command**, unlike what happens for submodules. Subtrees are not so much a feature as they are a **concept**, an **approach** to managing embedded code with Git. They mostly rely on the adequate use of classic porcelain commands (mostly merge and cherry-pick), along with a plumbing one (read-tree).

The manual approach works everywhere, and is **actually quite simple**, but requires a good understanding of the underlying notions so you execute the few procedures properly. We’ll use that as a starting point, because it offers the best degree of control over operations, and leaves us with **complete freedom** in how we manage history (including its graph) and branches…

#### The git subtree contrib script

In June 2012, with version 1.7.11, Git started bundling a **third-party contrib script** name git-subtree.sh in its official distro

### Subtrees, step by step

### Adding a subtree

#### Manually

Let’s start by defining a named remote for our subtree’s central repo, so we don’t clutter our CLIs with its path/URL later:

manually/main (master u=) $ **git remote add plugin ../remotes/plugin**  
manually/main (master u=) $ **git fetch plugin**

We now need to update our index with the contents of this plugin’s master branch, and update our working directory with it; and all this needs to happen in the proper subfolder, too. This is what read-tree does. We’ll use the -u option so the working directory is maintained along with the index.

$ **git read-tree --prefix=vendor/plugins/demo -u plugin/master**

Awesome. Now let’s finalize that with a commit:

$ **git commit -m "Added demo plugin subtree in vendor/plugins/demo"**

#### With git subtree

No need for a manual fetch though: git subtree will do it when necessary. We’ll use its add subcommand:

**$ git subtree add --prefix=vendor/plugins/demo plugin master**

the command **merged our plugin’s history with our container’s**. Let’s verify that with a log:

$ **git log --oneline --graph --decorat**e

you’re not too fond of polluting your container history with the commit details from the subtree… You have a solution in the --squash option git-subtre

$ **git subtree add --prefix=vendor/plugins/demo --squash plugin master**

### Grabbing/updating a repo that uses subtrees

After all, if we were to use submodules, they’d need either a git clone --recursive to grab it, or the bulletproof sequence of git fetch + git submodule sync --recursive + git submodule update --init --recursive for an existing repo.

With subtrees, **they don’t need to do anything special**. The reason is simple: there’s **just one repo**: the container.

To get an up-to-date repo, **you just need a regular clone/pull**.

### Getting an update from the subtree’s remote (Récupérer une mise à jour au sein d’un subtree

### )

#### Manually

It’s actually pretty easy; we just need to update our local cache from the subtree’s remote, then do a subtree merge (using a squash commit, too, to avoid merging histories)

$ **git merge -s subtree --squash plugin/maste**r

$ **git status**

$ **git commit -m "Updated the plugin"**

As always, a squash merge doesn’t finalize the commit; it’s quite handy, too, as we may need to adjust other parts of the container code to work properly with the subtree’s updated code. This way we can make a single, working commit.

#### With git subtree

This is what its pull subcommand is for. Just like the initial add, we’ll reduce the history noise by using --squash. And we need to repeat the entire settings for every call:

$ **git subtree pull --prefix=vendor/plugins/demo --squash plugin master**

### Updating a subtree in-place in the container

Let’s unroll a scenario in which we’ll mix four types of commits:

* Commits **touching only the subtree**, intended for backport (e.g. fixes);
* Commits **touching only container code**;
* Commits **touching both container and subtree code**, the latter part being **intended for backport**;
* Commits **touching only the subtree**, in a container-specific way that is **not to be backported**.

### Backporting to the subtree’s remote

#### Manually

We could create synthetic commits in the middle of nowhere, but that’s fugly. I favor creating a local branch specifically for backporting, and have it track the proper remote for our plugin:

$ **git checkout -b backport-plugin plugin/master**

Now let’s cherry-pick the commits we’re interested in (adding a -x into the mix so the commit message has extra lines detailing the source for each cherry pick)

$ **git cherry-pick -x master~3**

**$ git cherry-pick -x --strategy=subtree master^**

**$ git push**

However, it is prudent to specify --strategy=subtree (-s means something else in cherry-pick) to make sure files outside of the subtree (elsewhere in container code) will get quietly ignored, as would happen for main-file-2 in master^. If you forget this option, Git will refuse to complete the cherry-pick, as it would believe our side (backport-plugin) just removed that file (you’d see a deleted by us conflict). So you’d better use that specific option all the time, just to be on the safe side.

#### With git subtree

Sure, there’s a pretty git subtree push subcommand, but it has a significant drawback: **it backports every single commit that touched the subtree**. You can’t pick the relevant commits. So our last commit, which was container-specific, gets cargoed along… Grmbl. This is not what we want here, but I’ll show you the command anyway:

$ **git subtree push -P vendor/plugins/demo plugin master**

### Removing a subtree

It’s just a directory in your repo. A good ol’ git rm will do, regardless of the approach you used.

$ git rm -r vendor/plugins/demo

$ git commit -m "Removing demo subtree"

### Turning a directory into a subtree

This is the last “fun” use-case: you want to take code that **always was an integral part** of your container codebase, and **extract it for sharing** between multiple codebases.

#### Manually

The idea is to create a special branch for the future subtree, and **filter down its history so it only keeps commits that touched the subdirectory, rewriting the tree root as it goes**.

This sounds like heavy-lifting, but it precisely matches what a mode of the “bulldozer” git filter-branch command does: the --subdirectory-filter option. See for yourself:

$ **git checkout -b split-plugin**  
$ **git filter-branch --subdirectory-filter lib/plugins/myown**

Now we just need to push that to the proper remote:

$ **git remote add myown ../remotes/myown**  
$ **git push -u myown split-plugin:master**

#### With git subtree

There is a split subcommand intended for about the same thing.

**git subtree split -P lib/plugins/myown -b split-plugin**

ou can then repeat the remote-updating commands we had, to publish the final result:

$ **git remote add myown ../remotes/myown**  
$ **git push -u myown split-plugin:master**

# Mastering Git Reset: Commit Alchemy

HEAD is a pointer, a reference to our current position in terms of history.

Technically it’s just a text file stored in .*git/HEAD*:

$ cat .git/HEAD

ref: refs/heads/master

In turn, *.git/refs/heads/master* contains its tip commit’s SHA-1. Such a file then contains the commit’s metadata and tree information, which we can introspect using the plumbing command *git cat-file*:

$ git cat-file -p HEAD

When you peeked into your .git directory, you might have seen a file named *ORIG\_HEAD*. It’s related to *HEAD*, but always contains a raw SHA-1 instead of a named reference.

*ORIG\_HEAD* **backs up the position of *HEAD* before a potentially dangerous operation** (merge, rebase, etc.). This way, should things go awry, Git will be able to come back to the position before that by doing a *git reset --merge ORIG\_HEAD*.

Git manages your work through **3 major local areas**:

1. Your **working directory**
2. The **index, or stage**
3. The (local) **repository**

There are two other areas (the stash, and the remote)



**The working directory**

It is the complete set of directories, subdirectories and files you’re working with for a particular project, at the root of which you normally have your *.git* directory, as a result of having called *git init* there.

**The stage**

This truly is the **staging area** for your next commit: this is where you put snapshots of whatever parts of your ongoing work you’re greenlighting for the next commit.

You add stuff to the stage through the *git add* command.

The index name is most apparent in the name of the technical file that holds its current list of known files and trees: .git/index. You can see what’s in there in many ways, for instance through the plumbing command git ls-files --stage, which displays it as a tree (it is a tree, Git-wise):

$ git ls-files --stage

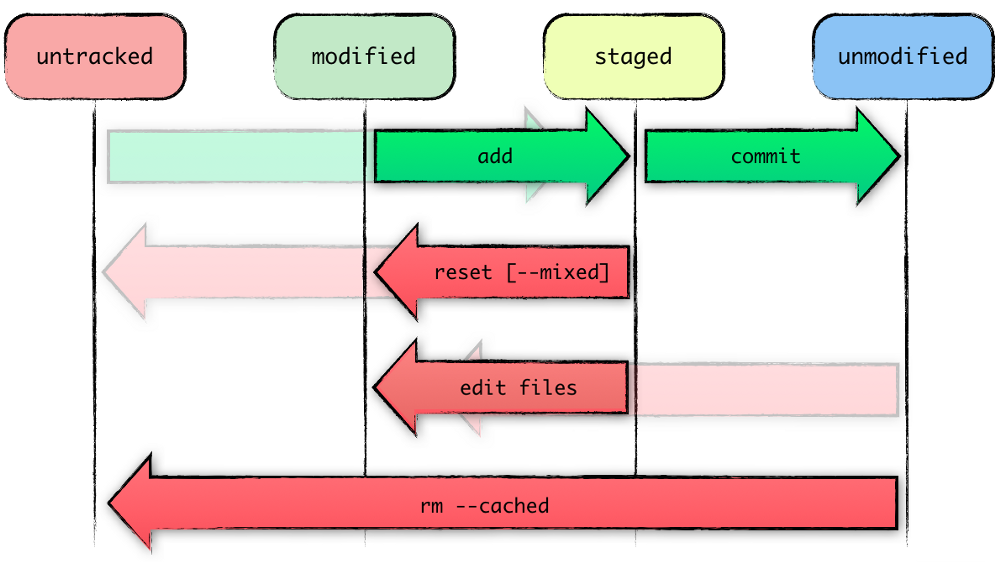
In short, **the stage contains all necessary info for Git to create a commit, including a merge commit**.

**The (local) repository**

This is all the metadata related to your versioned work: commits, references, local change history, configuration…

It’s sort of like an archive room where everything you send is neatly compressed, labeled and stored in a way that makes retrieval as fast as possible whilst still optimizing storage.

Sending stuff in there is what *git commit* does.



### So what about Git reset?

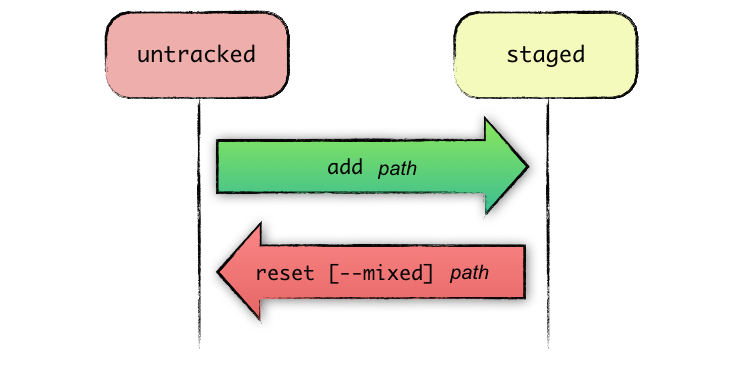
Because we’re in root commit state (there’s no commit yet, hence no HEAD), unstaging would require a git rm --cached instead of a git reset, and git status says as much:

#### Unstaging stuff

We’ve just alluded, twice, to the fact that git reset is great for unstaging stuff you don’t eventually want in your next commit. For instance, we may have lazily typed a git add . and only then realized we hadn’t ignored a sensitive private key file such as private.key. Fear not, we can just unstage that file:

git reset private.key

##### Cas du fichier untracked



##### Cas du fichier existant/unmodified Unstage d’un contenu existant

La commande git reset avec pour argument(s) un ou plusieurs chemins va permettre de sortir les modifications du stage pour les fichiers concernés. La commande git checkout dans ce cas viendra replacer les fichiers concernés dans leur dernier état avant modifications locales (unmodified, pointé par HEAD) dans le répertoire de travail et le stage.

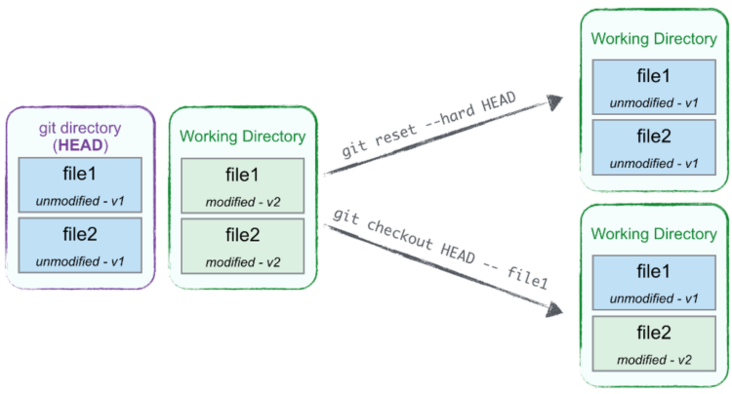
#### Removing local changes from our WD

Sometimes we’ve started with local changes that end up being unsatisfactory, and we’d like to get back to the state we were at in the latest commit. This is what git reset --hard HEAD is for.

**Caution:** any local change to versioned files will be lost for good: be extra careful before you resort to this command. Only new, untracked files will be left alone, as HEAD didn’t know about them

In such a situation though, hard resetting is like bulldozing through your WD, as it will undo every local change to every known file, not just the date file. You’ll probably be safer with a partial checkout, something *git status* also suggests:

git checkout HEAD date.txt



#### Undoing all or part of our latest commit

git reset --soft HEAD~1  
git reset private.key  
echo 'private.key' >> .gitignore  
git add .gitignore  
git commit -m "Adding the public key"

#### Augmenting our latest commit

There are really two subcases here:

* **You forgot to add some files** (a common culprit is resorting to git commit -a but not realizing you’re also depending on new, untracked files that won’t get auto-added this way).
* **You want to tweak the commit message** (those pesky typos…)

The former looks much like our previous situation: we want to get back to our previous stage, tweak it, then re-commit.

git reset --soft HEAD~1  
# Stage the missing files  
git commit -m "…"

Again, --amend to the rescue, saving some keystrokes:

# Stage the missing files  
git commit --amend --no-edit

#### Undoing our latest merge, rebase, cherry-pick…

If you read through the Fundamentals part of this article, you may remember that nifty little backup ref Git maintains for us in many situations: ORIG\_HEAD. It’s available for us in these situations. All we need to do is a merge-mode reset.

git reset --merge ORIG\_HEAD

Now, the merge mode isn’t just for undoing merges. It’s sort of like a hard reset, except it will preserve any local changes we had going that didn’t prevent the operation from happening (for instance, we were editing files untouched by the merge). Better safe than sorry, you know.

#### Restoring a local file from some version

Only checkout lets you do that, through a partial checkout. It’s sort of like a path-specific hard reset, with the important additional distinction that it doesn’t move the HEAD at all.

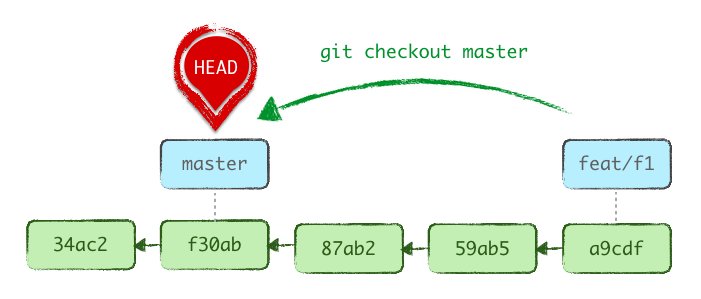
# "Gimme feat/f1’s version of README.md"  
git checkout feat/f1 -- README.md

# "Bring test/ and spec/ back 3 commits, keep the rest"  
git checkout HEAD~3 -- test spec

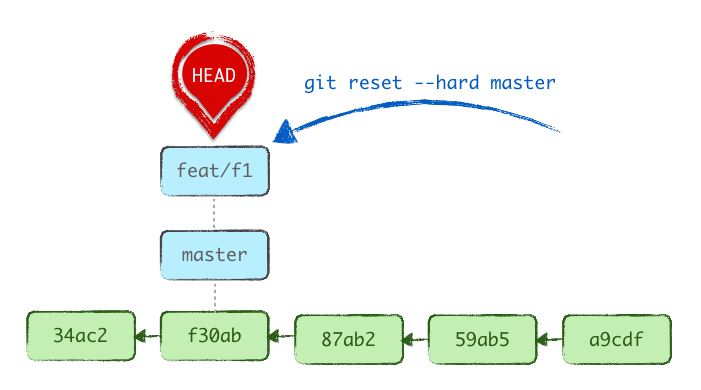
When “moving” towards specific branches, it can be tempting to use *git reset target-branch* instead of *git checkout target-branch*. But there is a fundamental difference:

* *checkout* will switch the *HEAD* so it refers to the target branch, leaving your original branch untouched, and you end up with the target branch becoming the active one.
* *reset* will move your current branch so it matches the target branch, “losing” any divergent commit line there might have been since their common ancestor. It doesn’t change the active branch.

Checking out master switches HEAD to refer to it (the .git/HEAD file stops saying ref: refs/heads/feat/f1 and now says ref: refs/heads/master). We now have a different active branch, and feat/f1 is untouched



On the other hand, resetting HEAD to another branch keeps the current active branch (.git/HEAD is unchanged), but drags that branch label along to the target commit, effectively making the two branches identical.



slides

**INTERACTIVE REBASE AS A PRE-PUSH REFLEX**

▸ Cardinal rule of Git: **commit often, push when needed**

▸ Not pushing early gets you **leeway to clean up** your log

▸ Pre-push reflex:

▸ git log --oneline --graph --decorate @{push}..

▸ git rebase -i

▸ Cleaner logs = more usable logs = **more productivity**

# Une configuration Git aux petits oignons

## Diffs améliorés

Par défaut, toute forme de diff signale les deux versions comme a et b :

Ce n'est pas toujours évident de savoir ce qu'est a et ce qu'est b… Le HEAD ? Le *stage* ? Le *working directory* ? Récemment, Git a fourni un réglage, diff.mnemonicPrefix, qui lorsqu'il est à true va remplacer, lorsque c'est possible, ces initiales anonymes par trois autres :

* c pour Commit (généralement le HEAD)
* i pour Index (le *stage*)
* w pour *Working directory*

Ainsi, un diff classique explicite qu'il est effectué entre le stage (l'index) et le working directory :

Un diff --staged indique bien qu'il est effectué entre le HEAD (un commit) et le stage (l'index) :

Quant à un diff HEAD, qui explicite son commit de référence, il confirme qu'il compare un commit et le working directory :

Un autre aspect un peu dommage de git diff est le comportement par défaut de git diff --word-diff. S'il est bien agréable d'avoir par défaut un mode word diff, qui permet de cibler, au sein de deux lignes sans doute très similaires, la zone de différence (un peu comme le changement de contraste dans des outils de diff graphiques), Git considère par défaut comme « mot » toute séquence de caractères non-whitespace. Dans un bloc de code, ça fait vite long, vu que les délimiteurs et opérateurs seront aussi comptabilisés.

Par exemple, [`@twitterClient.directMessage](mailto:%60@twitterClient.directMessage)(contact,` est considéré comme un mot. Snif.

C'est pourquoi git diff fournit une autre option, --word-diff-regex=…, qui permet de préciser une regex représentant un « mot ». Naturellement, en réduisant cette regex à ., on va tenter la correspondance minimale (jusqu'à un seul caractère). Dans la pratique, quand je fais un diff de niveau mot, je fais toujours comme ça.

C'est là tout l'intérêt du réglage diff.wordRegex : il permet de traiter un simple --word-diff comme un --word-diff-regex=….

## Submodules

**Les submodules sont une mauvaise idée**

divers réglages permettent, au fil des versions, d'ajouter des garde-fous. J'en préconise quelques-uns

* fetch.recurseSubmodules détecte automatiquement lors d'un fetch (et donc d'un pull) si les références des submodules ont bougé, et propose de les récupérer automatiquement si c'est le cas. Attention, ça ne fait pas le submodule update pour autant.
* status.submoduleSummary prend soin de signaler dans git status les submodules dont la référence a bougé, en listant les deltas de commits concernés.

## Regex étendues par défaut

Git propose à de nombreux endroits de recourir aux [**expressions rationnelles**](https://delicious-insights.com/fr/articles/enfin-maitriser-les-expressions-rationnelles/) (regex). Notamment git log --grep et git log -G, pour ne citer qu'eux. Par défaut cependant, seules les BRE (Basic Regular Expressions) sont traitées, c'est-à-dire que la majorité des caractères spéciaux tels que {}, '()', + et ? sont ignorés. Dans la pratique, quand j'utilise une regex, je veux toujours disposer de la **syntaxe étendue** (ERE, Extended Regular Expressions), qui active toutes les syntaxes possibles.

Plutôt que d'avoir alors à ajouter -E à ma ligne de commande, j'active le réglage grep.extendedRegexp dans la configuration globale.

Attention cependant : vos regex ne seront pas insensibles à la casse pour autant. Si vous voulez garantir cela, vous devrez généralement ajouter -i à vos lignes de commande

## Des fusions plus pratiques

Ainsi, Git injecte dans les fichiers textuels des **marqueurs de conflit** classiques. Par exemple :

<body>

<<<<<<< HEAD

<h1>Super site de gros bills</h1>

=======

<h1>Super site de malades</h1>

>>>>>>> feature

</body>

Il est déjà possible d'obtenir davantage d'informations à ce stade, en affichant entre les deux (la branche récipiendaire de la fusion et la branche en cours de fusion) la version du texte dans l'ancêtre commun (le point de divergence). C'est le mode **diff3**, que j'active automatiquement avec le réglage merge.conflictStyle :

<body>

<<<<<<< HEAD

<h1>Super site de gros bills</h1>

||||||| merged common ancestors

<h1>Super site de oufs</h1>

=======

<h1>Super site de malades</h1>

>>>>>>> feature

</body>

Par ailleurs, lorsqu'on recourt à un **outil de résolution de conflits** (un mergetool), celui-ci peut générer des fichiers temporaires ou de sauvegarde (souvent avec des extensions genre .rej, .orig, .local, .tmp, .bak…), qui viennent joyeusement pourrir votre working directory. Même si ces extensions sont protégées contre un ajout par inadvertance à l'aide d'un .gitignore ou équivalent, les fichiers restent là.

Vous pouvez vous assurer que votre outil va nettoyer derrière lui en désactivant les réglages mergetool.keepBackup et mergetool.keepTemporaries.

Pour finir sur les mergetools, lorsqu'on lance git mergetool, il demande par défaut confirmation quant au fichier à manipuler. Afin de vous éviter cette question le plus souvent superflue, désactivez mergetool.prompt

## Des pulls en rebase par défaut

Le seul danger potentiel, c'est qu'un rebase, par défaut, « inline » les fusions. C'est ajustable quand on git rebase manuellement, mais pas dans le cadre d'un pull. Prenons un exemple.

Disons que vous venez de procéder à une fusion légitime sur votre branche locale (par exemple, la branche feature était terminée et vous avez procédé à son rapatriement sur master à l'aide d'un git merge feature).

Le push vous est refusé car il s'avère que le master distant (origin/master) a bougé. Pas le choix, vous faites un pull avant de continuer. Si celui-ci est en mode rebase, vous allez automatiquement rebaser l'historique local récent (les commits depuis votre ancienne référence origin/master, dont votre commit de fusion) sur le nouveau origin/master. En temps normal c'est top, mais sur ce coup ça va inliner la fusion en remplaçant le true merge par un rebase. La branche, jusqu'ici bien visible dans le graphe du log, va disparaître.

Ma recommandation est la suivante :

1. Si vous vous apprêtez à fusionner une branche et que vous avez l'excellente habitude de nettoyer votre historique local avant de *pusher*, faites-le dès maintenant (vous ne pourrez plus le faire après sans *inliner* la fusion).
2. *Pullez* ensuite immédiatement, en mode *rebase*. Ce sera toujours ça comme risque en moins pour après.
3. Effectuez la fusion.
4. *Pushez* immédiatement ; si un petit malin a remis à jour le *remote* entre-temps, au pire, faites un git pull --no-rebase juste pour cette fois. L'historique ne sera pas bien joli, mais bon.