Git and Gitflow tutorial

Release 0.2

Mar 25, 2020

Tutorial

This page is only the README landing page for the repository, head over to https://git-flow-tutorial.readthedocs.io/en/latest/ *for the tutorial itself*

Welcome ! This repository contains a tutorial to learn how to use Git and Git flow to collaborate on code with others. Should you be a total newbie to git, you'll find (almost) everything you need to learn the basics and become autonomous on a small project. You may already be familiar with most git commands, but are not satisfied with the way you use it, because you do not really know how to collaborate effectively, what are the best practices, or you *feel* that you're not making the most of the branching system.

In both cases, this tutorial will help you see Git as a helpful coding partner, rather than that painful bash-thing that your team and teachers make you use without really explaining to you its principles, power and versatility. This is not a **Learn Git in five minutes** tutorial, rather something in between a single page tutorial and a small book.

CHAPTER 1

What you will find here

If you're totally new to Git, read the general introduction ! It is sure rather long, but it explains you what Git is without any technical considerations, so that you can start the tutorial with an already clearer idea of the big picture. I remember the first time I was presented with Git : everyone was talking about *do you commits*, *use branches*, *Git is super powerful*... Since no one really knew what it was for and how to use it the right way, it soon became an evil tool that was only repeatedly breaking my code with awful `<<<<<<< HEAD` lines, and that's it !

So take the time to understand Git without having to think about any terminal window, it will save you tons of time and energy wasted speaking a language that you know the words but don't understand yet.

CHAPTER 2

Contents

The first section is all about the basic commands that you would learn during your first encoutering with Git in any other tutorial. It is crucial since they are the ones that you'll use tons of time and that should become reflexes. Rather than giving you an unordered list of things you need to know, the tutorial is built so as to reproduce the usual steps of code development in a team.

If you've already been using Git for at least one or two months, you can go fast on that one, cherry-pick the subsections that are new to you, or go straight into the second and third sections, about branching models and successful collaboration as a team working with git.

Note that the second part is mostly about the *git flow* workflow. This is only one of the many git workflows available for software development. But even if it is not suited for your particulary needs (anyway you'd need to read all this to the end to know wouldn't you ?), it surely is full of great ideas and advise that you could apply for any workflow, so feel free to read even if you're not interested in git flow itself.

CHAPTER 3

Do code while reading

You can obviously read that tutorial without coding, but you're likely to forget almost everything when you're done. Feel free to use a personal repository to apply stuff you'll read here, or to clone the repository associated with these tutorial to have at hand an exemple of a repository following (most of the time) the ideas and principles you'll hear about here.

There's a little python script in it that makes a good material to train on. You can download and install python here. Do not fear to break things : this repository is protected from anyone but me trying to modify existing things. So please play with the instructions and do break things, for it is the best way to learn, and you'll always be able to delete everything and clone the tutorial again, or simply go back to before you broke something (that's precisely the thing about Git). If you don't know python, just use any type of text file you want to use as food for git.

CHAPTER 4

Contributing

I welcome suggestions through merge requests ! You'll see that the little code used to give examples in the tutorial uses pyux package, which I did too because I love that kind of little but helpful packages. Feel free to check it out. Comments and merge requests are welcome here too.

CHAPTER 5

About me

I'm a data scientist, not a real developer, and not a pro git user at all. But I like git, and I like to explain things, so here it is. I think not being a pro on the matter is a good thing to the purpose of this tutorial : make someone new to git and a little bit lost able to use it in a proper yet simple way. I'll try to keep the explanations user-oriented and not too technical, which you probably don't need.

5.1 A non-technical presentation of Git

5.1.1 Your code's diary

Strictly speaking, Git is a *distributed version control system* (DVCS), which means, a set of tools organised for a clear purpose : give you control over the state your code is at, past and present. That's for *VCS*. We're not interested for now in the *distribution* part, you can read some details about it here !

You can think of Git as a *high-end collective diary for your code*. It enables you to save and comment on your progression at any time, in a very detailed way if you want to, or in a bulky way if that's your preference. Just as you would read past pages of your diary to remember how you felt about those holidays last summer, you can tell Git to bring back an old version of your code, in case your recent work has broken something, for instance, and start over from there.

You can use it to work on ideas, sketch things, and decide later if you want to integrate those ideas to your code and how. You can even truly experiment on crazy ideas without affecting the existing, note things about it, decide if it is worth adding to the existing, show it to people to hear their opinion about it, maybe change some things here and there... The fun part comes with the *collective* in it. When you work alone, it is not that hard to know what has happened and what is happening. As soon as you collaborate with others on the same repository, knowing how to use this diary becomes harder yet critical.

5.1.2 Tools, knowledge and rules

Writing a diary requires several things : the diary itself, one or more pens. It also requires that you know how to write, so you and others can read and understand your work. When you get comfortable with writing, you start to use more

advanced tools : different colors and stylings, bookmarks, etc. Lastly, you implicitely or explicit set some rules : a title starts on the top, write today's date on the top-right corner, do not use more than one page for a single day, etc.

This tutorial will guide through the exact same things applied to **using Git to collaborate on code**. If you are new to it, you first need to get acquainted with the basic tools (I'll say *commands* from now on) that everything will be built upon. This you'll learn in the following section *Your first steps with Git (part I)* and *Your first steps with Git (part II)*.

At this point, you'll be ready to learn how to collaborate with others on a repository. Collaborating – except if you're looking for chaos – requires that you settle on a shared set of rules, meant to be abide by everyone. Those rules are often ideals, that are difficult to follow every time you code. Just remember that they often come from people that have already handled the exact same problems as the ones you'll face one day, and that those rules intend to ease them. So trust them ! We'll present in section *Gitflow branching model*, a workflow that is not the simplest but has proved to work in many situations : *git flow*.

Once you know the basics and the theory of collaboration with git flow, you'll be presented with other Git tools that tremendously help with maintaining a clean and usable history of your code. Those tools need some more attention, since they can erase things and change history : you don't really want to mess up with time ! But with Git, actually you can – sometimes. More on that in *Collaborating with git* section.

5.1.3 Combine Git bash with a cool Git client

To use Git, you enter commands that do stuff : save, comment, go back in time, merge different versions of your code, etc. The first way to do that – the one you *should* start with, is using Git bash or a terminal window (if you're working on Linux or MacOS). Writing instructions is easy there, but it is far less convenient to explore and visualise the history and changes within your code.

To do so, you can use a *graphical user interface* (GUI), also referred to as *Git clients*. These are softwares that allow you not only to visualise and explore the git tree, but also to run most of the git commands without having to type them in a terminal window. You'll often read that it is best to learn how to use git bash before using a GUI, because otherwise you may end up dependant of your GUI : it prevents you from truly knowing how to talk to git or how to handle difficult cases that the GUI is not built for.

GUI are of no help when it comes to using virtual machines : since, in most cases, you communicate with them from a terminal window, you *need* to know raw git commands. I thus totally agree with the idea that you should do the whole tutorial from Git bash or a terminal window. However, I also strongly recommand that, in the meantime, you download a GUI of your choice, *only to visualise what the commands do and the state of the git history*. A git tree can easily be messy and intricate, and Git built-in visualisation commands are not friendly for complex cases.

None of the exercices below require a GUI, but having it along the way will undoubtedly help you with knowing what you're doing and remembering what are the various commands for. A widely use git client is sourcetree. I personnaly use GitKraken, without any strong reason to except that it has a cool name and a dark theme. Both of them are free (at least for open source packages with GitKraken). All screenshots that you'll see here are taken from this repository's tree, visualised with GitKraken. You can find a list of GUI on Git official website.

5.2 Your first steps with Git (part I)

Encountering Git is not an easy task : as any powerful tool, it requires some practice before rendering your coding life easier than it already was. Before helping with collaboration with others, it is first and foremost meant to control and navigate the *history* of your code. That makes it something you also want tonuse on your own, because it helps with keeping track of what you've done before (and you *will* forget it), experimenting on things without risking any bugs thanks to the branching system, and, if you're that serious, even reviewing your own code !

There are a lot of commands, but the vast majority of your usage will revolve around less than a dozen ones. They are the building blocks of everything else, so take the time to know them well, because they will appear *everywhere* in the more advanced sections. Do not force you to remember them : first, you can use the recap of all the commands

mentionned in the *Basic Git commands* page, and also you'll use them so often that soon you won't have to think about them anymore.

We don't cover installing git on your machine here : there are plenty of ressources online that can help you with that, starting with the git website itself. Once git is installed on your computer, find a repository of codes that you can work on freely. If any, you can clone the repository you're now reading from. Open up a terminal window and go to the folder you want to clone the repository into (**Pandoc** has great instructions to do that from a terminal if that too is new to you), then type

```
git clone git@gitlab.com:roamdam/git-flow-tuto.git
```

This will download a folder with name `git-flow-tuto`, and if you go into that directory with `cd git-flow-tuto`, you're likely to see a `(master)` next to the path your terminal window is at. We'll dive into the `master` things later on, for now it is just the sign that the folder you're in is indeed a git repository. If you want to use your own folder, which may already exists, simply go to that folder from the terminal and type `git init`. This will initiate a git repository. You're good to go !

```
Romain@MBP-de-Romain ~
$ cd geek/git-flow-tuto/
Romain@MBP-de-Romain ~/geek/git-flow-tuto (master)
$
```

If you haven't done it yet, I suggest again that you install a GUI to help you visualise what's happening ! Here are again the links for GitKraken *or* Sourcetree

5.2.1 Fundamentals : the commit

What happens when you write to your manager, your team or your teacher to tell them what you've worked on today ? First you may make a list of what you did, to recap the changes. Then you may write the body of a mail describing, in a more or less detailed way, what you did. Finally, you may send the mail. In git, the *commit* is precisely that mail. Here is the definition from the Git glossary

```
A single point in the Git history; the entire history of a project is
represented as a set of interrelated commits
```

In other words, a commit is a bundle of selected changes in your files, with a message describing what those changes are. When you make a commit (or *when you commit*), you tell git to record that bundle of changes as a new entry in the history of your repository. Commits are the bare bone of your history : all that you can do with git – branching, simultaneously maintaining different versions, merging versions, going back in time – revolves around them.

Another way of seeing them is as checkpoints when you develop something. You jotted down a new function, and had it working after some modifications. You want to continue working on it but would like to be able to go back to that working version with ease, in case the modifications do not work out well. This is precisely when you would *commit* your changes, which is recording the state of your function at this time, and be able to come back at it any time you want.

A single file is modified

Say you've added a new function in your code, or just changed something in an existing one, and you want to record that change in the git history. First, you *add* the change to the soon-to-come commit. This is the moment you were writing the body of the follow-up mail to your manager in the previous paragraph. The command is simple :

```
git add myfile
```

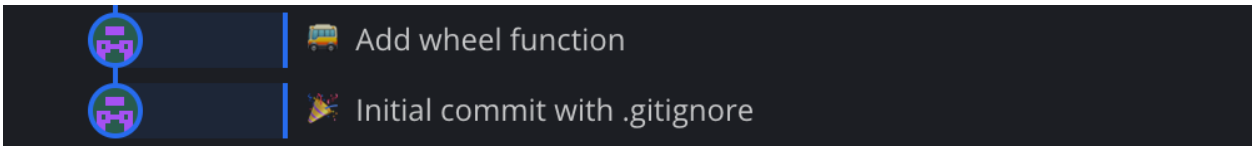
In the exemple below, I change a word in a printed message from the file `main.py`, and I want to record that change. After using `git add` to add the change to the commit, I actually do the commit. Making a commit is done with the command :

```
git commit -m "your commit message here (short, less than 80 chars)"
```

The `-m` option allows you to write the message directly within the command. You can also use `git commit` : it will open a terminal text editor (often Nano or Vim). It can be quite surprising at first, so for your first commits I suggest you to stick to using `-m "your message"` when you commit. Using text editors for commit messages is useful when you want to write long commit descriptions – we'll talk about that later.

```
(venv) romain@romain ~/projets/git-flow-tuto (master)
$ git add main.py
(venv) romain@romain ~/projets/git-flow-tuto (master)
$ git commit -m "Change 'excellent' to 'cool' in printed message"
[master 886fc12] Change 'excellent' to 'cool' in printed message
 1 file changed, 1 insertion(+), 1 deletion(-)
(venv) romain@romain ~/projets/git-flow-tuto (master)
$
```

That's it, your first commit is done. Below is a screenshot of the very first two commits of this repository, visualised with GitKraken. As you can see, a commit is indeed a *point* on your repository's time *line*, with a message that describes it to help remember what happened at that precise moment.



Several files are modified

Obviously, most of the time you want to commit changes to several files in the same commit, because it makes sense to gather thoses changes in the same history point. The first option is to manually add each modified file to the commit with `git add`. Soon enough you won't remember exactly all the files that were modified, so you need to get a list of what has changed. The command for that is

```
git status
```

In the exemple below, I made changes to several files and added a new folder. Suppose I had a coffee in between and I don't remember exactly what files I did change. Using git status allow me to see which files are candidates to be added to the commit, so that I can manually add the ones I want to commit, then commit.

```
(venv) romain@romain ~/projets/git-flow-tuto (master)
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         README.rst
        modifié :         requirements.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        build/
```

Note that git understands the `*` wildcard for names of files. So if you want to add to your commit all files that begin with `xyz`, you can simply type the following to add them all at once :

```
git add xyz*
```

Note also that when you change files within folders, you can simply add the folder and git will recursively add the files that have changed in it. Careful though, do that only if you're sure that all the changed files in the folder are to be added to the commit (beware the infamous `.DS_Store`).

The combination of `git status` and `git add` is useful when you want to select precisely the changes to add to the commit (which is, not adding *all* of them), but can be a rather heavy procedure. To add all modified files to a commit at once, simply use the `-a` option of `git commit` :

```
git commit -a -m "your message commit"
```

Again, always check with `git status` that *you do want to add all files* before doing so. Often, unwanted files are recorded in history just because of usage of the `-a` option (especially big files that you're not supposed to store on git). When a file is added to git, it is not that obvious to delete it as if it was never there...

Note that the `-a` option *does not include* new files that were never in git before. For those new files, you have to use `git add`. In such a case and if you want to add at once new files and all modified files, feel free to use the `&&` operator to make two commands on the same line : one to add everything, the other to commit :

```
git add . && git commit -m "Add all modified files !"
```

Caution though, `git add .` is quite a risky command since it literally adds every new or modified file in your repository. Always check with `git status` that there are no unwanted files to be commited before using it.

How to decide which files to add to a commit ? This is up to you really, but a good rule of thumb is that each commit should represent a coherent version of your code (I'm not saying functional, just coherent). Say you've modified two scripts, then documented them in documentation files, and finally wrote unit tests.

A sensible choice here could be a first commit that includes the first script, its documentation and tests, and a second commit that includes the second script, its documentation and tests ! Even if it is not the order in which you wrote things, it is an appropriate organisation of versions for every user that might have a look at your code later.

Several files are modified, several times

Until now I made as if you would commit just in time after each modification. All rigourous that you can be, sometimes it will not be the case. In that situation, you may end up with a lot of files modified, and these modifications would correspond to different steps in your work or different features.

In that case, you generally don't want to add the whole file into the commit, because you want to make it clear to others (or to future-you) that you first worked on feature A on files X and Y, then on feature B on files X and Z. Adding all X, Y and Z to a single commit makes it harder to know what happened in that particuliar moment of the history, and which feature development it was tackling.

The solution is to add *hunks* of your files to the commit. That is : you select the precise lines in your files that will be added, while the others remain in the list of non-committed changes. The easiest way to do that is the following command, for instance for file X :

```
git add -p X
```

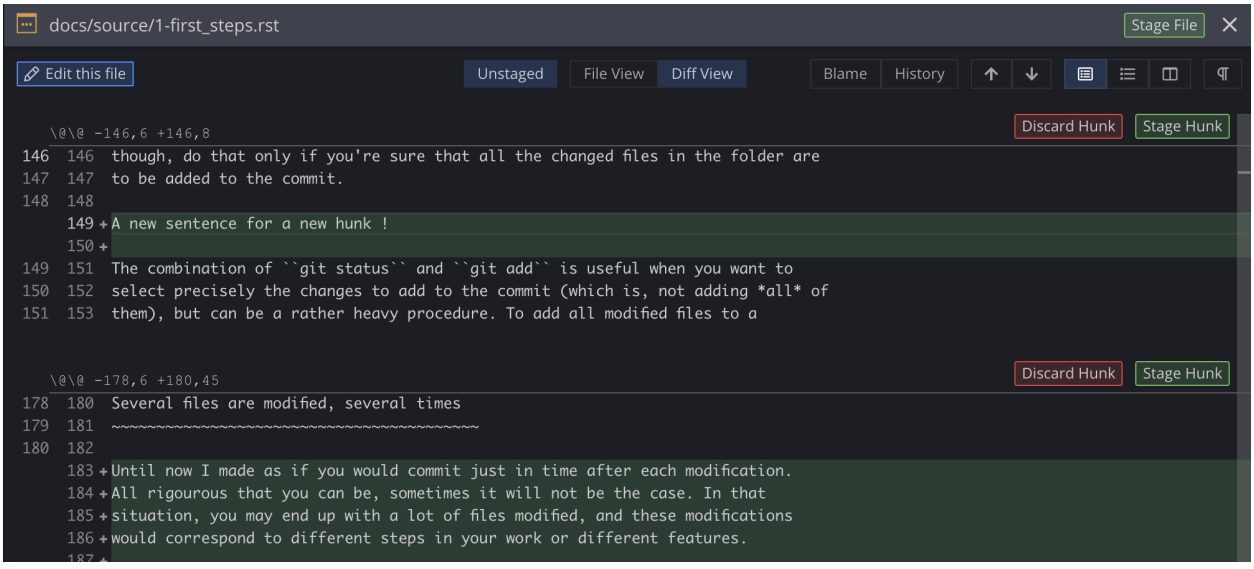
The `-p` option stands for *patch*. Git will display in the console each hunk of your file that is a candidate to be added to the commit. By typing `y` or `n`, you manually add lines of file X to the commit, so that you don't mix up features A and B in your commits ! That procedure is a part of interactive staging, which offers a lot of possibilites that we do not cover here.

```
(venv) romain@romain ~/projets/git-flow-tuto (feature/section-1-1-3)
$ git add -p docs
diff --git a/docs/source/1-first_steps.rst b/docs/source/1-first_steps.rst
index bffb33a..235a4ce 100644
--- a/docs/source/1-first_steps.rst
+++ b/docs/source/1-first_steps.rst
@@ -146,6 +146,8 @@ folder and git will recursively add the files that changed in it. Careful
 though, do that only if you're sure that all the changed files in the folder are
 to be added to the commit.

+A new sentence for a new hunk !
+
 The combination of ``git status`` and ``git add`` is useful when you want to
 select precisely the changes to add to the commit (which is, not adding *all* of
 them), but can be a rather heavy procedure. To add all modified files to a
Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]?
```

With this command, you're starting to see that Git bash is not that friendly when it comes to interact a little bit more than what you do for a simple commit. When you're in the case where you modified a lot of files, and still want to organise properly the commits by adding the right parts of the right files in each commit, do use a GUI to do the interactive staging operation.

It is not only easier, it is also safer since you clearly see what you're doing. From a terminal window, this is often not exactly right. Below is a screenshot from GitKraken. Adding hunks of files is easy and transparent there, as it is with most GUI.



Note that you do not control what Git sees as a hunk. It is in most cases quite effective at severing them at relevant lines, but sometimes this forces you to stage into the same commit lines that you would have wanted to keep in different commits. Well, that's a good reason to commit even more often. You can also decide to select lines one by one, but again I strongly advise to that with a GUI, for the sake of comfort and seing what you're doing.

We'll see later that you can *squash* numerous commits into few commits, so in that matter, when you're developing a new feature, or simply working hard and making a lot of changes, do commit very often. You'll have time to tidy up the commits when the rush is behind you, and you won't regret to have a detailed history of all the changes you made, and unmade, and made back...

5.2.2 Time travel : an introduction

To this point we've only used git to go forward : making changes to the code, then recording them, each commit adding a new point into the repository's timeline. Sometimes you commit something and it is only seconds before you realise that you either forgot to add a file to it, or conversely you've just added a 1Go csv data file that has no reason to be in your git history.

Here are some simple commands to fix this *when it happened in the last commit*. When it happened way before the last commit, head over to the time travel section !

Amend a commit to add missed changes

In this scenario, you've just commited and you realise after checking with `git status` that you've forgotten to add a certain file. No worries here : you can easily add supplementary changes to the last commit with the `--amend` option. It also allows you to rewrite the commit message. If you do not provide a message with `-m "my commit message"`, it will use the previous value but will still ask you to validate the message with the default text editor. The workflow to amend a commit is :

```
git add file1
git commit -m "Add file1 and file 2"
# Ah ! forgot to add file2
git add file2
git commit --amend -m "Add file1 and file2"
```

Amending a commit is already rewriting history, so be careful with that. We'll talk about that in more details below, but for now just remember that **you shouldn't amend a commit that is already synchronised with the remote version of the repository**.

Remove a file that has just been committed

In the second case scenario, you need to remove the file not only from the commit, but also from the whole git history (otherwise that annoying 1Go will stay there). In general, this is not an easy task, but we're here in a special case : the big file has appeared in the very last commit, which makes it not too dangerous to *cancel* that commit without losing everything.

There are several strategies for that, here is the one that is to me the simplest (but probably not the safest) : *reset* your repository to the commit preceding the falty one, while keeping the changes that you've made to the files. In other words, you say to git *go back to where I was just before I commited that last commit*, so that you keep the files in the state they currently are at, but the commit itself has disappeared from the history.

Technically, you have two options : reseting can be made with the option `--soft` or `--mixed`. The first will place you at the moment right before you commited, that is, when changes *were already staged for commit*. The second will place you at the moment where you made changes to the repository's files but staged none of them yet. There is

a third option `--hard` that speaks for itself : it cancels both the commit *and* the changes to the files. For the purpose of demonstration, we will chose here a soft reset. Here is the whole sequence, including the initial falty commit :

```
git add .
git commit -m "Read client names from csv datafile"
# Damn it, this has just added a large clientdata.csv file !

git reset --soft HEAD~1  # this will reset the history to the previous commit
git status               # list the files that were added to the commit in first place
                         # with --soft, they remain staged after reseting
git reset HEAD clientdata.csv   # this is it
git commit -m "Read client names from csv datafile"
```

You'll see that there are two *reset* instructions in the exemple. The first one, `git reset --soft HEAD~1`, applies to the whole repository, while the second one `git reset HEAD clientdata.csv` applies to a single file. The first one is the procedure described in the above paragraph. As it is for `--amend`, *you shouldn't reset after commits that were already synchronised*, since it will create diverging histories the moment you commit something new from the reset timeline. We don't like diverging histories at all : they mess up both contributing safely and the overall readability of the repository.

The second *reset* statement allows you to *withdraw the given file from the list of files that are to be commited*, which is exactly what we want. Remember that thanks to the first reset, git went back to the point where *clientdata.csv* was a new file not appearing in the history.

The last statement is a simple commit, except that now *clientdata.csv* is not included (you can check that on your git client) ! About this exemple, you would probably want to add the large file to the `.gitignore`, so that you won't risk adding it ever again. You can do that simply with the following command :

```
echo clientdata.csv >> .gitignore
```

Another usage of `git reset HEAD file` is when you want to add all files but a few to a commit. Instead of manually adding the ones you want with `git add ...`, you can first add all of them with `git add .`, then withdraw the few ones that you don't want to commit with `git reset HEAD unwanted_file`.

Cancel changes to a file

Until now we were happy with the modifications we did to the files. The previous commands helped us with organising the commits, but what happens if you're unhappy with the changes you've made on a file, even before thinking about a commit ? When there have been a lot of changes, reverting them with the famous `CTRL+Z` is not really a viable option.

Git offers a simple command to cancel all modifications made to a file, which is done by putting it back to the state it was at the last commit :

```
git checkout -- filetocancel
```

Note the difference between `git reset HEAD file` and `git checkout -- file` : the first one is used when you did stage the file with `git add file` and want to withdraw it from the list of files to be committed, *while keeping the changes that you've made on that file*. The second simply puts your file to the state it was at the last valid commit, which is before you changed anything on it.

In other words, `git reset HEAD file` is *not* a destructive operation since it only affects the commit, not the file itself. `git checkout -- file` do affect the file, so think twice before using it because it makes the changes simply vanish (for real) !

Cancel a (past) commit

It will happen that after several commits, you realize that you did something wrong several commits ago. The problem is, you want to cancel what you did in commit n-3 without removing what you did in commits from n-2 to n. You could use a soft or mixed reset as seen above, but that's likely to be complicated and unsafe.

Remember that a commit is a set of modifications to some files. So the right way to cancel those modifications is *to commit the reverse modifications*. That is, if the modifications in the falty commit are *add a new line after line 66 to file A and add file B*, the reverse modifications will be *remove line 67 in file A and remove file B*.

You have two ways of specifying which commit(s) to *revert* : either by giving the number of commits to look back from the current one, or with the falty commit id:

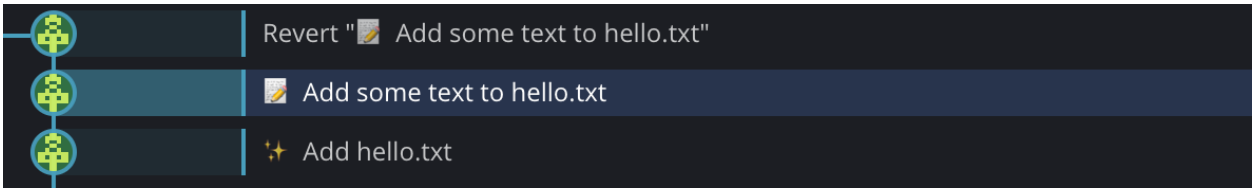
```
git revert HEAD           # revert the current (last) commit
git revert HEAD~1         # revert the previous commit
git revert HEAD~3..HEAD   # revert the last three commits

git revert 4be26a         # revert commit with id 4be26a, a git GUI is your friend
```

Note that reverting a commit *do create a new commit with the reverse modifications* and a default description "Revert commit X". This says that you *cannot change the past by changing what was done in commit X*, for it would mess up all the subsequent commits. Let's understand that a bit more.

Say that commit n-4 creates an empty file `hello.txt`, and commit n-3 adds a `"goodbye"` line to `hello.txt`. Should reverting commit n-4 actually change it, commit n-3 would then indicate writing a line to a file that wouldn't exist anymore, which is nonsense for git.

Instead, thanks to commiting the reverse modification of commit n-4, git simply adds to the timeline a new modification which is *delete file "hello.txt"*. This is much, much safer and robust. Below is a simple exemple of reverting the current commit.



This shows quite well why it is important to *atomize* your commits while developing, which is keeping the number of modifications appearing in a single commit low and coherent. The more you do that, the easier it will be to revert while knowing exactly what you're reverting and what you're not.

Temporarily store your changes

It *will* happen that you've made changes, and for any reason need to quit working on what you were. You're not really ready to make a clean commit, meanwhile you'd like to save the changes, at least temporarily, so that you don't accidentally start over a new development while having forgotten that something had changed.

When you use branches and collaborate with others, it will happen : while you're working on branch A, Joe asks for your help on a bug on branch B. If you try to switch to branch B, git will probably refuse because you have *unstaged changes that would otherwise be lost*.

Enters `git stash`. This command stores your changes so that you can do something elsewhere in the repository, and allows you to get back to it later, as if you were never gone ! It actually is quite similar to a commit, except that it does not record the changes in the timeline, but somewhere nearly *invisible* to you.

To yield your changes back, simply use `git stash pop`. Here is the workflow :

```
# doing stuff on files here, on branch A
# Joe is now asking for help on branch B
git checkout B  # going to branch B won't be allowed because of unstaged changes
git stash       # that's fine, just stash your changes
git checkout B  # now it's ok to switch to branch B
# helping Joe here
git checkout A  # go back to the branch you were working on
git stash pop   # yield the changes you made before so you can work again
```

At this point you undestand what is a commit and how to commit changes to your files with precision, and even forgiveness in case of small errors. Be sure to play with all the commands described here, since they are the ones that you will use the most.

The more you control what's in your commits and when to do them, the cleaner and easier to read your timeline will be. Future-you and your partners will be grateful for that. Now let's dive into the first step to work with others and develop new features safely : mastering branches and merge statements.

5.3 Your first steps with Git (part II)

In the previous section, we rapidly mentionned working with branches, which is all this whole tutorial is about. Now that you know how to manage commits, you will learn the upper-level usage of git : creating, managing and merging branches.

5.3.1 From commits to branches

When you first initialize a git repository, in the terminal window you'll see `(master)` appearing next to the folder path your at. This indicates *which git branch you're checked out at*, which is by default the `master` branch. We'll talk about those names when presenting the *Gitflow branching model*, so don't pay attention to them in this section. For now let's understand what is a branch and what they are here for.

A fantasy of a lifetime

To understand what is a branch, let's work with some fantasy. Imagine that your life follows a timeline onto which things that happen to you are stored (this is git). You can think of a branch as one possible *path* in that timeline. In the real case, your life timeline is purely linear, which means you have only one path where everything is happening, and it's impossible to revert things that have been done (spoiler alert, this is not suitable for your code).

Branches in git work as if you'd have the superpower to test different paths for your life, or even making different lives exist simultaneously. That could be : I have job A, and I would want to know what would be my life with job B. So I create a *job B branch*, experiment with what my life is with job B, and then I can decide if I prefer to keep job A or to move to job B. Pushing to the extreme, I decide that I want to live both so I'll create sort of a duplicated life, where I have job A in one timeline, and job B in the other, maintaining and living both of them simultaneously.

In the git world

This translates adequately to your code. Should it be a data science project or a web application (or any other you fancy), you're likely to have a working *main* version of your code. In the meantime, you want to improve or test new features, without risking to break anything on the main version. This is especially true for a web application that other people than you use or develop onto.

A branch roots from another branch (except the `master` branch which is the origin of your git tree), more precisely from another branch's commit. Most of the time you don't have to think about which commit you're creating the branch from, since it is likely to be the current one. But know that it is also possible to create a branch starting at a chosen commit, for instance when you know things went wrong after that commit, and you want to fix things without modifying the main code nonetheless.

Tarzan, or creating and moving through branches

The git command to navigate through your code's branches is `git checkout`. If you created a brand new repository for this tutorial, you should now be on the `master` branch. On the exemple below, git is checked out at a `develop` branch.

```
romain@romain ~/projets/git-flow-tuto (develop)
$
```

To create a new branch, use the following command:

```
git checkout -b my-new-branch
```

The `-b` option stands for *branch* and is used to indicate to git that the branch is to be created. That command actually wraps the two following commands:

```
git branch my-new-branch   # create my-new-branch but stay on develop
git checkout my-new-branch # move from develop to my-new-branch
```

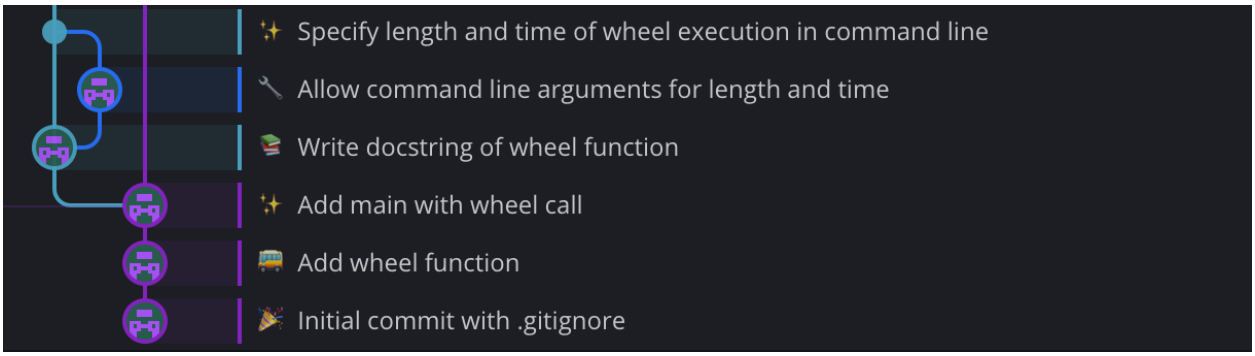
On the screenshot below, I created the branch and git automatically moved to that branch, whose name is visible in the parenthesis next to the current folder path.

```
romain@romain ~/projets/git-flow-tuto (develop)
$ git checkout -b my-new-branch
Basculement sur la nouvelle branche 'my-new-branch'
romain@romain ~/projets/git-flow-tuto (my-new-branch)
$
```

To see the list of available branches, use:

```
git branch        # display only local branches
git branch --all  # display all branches, including remote ones
```

So now that you have at least two branches, say `master` and `my-new-branch`, you can choose either to commit on `master`, or on `my-new-branch`. Working on `my-new-branch` allows you to work freely without affecting `master`. Once you're happy with what you did in `my-new-branch` and you want to add it to the main code, you will proceed to a *merge* of `my-new-branch` onto `master`.



On the screenshot, you see the very first commits of this repository. The purple branch is the main branch, always called `master`. From the third commit, I created a new branch to work on a feature. On that *feature branch*, after a first commit, I created a (blue) *sub-branch*, which was merged back on the first (light blue) feature branch. Once completed, the (light blue) feature branch was then merged onto the `master` branch.

Note that you *don't necessarily have to merge a branch*. You can use branches to make drafts that will never be merged onto anything. In that case, the branch allows you to try and break code without risk.

5.3.2 Merging branches and first collaborations

Now that you know both how to commit and create branches, let's dive into the merge flow. You *merge a branch onto another* when you want to update the target branch with the work (i.e. the commits) that have been done on the merging branch.

This opens up a whole world of organising the way you work with branches, that is how and when to merge branches. We'll call a *branching model* the set of names and rules you use to define how and when to merge branches in your project. The simplest flow is : you don't use branches at all and always work on master. A more reasonable flow is to always create a branch to develop something, then merge it onto master when the new feature is ready and satisfactory.

The more complex your project, the more critical it is to use a reliable and well defined branching model. One of them, which is the object of this tutorial, is the *git flow* branching model. We'll talk about it in the next section. For now, we'll see the various ways of merging a branch onto another, and how to synchronise your local repository with others'.

Three ways of merging a branch

Understanding how merges work is a critical part of a good usage of git and a fluid collaboration. It is that tricky moment you put together the work done in various parts of your code, and as such is subject to a lot of difficulties. So take a break before entering that realm, and more than never experiment with your own code to see how things work.

Git allows for three different types of merge. All of them do update the target branch using the merging branch, but they produce a different timeline on the target branch once the merge is done. Choosing which merge behaviour to use depends on various factors that we'll present here, and using this or that behaviour can be a part of your project's branching model.

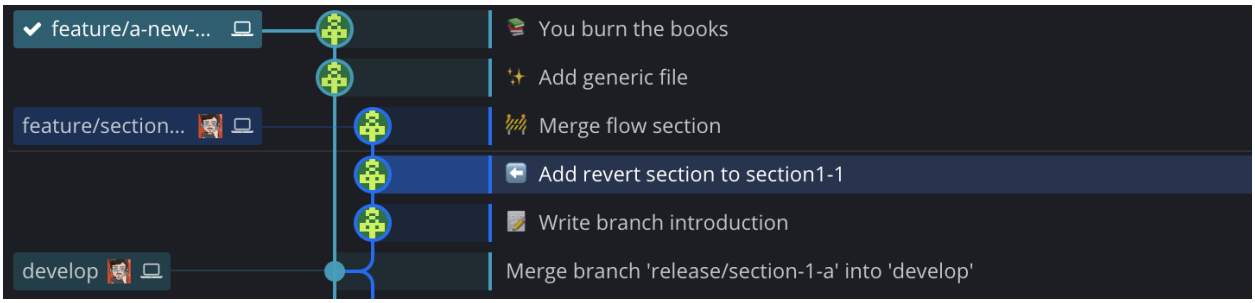
The generic instruction to merge a branch onto another is `git merge`. Say you have a *feature/a-new-hope* branch ready to be merged onto `develop`. I personnaly never remember which is the order to use in the git command, so I first checkout the receiving branch (in that exemple, `develop`), then merge. The flow is

```
git checkout develop          # checkout the receiving branch
git merge feature/a-new-hope  # merge the feature branch onto develop
```

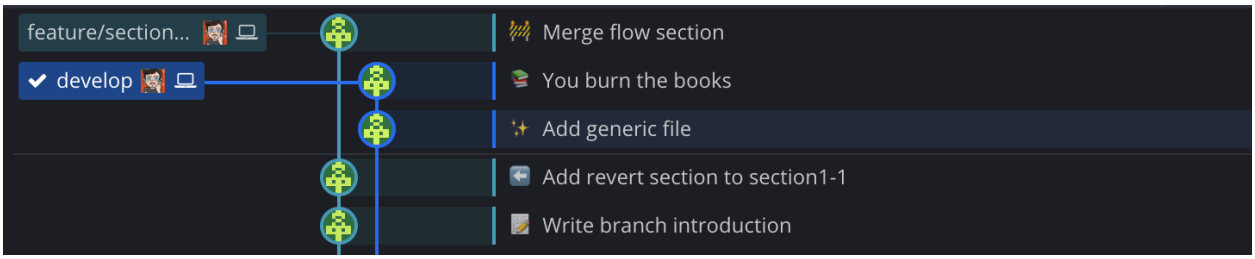
Specify one of the option presented below to choose a specific merge behaviour.

Fast-forward merge

The default behaviour of `git merge` is a fast-forward merge when possible. A *fast-forward* is really just placing the merging branch on top of the receiving one, to obtain a fully linear timeline, as if you had done the commits directly to `develop`.



On the screenshot above, two branches are available from the tip of `develop`. We want to merge branch `feature/a-new-hope` onto it, with a fast-forward. After merging (with the standard `git merge feature/a-new-hope` which uses a fast-forward by default), we get the following timeline :

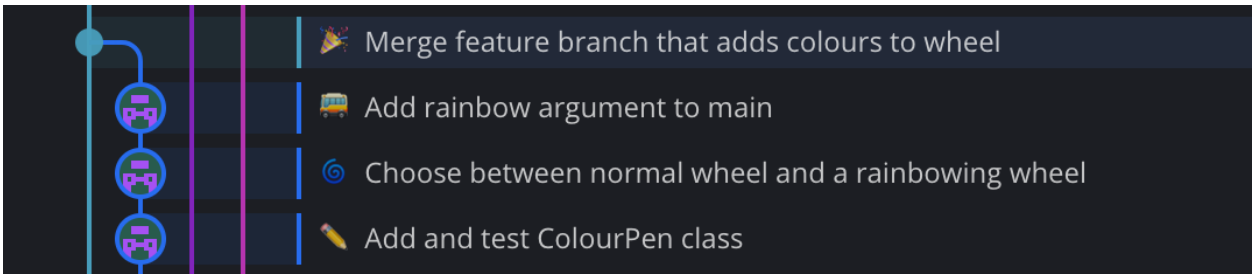


`develop` stays linear with the commits from `feature/a-new-hope` (which I deleted after merging) added. We also see that the branch `feature/section1-2` now departs from an older commit of `develop` than the tip of it. This means that you won't be able to use a fast-forward merge of that branch onto `develop`, since git cannot *add* the commits from it onto the last ones in a straightforward way. If, in that configuration, you try `git merge feature/section1-2 --ff-only`, git will abandon the merge because *a fast-forward is impossible*.

If you don't provide the `--ff-only` option, git will use the *merge commit strategy*. Sometimes you really want to use a fast-forward, so we'll see later how to *move* the merging branch along the tree, so that it can (almost) always be merged using a fast-forward.

Merge commit

The *merge* commit strategy is the one you'll use the most when following the *git flow*. It consists in agregating *on the receiving* branch all the modifications from the merging branch in a single commit, while keeping the full detail of the commits succession on the merging branch. Here is what it looks like :



On the screenshot above, you see a feature branch comprised of three commits, that were merged onto the receiving branch using a single commit that makes the junction between the two branches. There are two main advantages of using merge commits :

- it keeps the timeline of the receiving branch clean and concise, since every new functionality is added through a single commit, contrary to fast-forward merges where all the commits are added on top of the receiving

branch. This allows precise tracking of modifications on the important branches, while keeping the detail of the development history on the feature branch.

- the merge commit itself is a great place to provide detailed explanation about the new feature. Often your working commits are not all self-explanatory, and navigating throughout the history may be harsh, so having a merge commit that explains it all is of great help to other developers that need to know what modifications were added by this merge.

The command for a merge with a merge commit is (from the receiving branch) the following. It actually stands for *no fast-forward*, since, has said previsouly, git will by default perform a merge commit when a fast-forward is not available:

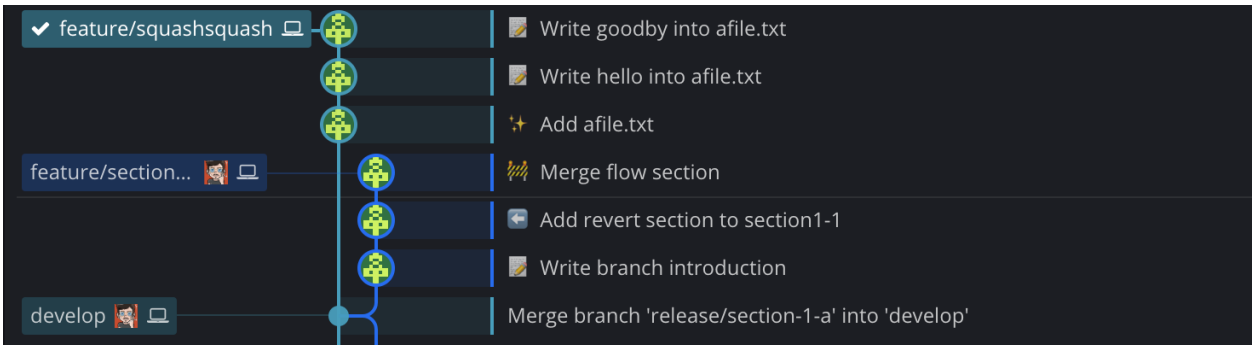
```
git merge --no-ff
```

Git provides a default value for the merge commit message which is something like *merge branch <merging branch> onto <receiving branch>*. You can keep that first line untouched while adding explanations below, or replace it with a more self-explanatory message. We'll see in the next section how to write good commit messages. Also know that the merge commit is often the default behaviour of git hosts such as github, gitlab or bitbucket (we'll see that in the pull requests section), because it perfectly embodies the flow of developing something on a feature branch then adding it all at once on the development or production branch.

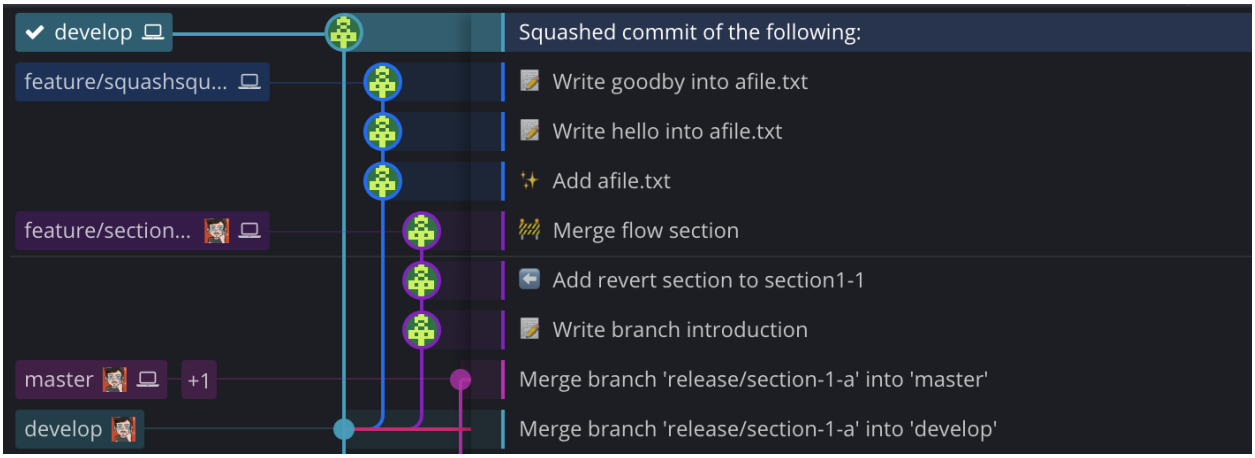
Merge-and-squash

There's a last option that I don't recommend using in general : a *squash merge*. To me, it is kind of a bad compilation of the two previous ones. The behaviour is the following : with a squash merge, all the modifications from the feature branch are *squashed* into a single commit, which is added on top of the receiving branch. Here it how it looks like, before and after.

Before, the feature branch `feature/squashsquash` departs from `develop` and is comprised of three commits.



I then squash-merged that branch onto `develop` with `git merge feature/squashsquash --squash`. When performing a squash, you're creating a new commit out of the branch's ones, so you need to use `git commit` with a message to validate the squash. Once done, the timeline looks as follow :



You see that a new commit was added directly onto `develop`, while the branch `feature/squashsquash` remains living and not seemingly merged into `develop`.

Here is why I say it is a bad compilation of a merge-commit and a fast-forward :

- since the result is a commit directly on `develop` timeline, it will be really hard to distinguish between commits that add a new feature to your code and commits that are *usual* commits (bug fixes, small commits, etc). When the `develop` timeline will be long (and it will), you'll have a hard time trying to find where (ie on which commit) you added that buggy feature.
- since you combine all the branch's commits into one, you'll loose the detail of the branch development history, which is quite useful for debugging. Note that once squashed, you're supposed to delete the merging branch, so you will definitely lose the branch details. Conversely, a merge commit, even if you delete the merging branch, keeps the detail of the development history.

The single case in which I use squash merges : you want to fix something or add a really simple thing to `develop`. You could do it with a normal commit, but you *know* that it is likely that the first trial won't be good enough.

So instead of ending up with three commits because the first wasn't working, with the last two being "Fix previous commit" (), or repeatedly amending that first commit until it works (we'll see later why one doesn't want to amend things on `develop`), you can create a branch, freely do your stuff there – even naming the commits with bad messages – and when you know the modifications are ready to be merge, perform a squash merge, then delete the branch. This is precisely what branches are for : allowing a trial and error development while keeping the important clean and tidy.

There's a good thing about squashes though : they can agregate numerous commits into a single one. The idea of keeping a relatively short development history is good, since it will help identifying milestones in your code, thus making debugging a lot easier than having to look for a bug in a thousand commits.

There are other ways to squash your timeline than the squash merge. We'll see that in the collaboration part of this tutorial. For know, just remember that in most cases the merge commit strategy is the preferred one.

Synchronise with the rest of the world

For now we've only talked about managing our own version of the repository you're working onto. You might remember though that I said to avoid as much as possible rewriting history for branches that are shared with others (mostly `develop` and `master`). We'll see know how sharing with others work.

Remote and local version

In most cases, the repositories you're working onto are hosted and synchronised somewhere on the web, on dedicated websites such as github, gitlab (where this repository is hosted), or bitbucket. The version of the repository that is hosted on the cloud and thus shared by everyone is called the *remote* version of the repository. You can find easliy the remote url of any repository by typing

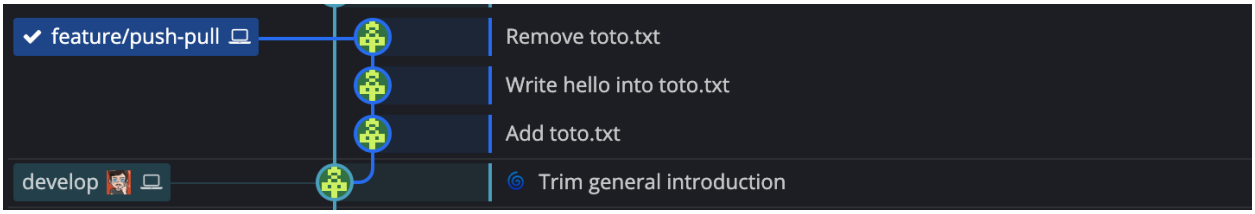
```
git remote get-url origin
```

Contrary to simple files hosting services such as the infamous Dropbox (never use it for hosting code. Never), git does not automatically synchronise changes between your local version and what's on the online version, for a simple reason : it is not always straightforward to know how to merge various changes that have been made in several places. On Dropbox, that would result in a *conflicted copy of Danny's version.*

There's another reason too : you don't have to synchronise everything you do. As the remote version is the one that is shared and accessible by everyone, you might want to keep it clean and sound, without adding all of the sketches that you may have made locally. Let's see what the instructions to synchronise your work and get other's are.

Synchronising your changes : pushing

The first thing to know is how to *publish* your changes on the remote. That is, you've made a series of commits on a given branch – locally – and you want those commits to be saved on the remote version (for backup reasons, or to share it with others). On the below example, I have a branch `feature/push-pull` comprised of three commits. As you can see, there's only a computer icon next to the branch's name : that is GitKraken's way of indicating that it is only a local branch. You can compare with the icons next to `develop`, where there's the computer and the avatar of the gitlab account where the remote is.



I want to synchronise this branch with the remote so that, for instance, my friend Danny can see what I've made. The git term for that is *push*. It is rather intuitive : you've made some changes locally, and you *push* them onto the remote version so that they appear here to, thus being available to anyone contributing to the repo. The command for that is

```
git push    # push when being checked out at the branch you wish to push
```

If you try to push a branch that you've just created locally, git will refuse that push with the indication that it couldn't find a remote reference matching with the branch you're trying to push, and will hint the following command

```
git push --set-upstream origin feature/push-pull
```

What happens here is that the branch does not exist yet on the remote version, and pushing can only work on a branch that exists. So `--set-upstream origin feature/push-pull` indicates that the local branch `feature/push-pull` must be pushed to a new branch on the remote called `feature/push-pull` too. You can mismatch names between local branches and remote ones, but that's only prone to confusion and not advised at all.

Update your local with the remote : pulling

The reverse operation is called (surprise) *pulling*. When you *pull* a branch from the remote, you're telling git to apply to your local branch the commits that were found on the repository and that you don't have yet locally. *Pulling modifies*

your files locally according to the modifications listed in the commits you're pulling from the remote. So you might want to detect changes in the remote *without pulling them*, in case those changes mess up with what you have locally, or just because you don't want them. The command for that is

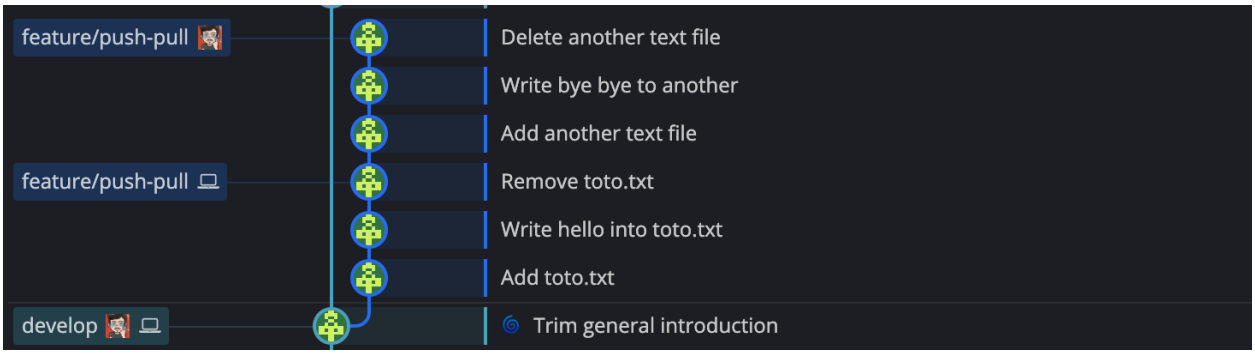
```
git fetch
```

This simply tells git to kind of ping the remote to know what the available changes are. It will detect not only new commits on the current branch, but also new branches. This is of use when you want to get locally a new branch that was created on the remote by someone else. You could create the branch locally then set its upstream to the remote one, but that's a rather heavy procedure for the task.

Instead, simply fetch first, then checkout to the branch as if it already existed locally. Since git has fetched the remote branch, it knows that it exists on the remote, and just has to make a copy of it on your local version. That usually is done in one single command

```
git fetch && git checkout a-new-remote-branch
```

Pulling is full of surprises and often doesn't work as expected. Let's see how it works in the simplest case, were you did nothing locally but there were modifications pushed by others. On the below example, the computer icon indicates where is my local version of `feature/push-pull`. As you can see, there are three commits available on the remote that I don't have on my local branch.



If you use `git status` while being checked out at `feature/push-pull`, git will indicate that your branch is *late* and can be fast-forwarded, using the following command

```
git pull
```

And that's it ! Your local version of `feature/push-pull` has just been updated with the commits that were available on the remote branch `feature/push-pull`. The basic workflow is thus pretty simple : use `git fetch` to know the changes available for pulling, then `git pull` to actually get those changes (you can also just pull, because it fetches automatically before). Then, commit as you want, and use `git push` to update the shared version of the repository with your changes.

You might have noted that pulling was made in the above example with a *fast-forward*, which is the same term that is used for merging branches. That's for a reason : pushing and pulling is actually nothing more than merging two branches, the local and the remote. When you *pull*, you're merging the remote branch onto your local branch, that is applying on your local branch the commits that were made on the remote one. Conversely, when you *push*, you're merging your local branch onto the remote one.

Distribution and priorities

Since pushing and pulling is actually merging, the same problems that can occur with merges occur when synchronising local and remote versions. In most cases pushing and pulling is done with a fast-forward, but it is not always possible. Being unable to push and pull freely is one of the most annoying thing when you start using git.

You made changes that you're not allowed to push because someone else has also made changes and pushed them before. What's the point of using such a complex tool if it is not able to handle that kind of recurrent situation on its own ? This is when the *distributed* part of *DCVS* that I mentioned in the introduction comes into play.

Git does not consider the remote version of the repository as primary. Even if it is often the *origin* for all the various local repositories that you and your team work on (you can read more about that here), the commits there will not be by default applied before yours.

Instead, *any* clone of the repository can contribute equally to all the other clones, including the remote, so that when there are several possibilities of merging branches or adding new commits, it's up to you to place git in a situation where there will be a non-ambiguous way to merge things, or to tell it which way it should merge among all the possibilities. **The responsability is distributed among all clones of the repository, without any being central to others.** In the *Collaborating with git* section, we'll dive into that subject to learn what to do when a fast-forward is not available, and how to avoid being in that situation in the first place.

Up to this point you already know all that is needed to contribute to a repository when things are straight-forward : you know how to commit changes, amend them if needed, create and navigate through branches, merge them and push/pull the changes to the remote and shared version of the repository.

As you can guess, things can get tricky – and they will – and that's why one defines branching models, *which are a set of rules and conventions that dictates how branches should be created and interacted with, to make the development process of a software safer and more fluid for everyone.*

5.4 Gitflow branching model

Branches in git are super effective for developing without affecting the work of others. Yet, without any pre-defined organisation between your team, branches will soon become a nightmare : you have plenty of them, you don't know where is what, you developed something really useful but can't find how to merge it to the published version of your repository without breaking everything, etc.

Branching models are here to help you avoid that. They define a set of branch patterns with a certain function, so that, depending on what you want to do, you know how to do it in git. There are plenty of git workflows, some of them suited for really small teams with simple development patterns, other suited for big teams or open-source projects where a lot of people contribute, developement is long and having at least one version of the software that *always* work is vital.

I present here what is called the *git-flow*. It was first presented by Vincent Driessen in 2010 'A successful git branching model'_. Git flow works really well in a vast majority of developement setups : when you're developing a package on your own, when you're working as a team of three people on a small data analysis project, when you're developing a software with frequent updates with more than a dozen of people.

It is sure rather heavy compared to some other workflows, but thanks to that is able to handle most development problems. After reading this section, you might think that it is far too heavy for your needs. That's perfectly fine ! Even if you don't apply it, consider it as a theoretical reference, which you can cherry-pick ideas and concepts from to find your perfect development workflow. All the images you'll find in this section are taken from this this excellent tutorial from Atlassian.

5.4.1 Dedicated branches for a functional development

The core idea behind git-flow is to maintain a single, tested and working version of your software, with frequent releases, while allowing the development of new features that will not hinder the fact that, well, it's working. It is thus not really suited for softwares that maintain and distribute simultaneous versions. Conversely it works really well as part of an agile team organisation.

As you *will* encounter bugs, even on the main version that has already been released, it is also necessary to know how to handle those bugs without affecting development, and without having to fix them again and again because said fix was lost in the development history.

Let's dive into the subject. Git flow defines a set of branch patterns that *every branch you create will fall into*. For each type of branch, there are rules as to when and where to create them from, when and where to merge them to, what you should do and not do with them, etc. I present them by family here, but you can find a shorter synthesis by branch pattern in the recap page *Basic Git commands*.

Main branches

The git-flow defines two core branches : `master` and `develop`. These are branches that receive tested and verified code, and are later used to release working versions of the software. By definition, they are quite *passive* branches : you don't commit directly on them (except on `develop` on some rare cases), you rather just merge other branches onto them, often after a whole process of code reviewing. Because of that, they are also the most *collective* branches of the repo, receiving new code from various people. Which makes them both fragile and prone to conflicts.

`master` holds the working and distributed version of your software, so it is by far the most important and sensitive part of your repo's history, as its names indicates. One could say that the whole goal of git flow is to allow `master` to be (almost) always working, without bugs, so that it can be distributed or used at any time.

You never work directly onto master. I mean it, never. Any modification, even the smallest one, could break whole parts of your code, and you really don't want that to happen, especially if other people than your team are using the repository (or the software that is deployed from it).

There are ways we'll see below to fix even urgent bugs on `master` without directly working on it, allowing proper review and verification before changing it. Again, do not work directly on `master`. This branch must be composed of only merge commits, and a few if possible compared to other branches. Having a lot of commits on master is a good sign that you don't review and test enough before merging.

`develop` is the receiving branch for every new feature. It is the most active branch of the repository, and as `master`, it is also a collective branch by definition. Because it is on `develop` that you aggregate all the work that has been done here and there, it must be reviewed thoroughly, to be sure that a new feature does not break the existing or introduce regressions.

It is from `develop` that you trigger a merge on `master` (a *release*), which actually makes it the bare bone of your repository, where value is added to the project. Contrary to `master`, it is not supposed to be accessible by users, so it does not have to be working at all times, even if it should. That is why it is allowed to commit stuff on it, when it is simple and safe. In any case, the content of `develop` will be reviewed before merging it onto `master`.

It is a good idea not to allow rewriting history on `develop`, because it would mess up everyone's git history, and first and foremost because it makes it harder to monitor the development history of your code, and find the origin of a bug

Feature branches

A feature branch is a branch dedicated to the development of... new features ! It is where, as a developer, you will spend most of your time. The idea of a feature branch is to allow a safe space to develop a whole new feature. When you start working on it, there is nothing. Where you're finished, the feature branch contains new code, that is supposed to work and can be merged onto the main code. Or it can modify existing code, but again is to be merged only when the modifications end up to something that is working.

Naturally, a feature branch roots from `develop` and will be merged back onto `develop`. Several people can be developing on the same feature branch, or you can work alone on it, even not synchronising it with the remote until it's ready to be merged (which is not advised for obvious back-up reasons).

You can virtually do anything on a feature branch : willingly break things to see how it works, rewrite history again and again, make a thousand commits (we'll see how to fix that afterwards). The only thing is, since it is meant to be merged on `develop` to add value to the repository, it is a good (if not necessary) idea to clean and review it before merging, so that what will be added to `develop` is not only clean and working code, but also features a clean git history to ease future bug tracking.

In general, a feature branch's scope is a single feature. That means you don't develop various features on the same branch : create another branch instead, don't mix things. This allows an easy review, a better understanding of what has changed in the code, and thus makes the whole development process safer.

Conversely, for some really big features that needs a lot of development (both in terms of time and code lines), it is a good idea to sever the work into several feature branches, provided that each feature branch adds a coherent value on its own. It doesn't have to yield an already working new code, but should at least encompass changes that have a meaning altogether.

Another possibility for big features is to create *subfeature* branches, that root from and are merged back to the feature branch itself. In that case, there will be only one final and heavy merge of the full feature branch onto `develop`. I don't really advise that since it is prone to missing changes from `develop` while developing, making the final merge tricky and possibly introducing unforeseen bugs. Testing and merging every subfeature onto develop on the go is much safer to me.

For teams, several feature branches are likely to coexist. That is not at all a problem but requires some precautions as to how and when to merge, and how to update an existing feature branch with work from already merged feature branches. We'll explain thoroughly how to handle such situations in the *Collaborating with git* section.

Releases

There will be a moment when `develop` has received enough new features from feature branches, and you want those new features to be available on the main version of your project (hello `master`). One could simply merge `develop` onto `master`, but that's a bad idea for two reasons :

First, even if every work that was added to `develop` was reviewed, it is important to be sure that the whole bunch of new features that were developed work well together. It is hard – but possible – to test that for every feature merge, but it is in any case safer to make a thorough test of what we will be released on `master`, and to have a dedicated environment (a branch and possibly other testing infrastructures that we don't talk about here) to do that.

Second, it is very likely that you'll need to changes little things when testing the release, at least version number, documentation and so on, that usually get forgotten during development. These are not feature nor bug changes, but needs to be done before a release. Since you're about to commit some stuff, *you don't want to mess up with the development process on* `develop`, which might as well continue in the meantime.

Enter the release branch. It is a simple, short-lived branch, that is created from `develop`, on the last commit that is to be included in the release. This has the benefit of fixing once and for all the version of the code that will be released, no matter what continues to happen on `develop` (that's one of the reasons you don't rewrite history on `develop`).

Here, you might review all the new code once more, but at that point every new line should already have been thoroughly reviewed during feature development. This is also when you test that your software is working as expected for users, or that you're happy with the results you have if it is an analysis project, etc.

Once you're certain that everything is fine (no bugs, no typos, expected behaviour), you *release the version*, which is merging the release branch onto `master`. That way, `master` has been incremented in a single commit with all the new features that were on `develop`, allowing easy tracking of version updates on the main version of the project.

It is not compulsory but a good practice to tag each release on the master branch. A git tag is simply a label that is added to an existing commit, making it easier to track versions or to trigger some specific behaviour (for instance

deployment to a webserver) from the git hosting service. As for commits, git tags must be pushed to the remote

```
git tag -a v0.1 -m "Version kitty cat"    # tag the release commit on master
git push --tags                           # push to new tag to the remote
```

A release branch has to be merged twice : once onto `master`, once onto `develop`, since you may have commited stuff onto it that you need `develop` to get too. Otherwise you'll end up developing on a branch that is not up-to-date with the main and distributed version of your project, which is bad. I mean, really bad, since further releases will be hindered by the fact that `develop` and `master` have diverged.

Fix branches

In an ideal development workflow, you don't have to manage bugs in your code. You develop features, merge them onto `develop`, then release regularly to `master` and deploy your software or publish the updated version of your package.

However, since bugs will occur, one needs a way to handle them that can firstly fix the bug without preventing new features to be developed in the meantime, secondly to be sure that the fix gets applied for the current main version and for all the future ones.

Git flow defines `hotfix` branches for that. Actually, most git hosting services also define `bugfix` branches on top of `hotfix`. In both cases, these are branches dedicated to bugs, which are not feature development. The difference between them revolves around the bug being found on `master` or on `develop`.

`hotfix` are for urgent bugs, that are found on `master` and need to be fixed as soon as possible, without being able to wait for the next release. A hotfix branch roots from master and merge back directly to master, *bypassing the usual feature then release development workflow*.

For that reason, they are quite delicate to handle and they really need to be limited to the bug fix only. Any other change can wait, it will be published with the next release. This allows precise review and testing of the bug fix, and makes it easier to merge onto master having the confidence that the changes will not break other things.

As for release branches, any hotfix branch that is merged onto `master` must also be merged onto `develop`. Without this supplementary merge, the bug will occur again with the next release, since the buggy part of the code wouldn't have been fixed on `develop`, and the next release might erase the bug fix that was done only on `master`.

That's another reason why it's critical not to allow `master` and `develop` to diverge. Diverging `master` and `develop` makes it really hard to dispatch changes from one onto the other, triggering merge conflicts everytime. We'll talk later about merge conflicts. They are common and healthy between features, but dangerous when found between `develop` and `master`, because depending on how you handle them, you can lose information on the choices made and reintroduce lost bugs. So for now remember one thing : **merge back hotfix branches onto develop**.

`bugfix` are simpler to handle, since they are here to fix bugs that are found on `develop` during the development process, that is before a release. That can occur when two new features, despite working well on their own, have appeared to introduce an unexpected behaviour once merged onto `develop`. In that case, a `bugfix` branch is opened from `develop` then merged back onto it, waiting to be published with the next release.

All in all, a `bugfix` branch behaves exactly like a `feature` branch. It is just named differently to allow a more precise product management. That process also shows the robustness of a well-done git flow : it is really not a problem if unexpected bugs are introduced while developing, provided you detect them *before* the next release. If so, the main distributed version will not be affected and you don't have to risk modifying it on the fly. That also strengthen the importance of peer-reviewing and functional testing for every new feature, and before every release.

5.4.2 How to work with git-flow

A summary of the *git flow* workflow

Feature developement

Releases

Bug handling

Integration within your git hosting service

Branching models

Branch permissions

Issue tracking

Those annoying yet critical pull requests

5.5 Collaborating with git

Know the power of rebase

Rebase or merge

How to handle conflicts

5.5.1 Commit messages conventions

5.5.2 Continuous integration

Conclusion

5.6 Basic Git commands

5.6.1 Commiting changes

```
git add file1    # add file1 to soon-to-come commit
git add file2    # add file2 also
git commit -m "Update file1 and file2"

git add file*    # add all files starting with `file`

# Add all changed files and commit, does not include new files
git commit -a -m "Your commit message"

git status       # display the list of changed / added files

# Add all files including new ones, then commit
git add . && git commit -m "Your commit message"
```

(continues on next page)

(continued from previous page)

```
git commit -a      # No -m option will open up a text editor

git add -p file1   # Add hunks of file1 in interactive mode
```

5.6.2 Time flexibility : reset, checkout, amend, revert and stash

```
# reset the git history to the previous commit while
# keeping the changes to files, and keeping them added
git reset --soft HEAD~1

# same but changes are not added here
git reset --mixed HEAD~1

# changes are all cancelled /!\ DANGER /!\
git reset --hard HEAD~1

git reset --soft HEAD~n  # soft reset to the n-th commit backwards

# withdraw file1 from the list of files included in the commit to come
git reset HEAD file1

# cancel uncommited changes to file1
git checkout -- file1
git checkout -- .   # cancel all uncommited changes

git commit --amend  # rewrite the last commit

# revert previous commit
git revert HEAD~1

# revert last three commits
git revert HEAD~3..HEAD

# revert specific commit
git revert 34abe9

git stash           # stash your changes to store them temporarily
git stash pop       # apply the stashed changes to start over from there
```

5.7 Branches basics

5.7.1 Create and navigate

```
# create a new branch with name mybranch
git branch mybranch

# move to (checkout) that branch
git checkout mybranch

# create the branch then checkout
git checkout -b mybranch
```

(continues on next page)

(continued from previous page)

```
# list branches
git branch         # local branches only
git branch --all   # include remote branches
```

5.7.2 Merges

All the following merges are to be made while being checkout out at the receiving branch.

```
# default merge instruction, fast-forward if possible, other wise
# merge commit
git merge mybranch

# fast forward only (will abandon if not possible)
git merge mybranch --ff-only

# no fast-forward, which means a merge commit
git merge mybranch --no-ff

# squash merge - not to use in general
git merge mybranch --squash
```

5.7.3 Synchronise

```
# ping the remote to list all changes available
git fetch --all

# pull the changes available (fast-forward if possible) for the current branch
git pull

# push the local commits to the remote branch
git push

# add a tag an push it to the remote
git tag -a mytag -m "Message for that tag" && git push --tags

# remove a tag locally and remotely
git tag -d mytag && git push --delete origin mytag
```

5.8 Git-flow definitions

5.8.1 Branches

`master`

`develop`

`feature`

`release`

`hotfix`

```
bugfix
```