A Git and Github Starter document

Philippe Rosales

7 février 2023

Table des matières

Ce document est prévu pour donner les commandes de base afin de s'en sortir avec Git et GitHub.

Avant toute chose voici le lien de la Doc de Git. Pour qui veut savoir ce qu'est Git, elle explique très clairement d'où ça vient, pourquoi ça existe et comment git fonctionne. Je donne une brève explication mais terriblement sommaire, si vous êtes curieux la doc vous expliquera tout.

Pour l'installation de Git voir ici.

Je vous encourage à tester toutes les commandes présentées sur un dépôt `test` qui vous permettra de tout essayer sans risquer d'abimer un vrai projet. Rien n'est difficile mais il faut quand même rester vigilant (comme le conseille raisonnablement Fol Œil).

1 Phase Hors ligne : Git en local

1.1 Fonctionnement sommaire

Même si on entend beaucoup parler du site GitHub, Git est tout d'abord un programme qui fonctionne sans avoir besoin d'un accès à internet. Il fait du contrôle de version, et comme ce mot l'indique assez clairement ça permet de gérer les versions qu'on a d'un programme en cours de développement.

Supposons que vous commencez un projet scolaire, Git peut vous permettre de faire des sauvegardes à certains moments, par exemple quand vous avez fini une étape dans le développement, ou que vous voulez "garder en sûreté" une version qui marche avant de continuer. C'est exactement le rôle rempli par les dossiers `Projet`, `Projet-backup`, `Projet-backup(1)`, `Projet-backup-vendredi`...

Pour le vocabulaire, un dossier géré avec Git s'appelle un **dépôt Git**.

Git fonctionne en créant des "instantanés" des fichiers de votre projet. Sans plus attendre voyons comment créer un dépôt vide ou à partir d'un projet existant.

1.2 Création d'un dépôt Git : `git init`

La commande pour créer un dépôt :

```
1        git init
2
```

Ceci devrait afficher une ligne du style : (ici avec linux, Windows afficherait un chemin différent)

```
1        Initialized empty Git repository in /home/User/.../DossierProjet/.git/
2
```

Cette commande crée un dossier caché `.git` qui contient les informations nécessaires au fonctionnement de git. La présence de ce `.git` transforme votre simple dossier de projet en dépôt Git surpuissant ! Si vous supprimez ce dossier, votre dossier contenant votre projet redeviendra un dossier banal... (mais vous conserverez vos fichiers pas d'inquiétude).

Et voilà c'est tout. Maintenant on va voir comment gérer ce dépôt.

1.3 Savoir où on en est dans notre dépôt : `git status` et `git log`

La commande indispensable pour s'y retrouver avec Git c'est `git status`. Git est très explicite, il décrira parfaitement la situation actuelle et vous dira quelles commandes vous pouvez effectuer. **C'est une commande à retenir en priorité**. Un `git status` dans un dépôt sain donnera :

```
        On branch master
        Your branch is up to date with 'origin/master'.

        nothing to commit, working tree clean

```

`git status` vous dit toujours sur quelle branche vous êtes, l'état du dépôt, et les choses à faire.

S'il y a des actions à effectuer, `git status` pourra dire différentes choses :

Il y a des fichiers non suivis

```
        On branch master
        Untracked files:
          (use "git add <file.extension>..." to include in what will be
    committed)
            file.extenstion

        nothing added to commit but untracked files present (use "git add" to
    track)

```

Il y a des fichiers modifiés

```
        On branch master
        Changes not staged for commit:
          (use "git add <file.extension>..." to update what will be committed)
          (use "git restore <file.extension>..." to discard changes in working
    directory)
            modified:   file.extension

        nothing added to commit but untracked files present (use "git add" to
    track)

```

Ces fichiers ont été ajoutés aux changements qui seront enregistrés

```
        On branch master
        Changes to be committed:
          (use "git restore --staged <file.extension>..." to unstage)
            new file:   file.extension
            modified:   file_2.extenstion

```

Il y a aussi `git log` qui peut vous éclairer très efficacement en vous disant où vous en êtes précisément grâce aux options :

```
        git log --graph --all

```

Si l'affichage n'est pas très joli vous pouvez essayer d'ajouter l'option `--decorate`.

Ceci produit un affichage de ce genre :

```
    * commit 5dd015f113d8d96028877fe42120c7088d7659c3
    | Author: Utilisateur <user@mail.com>
    | Date:   Fri Nov 11 15:22:30 2022 +0100
    |
    |     Changed the example with parameters, the n definition needed m and M to be defined as well. Updated the README.md accordingly.        # Example of commit message
    |
    * commit d3a4e374aad615e292a7def8f47393aa3829b060
      Author: Utilisateur <user@mail.com>
      Date:   Fri Nov 11 03:06:15 2022 +0100

          Initial commit        # Example of initial commit message


```

Ou comme ça, s'il y a plusieurs branches (ce que nous verrons plus tard) :

```
    ...
    | * commit 8d7562e660851382161ef4eaccd12335cd5cb1e2 (origin/music)
    |/  Author: Utilisateur <user1@mail.com>
    |   Date:   Wed May 25 23:54:22 2022 +0200
    |
    |       music implementation
    |
    *   commit 356b79ad3a23346a271a60163ad882d6771e7e87
    |\  Merge: ccc3eaf 903f59b
    | | Author: Utilisateur2 <user2@mail.com>
    | | Date:   Tue May 24 17:07:29 2022 +0200
    | |
    | |     merged levelTest
    | |
    | | * commit 123132ecd139895a3c4c581abcbe2944bbb5868f (origin/mergelevel)
    | | | Author: Utilisateur3 <user3@mail.com>
    | | | Date:   Wed May 25 23:44:06 2022 +0200
    | | |
    | | |     triangle select
    | | |
    | | * commit 5f102b7cb2ac677544eef505d6a3e717ac77df19
    | | | Author: Utilisateur3 <user3@mail.com>
    | | | Date:   Wed May 25 01:29:19 2022 +0200
    | | |
    | | |     level harmonise
    | | |
    | | * commit da247974664ee4159cc6265f1e2b27f4fada1f16
    | |/  Author: Utilisateur3 <user3@mail.com>
    |/|   Date:   Tue May 24 21:07:31 2022 +0200
    | |
    | |       merge niveaux
    | |
    * | commit ccc3eaf9d0620a567964b9e43c5c4466e537239b
    | | Author: Utilisateur2 <user2@mail.com>
    | | Date:   Tue May 24 15:26:25 2022 +0200
    | |
    | |     Cleaned menu functions, currently works by bypassing the quadtree
```

```
        ...
```

1.4 Ajouter des fichiers au dépôt : `git add`

Quand vous allez créer vos fichiers source il faut dire à Git qu'il faut les "traquer", disons plutôt suivre, c'est à dire qu'il doit surveiller leurs changements (afin de contrôler leur version).
La commande pour ajouter un fichier ou un dossier aux fichiers suivis est :

```
        git add fichier.extension
        git add Dossier
```

Si cette commande n'affiche rien, c'est que tout se passe bien.

N'hésitez pas à utiliser `git status` pour vérifer que tout est en ordre.

1.5 Arrêter de suivre des fichiers : `git rm`

Si vous voulez arrêter de suivre un fichier la commande à utiliser est

```
        git rm file.extension
```

Si vous voulez constamment ignorer des fichiers vous pouvez ajouter leurs chemins à un fichier nommé `.gitignore` (s'il n'existe pas vous pouvez le créer vous-même). Exemple de fichier :

```
        # .gitignore
        *.o     # Va ignorer tous les fichiers .o
        *.save  # Ignore les fichiers de backup de certains editeurs de texte
        main    # Ignore l'executable cree
```

Une fois ceci fait, effectuons un `commit` !

1.6 Enregistrer l'état actuel de notre projet : `git commit`

Pour que Git enregistre une version, un "instantané" de notre projet, on utilise la commande :

```
        git commit
```

Qui ouvre un fichier texte contenant un message de base. En fait chaque commit doit avoir un message de commit, pour dire ce qui a été effectué depuis la dernière fois. En sauvegardant et en fermant ce fichier texte on valide le commit. On peut obtenir le même résultat en donnant le message directement dans la ligne de commande :

```
        git commit -m "Message de commit."
```

Si le commit a fonctionné vous obtenez quelque chose du genre :

```
        [master (root-commit) 221ec6e] <ici votre message de commit>
         2 files changed, 14 insertions(+)
         create mode 100354 file.extension
         create mode 100354 file_2.extension
```

Félicitations, ça a marché !

Si ça ne fonctionne pas il y a trop de raisons possibles pour en parler ici mais je suis sûr que Git et Google vous aideront.

Si vous avez oublié un fichier à ajouter dans votre dernier commit, ou un à supprimer, pas de problème car `git commit --amend` est là pour ça. Faites les changements que vous voulez, puis utilisez cette commande pour mettre à jour le dernier commit. Si vous ne voulez pas changer le message vous pouvez utiliser `git commit --amend --no-edit`.

Encore une fois, vous pouvez vérifier que vous n'oubliez rien avec `git status`.

(Si vous utilisez un dépôt avec une branche remote il faudra ensuite utiliser `git push --force`. Mais nous verrons ça plus tard.)

1.7 (Optionnel) Revenir en arrière sur d'autres commit : `git reset` et `git rebase`

Si vous voulez revenir en errière, pour modifier des commits ou carrément repartir dans une autre direction vous devrez explorer les commandes `git reset [--hard|--soft] [<commit number>]` ou `git rebase '<commit number>'`. Comme ces commandes sont un peu plus avancées je n'en parle pas trop ici, mais elles sont quand même largement utilisable et largement documentées.
Résumons juste l'utilisation de `git reset` :

Ceci permet de retourner au commit dont on donne le numéro **sans toucher les fichiers actuels** :

```
1        git reset --soft b8bc133315cb359090b7208e64b5df8519d98458       #
   Example of soft reset with an arbitrary commit number
2
```

Ceci permet de retourner au commit dont on donne le numéro **en remplaçant les fichiers actuels par ceux du commit donné (changements depuis le commit donné perdus)** :

```
1        git reset --hard b8bc133315cb359090b7208e64b5df8519d98458       #
   Example of soft reset with an arbitrary commit number
2
```

1.8 (Optionnel) gestion des branches : `git checkout` et `git branch`

Si vous avez lu la doc vous savez déjà ce que c'est. Sinon, il faut savoir que Git fonctionne avec des branches, pour des projets simples vous pourrez n'en avoir qu'une, en utiliser plusieurs peut servir pour implémenter des fonctionnalités dont on n'est pas sûr avant de les fusionner avec le projet principal.

Comme toujours, et presque plus encore pour travailler sur les branches, un `git status` ou un `git log` seront d'une grande aide pour vous y retrouver.

Pour créer une nouvelle branche, utilisez :

```
1        git checkout -b newbranch
2
```

Cela vous placera sur la nouvelle branche directement.

Pour changer de branche c'est la même commande, sans le `-b` :

```
1        git checkout <branch>
2
```

Si vous souhaitez supprimer une branche (attention pas d'annulation possible sauf en recopiant une autre copie du dépôt) c'est :

```
        git branch --delete <branche>

```

Après avoir changé de branche tout se passe comme d'habitude, `git add && git commit` tout ça... Pour les besoins de fusion c'est la section suivante.

1.9 Fusion de branches : `git merge`

C'est une opération un peu complexe, voici la doc qui vous expliquera tout : `merge`.
Un `merge` crée un nouveau commit (sur la branche actuelle par défaut). La commande est :

```
        git merge <branch>

```

1.9.1 Le cas facile

L'opération `merge` refait sur la branche courante toutes les modifications enregistrées sur la branche renseignée (qu'elle soit locale ou distante), la branche upstream choisie par défaut. Ceci permet d'obtenir localement une copie de la branche distante.
Si la branche locale est juste en retard, sans avoir fait de commit, alors le merge fait seulement un "fast-forward", de manière assez évidente ça remet la branche locale à jour en l'avançant au dernier commit.

Si les deux branches ont subi des modifications et des commit il faut que les changements de la branche qu'on merge ne touchent pas les mêmes endroits que les changements effectués dans la branche courante, sinon il y a un conflit de fusion, la bête noire de beaucoup de programmeurs, le fameux...

1.9.2 Et le terrifiant `merge conflict`

```
            Auto-merging file
            CONFLICT (content): Merge conflict in file
            Automatic merge failed; fix conflicts and then commit the result.

```

En cas de merge conflict vous serz averti par Git. A ce moment là si vous voulez conserver les changements de la branche fusionnée vous pouvez faire un `reset` de la branche actuelle au dernier commit commun, puis refaire un `merge` qui cette fois pourra se faire en fast-forward. Si c'est la branche courante dont vous voulez garder les changements le plus simplle est de se mettre sur l'autre branche et de fusionner la branche dont vous voulez garder les changements, en faisant un `reset` sur celle à écraser.
Dans le cas où vous voudriez choisir au cas par cas, je vous conseille d'utiliser un outil comme VS Code (mon favori pour les `merge conflict`) qui vous permettra de choisir en un clic pour chaque conflit.

La résolution des conflits n'est pas facile à expliquer comme ça, vous vous ferez la main avec vos premiers dépôts. Mais dans tous les cas je vous rassure, ça fait peur mais c'est largement gérable. Il ne faut pas se laisser impressionner par le fait qu'on ne voit pas directement pourquoi ça plante.

2 Phase En ligne : GitHub

Personal Access Token

Pour la question du token demandé lorsque vous essayez de renseigner votre mot de passe en ligne de commande, il vous faut un **Personal access token (classic)**. Référez-vous à cette page, qui vous dit tout très clairement. Je préciserai seulement que le nom du token c'est juste pour vous, vous pouvez l'appeler

Mickey si vous voulez, pour l'expiration vous pouvez ne pas en mettre pour ne pas avoir besoin de recréer un token plus tard, et pour les cases à cocher je conseille de *ne pas* cocher **delete_repo**.

2.1 Quelle différence ?

Utiliser Git conjointement avec GitHub ne veut pas dire qu'ils sont la même chose. Il faut bien différencier Git d'une part qui est le programme gérant vos commits, push, pull et autres bricolages ; et GitHub d'autre part, un site hébergant une copie de votre dépôt Git. C'est un peu comme un drive qui serait orienté et prévu spécifiquement pour les dépôts Git.
On parle de dépôt `remote` pour la copie en ligne et de dépôt `local` pour le dépôt... local (sur votre ordi).

2.2 Télécharger un dépôt depuis GitHub

Un dépôt sur GitHub est là pour pouvoir être téléchargé (on dit aussi **cloner** un dépôt), ce qui se fait avec la commande suivante :

```
        git clone <url.git>

```

Exemple :

```
        git clone https://github.com/url/depot.git

```

Ce qui doit donner quelque chose de ce genre :

```
        Cloning into 'depot'...
        remote: Enumerating objects: 15, done.
        remote: Counting objects: 100% (15/15), done.
        remote: Compressing objects: 100% (9/9), done.
        remote: Total 15 (delta 6), reused 14 (delta 5), pack-reused 0
        Receiving objects: 100% (15/15), 5.16 KiB | 5.16 MiB/s, done.
        Resolving deltas: 100% (6/6), done.

```

Ceci crée un dossier nommé d'après le dépôt, là où vous avez exécuté la commande. Il faut donc la lancer là où vous voulez ce nouveau dossier.
Si d'aventure le contenu du dépôt sur GitHub changeait il faudrait le mettre à jour manuellement, le dossier ne se met pas à jour tout seul.
Comme on l'a vu dans la partie sur le Git local, le dossier nouvellement téléchargé est un dépôt Git grâce au sous-dossier `.git` contenu dedans. Nous pouvons utiliser `git status` pour vérifier que tout s'est bien passé.

Maintenant que nous avons vu comment télécharger (cloner) un dépôt, nous verrons plus tard comment le mettre à jour après que le dépôt local ou en ligne (ou les deux) a été modifié.

Nous allons maintenant voir comment créer un dépôt `remote`.

2.3 Comment et quand créer un dépôt sur GitHub

On crée un dépôt sur GitHub au moment où on veut partager son dépôt, par exemple pour travailler à plusieurs sur le même projet. Soyons honnêtes, si ce n'est pas pour le rendre visible (pour son CV par exemple) ou pour collaborer ça n'a pas beaucoup d'intérêt. Le dépôt GitHub peut être créé à n'importe quel moment, avant ou après la création du dépôt local.

2.3.1 Création du `online` avant le `local`

Dans l'ordre on crée souvent le dépôt en ligne après le dépôt local, mais l'inverse est possible. Dans ce cas on utilise le service voulu (ici je présente GitHub mais GitLab en est un autre), on crée un dépôt vierge en ligne en suivant les instructions du site correspondant, et on clone le nouveau dépôt comme vu précédemment. Et voilà. Dépôt local créé ¯_(ツ)_/¯.
En téléchargeant ainsi directement le dépôt le `remote` et la branche upstream sont définis automatiquement.

Cette méthode est un petit peu plus rapide mais j'encourage tout de même un éventuel lecteur impatient à lire la partie suivante qui permet de mieux comprendre comment le local et le distant sont reliés.

2.3.2 Création du `online` après le `local`

Cette partie ne comporte que deux commandes très simples. Tout le texte ne sert qu'à expliquer ce qu'elles font

Lorsque le git existe déjà localement, on crée un dépôt vide sur GitHub, sans rien dedans. En particulier il faut décocher les cases qui permettent de créer un readme ou d'autres fichiers, ils sont facultatifs et si vous en avez besoin ils sont de toute façon déjà dans votre dépôt.
Ensuite les instructions de la page sur laquelle vous arrivez devraient être suffisantes pour que vous vous en sortiez, mais je les remets quand même ici.

Résumé de la section :

```
1             git remote add origin https://github.com/user/repo.git
2             git branch --set-upstream-to origin master
3
```

Développons ce que je viens d'écrire pour ceux qui le souhaitent (la compréhension de ces deux lignes n'est pas nécessaire, mais j'encourage encore une fois tout lecteur à poursuivre la lecture de la section).

Admettons que vous ayez comme nom d'utilisateur `user`, et que vous ayez créé le dépôt `repo` qui hérite de l'url `https://github.com/user/repo.git`.

Il faut dire à git l'url où se trouve la copie de votre dépôt en ligne, pour qu'il sache à quel endroit il doit télécharger et envoyer les fichiers. Cela se fait avec la commande :

```
1             git remote add origin https://github.com/user/repo.git
2
```

> Et je profite de la possible question "c'est quoi `origin` ?" pour faire un aparté à son sujet. Une url enregistrée dans un dépôt git est une `remote`, et il peut exister plusieurs `remote` pour un seul git, mais c'est bien loin d'être utile pour nous, alors nous n'en parlons pas ici. A nouveau la meilleure doc du monde vous tend les bras si vous voulez explorer les multiples `remote`.
> Dans tous les cas nous n'avons ici besoin que d'une seule `remote`, qui se nomme assez généralement origin par convention (mais vous pourriez la nommer `piano` si ça vous chante), et c'est pour ça que ce mot apparaît dans la commande. Fin de l'aparté.

On pourrait illustrer le `remote` ainsi :

Vous pouvez essayer de push mais à ce stade il faudra probablement lui dire avec quelle branche distante synchroniser la (ou les) branche(s) locale(s), s'il ne le fait pas tout seul. Si vous devez faire cette configuration il faut utiliser la commande `git branch` (encore une fois la doc pour plus de précisions) avec l'option `--set-upstream-to` ou son raccourci `-u`.
C'est possible d'effectuer ceci pendant le push, nous le verrons après.

Cette opération, maintenant que Git sait où est votre dépôt dans le vaste internet, fait correspondre les branches locales avec les distantes.

La commande est de la forme :

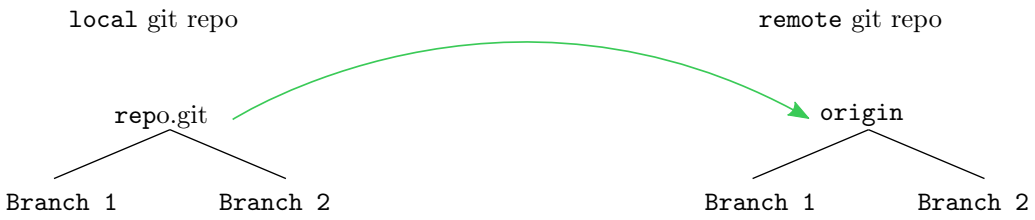

FIGURE 1 – addremote

```
1            git branch --set-upstream-to <remote name> <remote branch name>
    [optional : <local branch name>]
2
```

Ainsi, supposons que votre dépôt ne contienne que la branche `master`, il vous faudrait utiliser :

```
1            git branch --set-upstream-to origin master
2
```

Ou de manière équivalente :

```
1            git branch -u origin master
2
```

Ici on dit "La branche actuelle est à synchroniser avec la branche `master` du remote `origin`".
Ce qui peut être illustré ainsi :

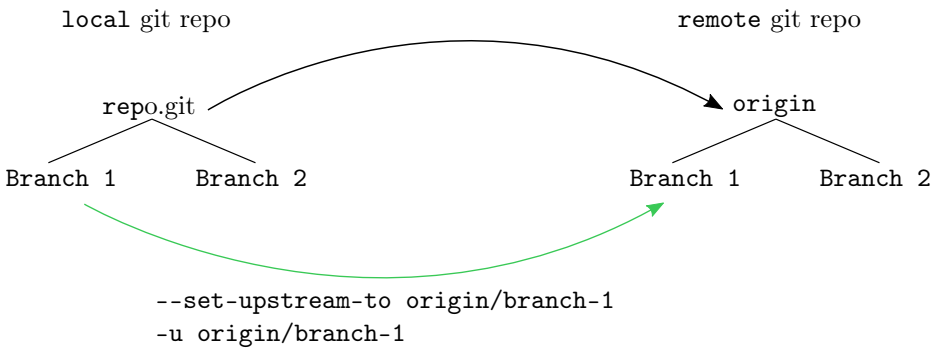

FIGURE 2 – setupstream

On peut éventuellement spécifier la branche à relier :

```
1            git branch --set-upstream-to origin master master
2
```

Dans ce cas le premier `master` est le nom de la branche distante, le second est le nom de la branche locale.
Voilà qui vous a donné les bases et un peu plus pour vous en sortir avec la création de dépôt de A à Z.

Voilà qui clôt la création de notre dépôt sur GitHub et son association avec notre dépôt local.

C'est maintenant l'heure de découvrir comment synchroniser dans un sens et dans l'autre (sens dessus dessous) nos dépôts. En route pour de nouvelles aventures !

La synchronisation de dépôt, à partir du local ou du distant

Note : Les sections suivantes sont toutes concernées par les fameux `merge conflict` qui seront vus à la fin. Ne nous embêtons pas dès le début.

Quand vous voulez mettre à jour le dépôt distant après avoir modifié les fichiers locaux, il faut "pousser" les modifications sur le serveur. Ceci se fait avec la commande :

```
        git push

```

Cette commande est un raccourci de `git push origin` (`origin` le remote par défaut). Ceci synchronise la branche distante en entier, pour qu'elle soit identique à la branche locale.

Ca doit vous afficher quelque chose dans ce genre :

```
        Username for 'https://github.com': user
        Password for 'https://user@github.com':
        Enumerating objects: 5, done.
        Counting objects: 100% (5/5), done.
        Delta compression using up to 8 threads Compressing objects: 100% (4/4),
    done.
        Writing objects: 100% (5/5), 909 bytes | 909.00 KiB/s, done.
        Total 5 (delta 1), reused 0 (delta 0), pack-reused 0 remote: Resolving
    deltas: 100% (1/1), done.
        To https://github.com/user/repo.git
           3ff92aa..c488d39  main -> main

```

Les "numéros" en hexadécimal `3ff92aa..c488d39` sont les numéros des commits respectifs, le commit précédent `3ff92aa` et le nouveau `c488d39`.

Si la branche est nouvelle pour le remote (premier commit par exemple, ou nouvelle branche de test) la dernière ligne l'indiquera :

```
        To https://github.com/user/repo.git
         * [new branch]      main -> main

```

Synchronisation avec la branche upstream finie ! Nous pouvons maintenant voir comment faire pour synchroniser les modifications depuis une branche remote.

2.4 Synchronisation depuis le remote : `fetch`, `merge` et `pull`

Commençons par un point important, dans la configuration par défaut `git pull` est un raccourci de `git fetch` suivi de `git merge`, je vais donc détailler ces deux commandes.

2.4.1 `git fetch`

`git fetch` synchronise l'arbre local avec l'arbre distant, ceci signifie que Git a connaissance des différentes branches du remote et de leurs commit. Cela s'utilise avec :

```
        git fetch <remote>

```

Il est possible de fetch seulement une branche en la spécifiant :

```
        git fetch <remote> <branch>

```

C'est tout ce que fait cette commande.

2.4.2 `merge`

A la suite d'un `fetch` on effectue généralement un `merge`. Maintenant que vous êtes des pros du `merge`, je suis sûr que ça va se passer comme des roulettes à la Poste.
Si vou savez des moments contrariants à ce stade je ne peux que vous recommander notre cher moteur de recherche préféré ainsi que le grand StackOverflow qui sauront vous éclairer comme ils l'ont fait pour moi.

2.4.3 `pull`

Le pull est donc une succession de `fetch` et `merge`. Tout le monde utilise `pull` plutôt que d'enchapiner les deux commandes, faites-le aussi, mais souvenez-vous que ce n'est qu'un raccourci, ça vous évitera d'être surpris au premier merge conflict suivant un `pull`.

2.5 En cas de catastrophe : le guide survie

Les techniques étudiées ici sont à utiliser en dernier recours, quand il ne vous reste qu'une seule chance avant de tout casser, quand plus rien ne marche, pas même un bon `reset` ou le chuchotement de mots doux à votre ordinateur. Ces techniques magiques ultimes ont fait leurs preuves, mais essayez tout de même de trouver des solutions autres car ce ne sont pas des pratiques très propres.

2.5.1 Git distant cassé mais local en bon état

Supprimez le dépôt distant et refaites-le à partir du dépôt local en bon état.

2.5.2 Git local cassé mais distant en bon état

Supprimez votre dépôt local et reclonez le distant.

2.5.3 Git local cassé sans dépôt distant

Copiez vos dossiers et fichiers ailleurs, faites un `reset` au dernier commit (`git reset --hard HEAD`), supprimez les fichiers dans votre dépôt local (mais PAS le dépôt local, c'est à dire conservez le dossier `.git`), remettez les dossiers et fichiers préalablement sauvés dans votre dépôt, puis normal `git add`, `git commit`.

2.5.4 Git distant cassé sans dépôt local

Il y a des jours où il vaut mieux faire des gaufres.

Voilà qui conclut notre découverte de Git, je n'ai plus qu'à vous souhaiter bonne chance. Vous en aurez besoin...