Débutant · Aucun prérequis

Environ 25 min de lecture

C'est quoi Git et GitHub, et à quoi ça sert ?

La réunion se termine. Quelqu'un demande : « on peut revenir à la version d'avant ? » Silence. Personne ne se souvient de ce qui a changé, ni qui l'a changé.

Git existe pour que ce moment n'arrive plus. C'est un carnet de bord automatique : il note chaque étape de ton travail, te laisse essayer sans rien casser, et te permet de revenir en arrière.

Ce cours part de zéro. À la fin, tu sauras lire un historique, créer une branche, fusionner deux versions — et comprendre ces mots que tout le monde emploie sans les expliquer : `commit`, `merge`, `fork`, `push`.

1. Le problème que Git résout

Ouvre le dossier où tu ranges tes documents. Tu y trouveras probablement ceci :

```
rapport.docx
rapport-final.docx
rapport-final-2.docx
rapport-final-2-vraiment-final.docx
rapport-final-2-vraiment-final-OK.docx
```

Chaque copie avait une bonne raison d'exister au moment où elle a été faite. Le résultat, lui, a trois défauts : on ne sait plus ce qui a changé entre deux versions, on ne sait pas *qui* a changé quoi, et on finit par garder la mauvaise.

Git règle ces trois problèmes. Au lieu de recopier le dossier entier à chaque fois, il note l'état de tes fichiers à l'instant où tu le lui demandes — et il ne stocke jamais deux fois un contenu identique. Chaque note porte un message, une date et un auteur. C'est un carnet de bord automatique.

> **À RETENIR**
>
> Git n'est pas un logiciel de sauvegarde. C'est un **enregistreur d'étapes**. La différence : une sauvegarde écrase, un historique empile. Tu peux revenir à n'importe quelle étape, sans rien perdre de celles d'après.

Git et GitHub, ce n'est pas la même chose

C'est la confusion la plus courante, et elle est légitime : les deux noms se ressemblent.

Git	GitHub, GitLab, Forgejo…
Un logiciel installé sur ton ordinateur. Il fonctionne même sans Internet.	Des sites web qui hébergent une copie de ton dépôt, pour la partager et la sauvegarder ailleurs.
Garde l'historique de ton travail.	Garde une copie de cet historique, accessible depuis n'importe où.

Autrement dit : tu peux utiliser Git toute ta vie sans jamais créer de compte nulle part. Et une plateforme sans Git ne sert à rien.

2. Le dépôt

Un **dépôt** (en anglais *repository*, souvent abrégé « repo »), c'est un dossier ordinaire auquel on a ajouté un carnet de bord. Rien de plus.

Ton dossier contient toujours tes fichiers, exactement comme avant. En plus, un sous-dossier caché nommé `.git` stocke tout l'historique. C'est lui qui fait la différence entre « un dossier » et « un dépôt ».

```
mon-projet/
|-- .git/          <- l'historique (caché, on n'y touche pas)
|-- index.html     <- tes fichiers, comme d'habitude
`-- notes.txt
```

Pour transformer un dossier en dépôt, une seule commande, à faire une fois :

```
git init
```

Il existe aussi une autre façon de partir : copier un dépôt qui existe déjà, avec `git clone`. On y revient à la section 10.

> **UNE ERREUR FRÉQUENTE**
>
> Ne supprime jamais le dossier `.git` en pensant faire du rangement. Tes fichiers resteraient, mais tout l'historique disparaîtrait d'un coup — c'est la seule façon simple de vraiment tout perdre avec Git.

Trois zones à connaître

Quand tu travailles, un fichier passe par trois états. Comprendre ces trois mots t'évitera 90 % des blocages de débutant :

1. **Modifié** — tu as changé le fichier, Git l'a remarqué mais n'a rien enregistré.
2. **Indexé** (*staged*) — tu as dit à Git : « celui-là, mets-le dans la prochaine photo ». C'est la salle d'attente.

3. **Enregistré** (*committed*) — l'étape est écrite dans l'historique. Elle est à l'abri.

Le mot « index » (ou « stage ») déroute souvent. Retiens l'image du photographe : avant de déclencher, tu rassembles les personnes qui doivent être sur la photo. `git add`, c'est faire entrer quelqu'un dans le cadre.

3. Le commit

Un **commit** est une étape enregistrée. Il contient trois choses :

ce qui a changé depuis l'étape précédente ;

un message que tu écris, pour te souvenir du pourquoi ;

un lien vers l'étape d'avant, ce qui forme une chaîne.

C'est ce dernier point qui rend Git puissant. Chaque étape connaît celle qui la précède. En remontant la chaîne, on peut reconstituer l'état exact du projet à n'importe quel moment de son histoire.

Créer un commit

```
git add notes.txt        # je mets le fichier dans la salle d'attente
git commit -m "Ajoute les notes de la réunion"
```

Le `-m` sert à écrire le message directement. Sans lui, Git ouvre un éditeur de texte — utile pour un long message, déroutant quand on débute.

Écrire un message utile

Un bon message décrit *ce que fait* le changement, pas ce que tu as fait de ta journée. Et il tient sur une ligne.

Message faible	Pourquoi
`modif`	Ne dit rien. Dans six mois, tu ne sauras pas ce qui a changé.
`fini`	« Fini » quoi ? Le projet n'est jamais fini.
`Ajoute la page de contact et corrige le numéro de téléphone`	Bon message : on sait quoi, sans ouvrir le diff.

ASTUCE QUI CHANGE LA VIE

Si tu as du mal à écrire un message clair, c'est souvent le signe que ta modification fait deux choses à la fois. Sépare-la en deux commits. Un commit = une idée.

CAS RÉEL

Voici un message de commit authentique, tiré d'un vrai projet :

```
fix(llm-deepseek-pmu): resolve the endpoint like the official route
```

Traduit : « correction, dans le module *llm-deepseek-pmu* : utiliser la même adresse de serveur que la route officielle ». Les équipes habituées à Git adoptent ce format — un type (`fix` pour une correction, `feat` pour une nouveauté), puis la partie du projet concernée — parce qu'il permet de retrouver un changement des mois plus tard.

Rien ne t'oblige à écrire en anglais, ni à suivre cette convention. Un message en français clair vaut mieux qu'un message anglais approximatif. L'essentiel : qu'on comprenne **quoi** et **où**, sans avoir à ouvrir le détail.

4. Relire l'histoire, revenir en arrière

La commande que tu utiliseras tous les jours :

```
git log --oneline
```

Elle affiche la liste des étapes, la plus récente en haut. Chaque ligne commence par un identifiant étrange, comme `bcc6798bb5` : c'est l'empreinte unique du commit. Elle est calculée à partir de son contenu, ce qui la rend impossible à falsifier sans s'en apercevoir.

```
$ git log --oneline
bcc6798bb5 Merge de la version amont
45978540dc Corrige l'adresse d'API du plugin
56253a0544 Retire un raccourci dangereux
```

Voir précisément ce qui a changé

```
git show 45978540dc     # le détail d'un commit précis
git diff                # ce que tu as modifié et pas encore enregistré
```

Ces deux commandes sont ton filet de sécurité : elles ne modifient rien, elles montrent. `git diff` juste avant un commit est une excellente habitude — c'est le moment où l'on repère une ligne oubliée, un mot de passe laissé dans le code, un fichier en trop.

Revenir en arrière

Deux situations très différentes, qu'on confond souvent :

Situation	Commande	Ce qui se passe
Tu viens de modifier un fichier et tu le regrettes — tu n'as rien enregistré.	`git restore fichier.txt`	Le fichier revient à sa dernière version enregistrée. Tes modifications non enregistrées sont perdues.

Situation	Commande	Ce qui se passe
Tu veux rester dans l'historique, mais regarder l'état d'avant.	`git checkout 4597854`	Tu consultes une étape passée. Rien n'est effacé — mais ne travaille pas ici, reviens sur une branche ensuite.

BONNE NOUVELLE

Un commit que tu crois avoir perdu est presque toujours récupérable. Git conserve les étapes pendant des semaines avant de les nettoyer, même si elles n'apparaissent plus à l'écran. Avant de paniquer, cherche — et demande de l'aide plutôt que de tout recommencer.

5. La branche

Voici la notion qui bloque le plus de débutants, alors qu'elle est simple.

Une branche n'est pas un dossier, ni une copie du projet. C'est une étiquette collée sur un commit. Rien de plus.

Quand tu enregistres une nouvelle étape, l'étiquette *avance* toute seule pour pointer sur cette nouvelle étape. C'est tout le mécanisme.

```
étapes    :  A --- B --- C
étiquette :  master          <- pointe sur C, la dernière étape enregistrée
```

Tu enregistres une étape D. L'étiquette suit :

```
étapes    :  A --- B --- C --- D
étiquette :  master               <- a avancé : elle pointe maintenant sur D
```

À quoi ça sert, concrètement

À travailler sur quelque chose **sans abîmer ce qui marche**. Tu veux essayer une nouvelle mise en page, mais ton site est en ligne et fonctionne ? Tu crées une branche, tu bricoles dessus autant que tu veux, et si le résultat ne te plaît pas, tu jettes la branche. Le site n'a jamais bougé.

```
git branch essai-mise-en-page     # créer l'étiquette
git switch essai-mise-en-page     # aller dessus
# ... tu travailles, tu commits ...
git switch master                 # revenir à la version stable
```

`git switch` (ou l'ancien `git checkout`, qui fait la même chose) déplace ton répertoire de travail sur la branche choisie. Tes fichiers changent sous tes yeux : Git remet le contenu tel qu'il était au moment de cette étiquette.

LE RÉFLEXE À PRENDRE

Une branche par sujet. « Je corrige le menu » et « je réécris la page contact », ce sont deux branches. Si tu mélanges tout sur une seule, tu ne pourras plus livrer l'un sans l'autre.

CAS RÉEL

Dans un projet suivi sérieusement, on trouve facilement une dizaine d'étiquettes qui cohabitent : une branche principale de travail dont le nom indique la version visée et sa date, une branche par chantier en cours, et des branches de secours dont le nom commence par `backup/` — créées automatiquement avant chaque opération risquée. Ce n'est pas du désordre : chaque étiquette a un rôle, et on sait laquelle ouvrir selon ce qu'on cherche.

MASTER OU MAIN ?

La branche principale ne porte pas toujours le même nom. Les dépôts anciens l'appellent `master` ; les plateformes récentes (GitHub, GitLab, Forgejo) créent par défaut une branche nommée `main`. Ce cours utilise `master` dans ses exemples, mais si tu crées un dépôt aujourd'hui, remplace simplement par `main` : le rôle est identique. Un `git switch master` qui échoue sur un dépôt récent vient presque toujours de là.

6. HEAD : « tu es ici »

Tu as maintenant plusieurs étiquettes dans ton dépôt. Comment Git sait sur laquelle tu travailles ? Grâce à un marqueur nommé **HEAD**.

HEAD, c'est la flèche « vous êtes ici » des centres commerciaux. Il indique la branche sur laquelle tu te trouves, et donc l'étape à partir de laquelle tes prochains commits s'enchaîneront.

```
étapes    :  A --- B --- C --- D
branche 1 :  essai   <- pointe sur C
branche 2 :  master  <- pointe sur D
HEAD      :  essai   <- la branche sur laquelle tu travailles en ce moment
```

Pour savoir où tu es, à tout moment :

```
git status
```

C'est la commande la plus utile de Git, et celle qu'on oublie d'utiliser. Elle répond à trois questions : sur quelle branche je suis, quels fichiers j'ai modifiés, quels fichiers sont prêts à être enregistrés.

> **LE « DETACHED HEAD »**
>
> Si tu vas consulter une étape ancienne directement (section 4), HEAD ne pointe plus sur une branche mais sur un commit précis. Git appelle ça un *HEAD détaché*. Ce n'est pas une erreur, mais ce n'est pas un endroit où travailler : reviens sur une branche avec `git switch` avant de reprendre du travail.

7. Les copies ailleurs

Jusqu'ici, tout se passe sur ta machine. Or un dépôt qui n'existe qu'à un seul endroit reste fragile : disque qui lâche, ordinateur volé, café renversé.

Git permet donc de déclarer des **dépôts distants** (en anglais *remotes*) : des copies de ton dépôt situées ailleurs, identifiées par une adresse et un surnom.

```
git remote -v
```

La convention veut que la copie principale s'appelle `origin`. C'est juste un surnom : tu peux le changer.

CAS RÉEL

Voici les deux distants d'un dépôt réel :

```
origin    https://github.com/exemple/projet.git      (fetch)
origin    https://github.com/exemple/projet.git      (push)
forgejo   http://mon-serveur.local/moi/projet.git    (fetch)
forgejo   http://mon-serveur.local/moi/projet.git    (push)
```

`origin` pointe vers le projet d'origine, chez quelqu'un d'autre : on y **lit** les nouveautés, on n'y écrit jamais. `forgejo` pointe vers un serveur personnel : c'est là que vont les sauvegardes. Deux rôles distincts, deux distants.

Pourquoi plusieurs copies plutôt qu'une

La source : le projet d'où vient ton travail. Tu y récupères les corrections des autres.

Ta sauvegarde : ton propre serveur ou compte. C'est là que ton travail est mis à l'abri.

Le partage : tes collègues lisent la même copie que toi.

Un dépôt peut n'avoir aucun distant (usage personnel), un seul, ou cinq. C'est toi qui décides.

8. Envoyer et recevoir

Trois commandes, souvent confondues. Elles font pourtant trois choses différentes.

Commande	Ce qu'elle fait	Ce qu'elle change chez toi
`git push`	Envoie tes commits vers le distant.	Rien : tes fichiers sont déjà à jour, on *envoie*.
`git fetch`	Va **regarder** ce qui existe en face, sans y toucher.	Rien du tout. C'est une simple visite.
`git pull`	`fetch` suivi d'un merge : récupérer *et* intégrer.	Tes fichiers peuvent changer : tu importes le travail de l'autre.

```
git push forgejo master      # envoie la branche « master » vers le distant « forg
git fetch origin             # va voir s'il y a du nouveau côté origin
git pull origin master       # récupère et intègre
```

LE BON RÉFLEXE

Utilise `git fetch` avant `git pull`. Cela te permet de voir ce qui arrive *avant* de le mélanger à ton travail. `git pull` fait le travail à ta place, ce qui est agréable — sauf quand ça se passe mal, parce que tu ne sais alors pas ce qui a été combiné.

UN PUSH QUI ÉCHOUE, CE N'EST PAS UNE PANNE

Si Git refuse ton `push` en disant que ta branche est « en retard », c'est une protection : quelqu'un a envoyé des choses que tu n'as pas encore. Récupère-les d'abord (`git pull`), puis renvoie. Git empêche ainsi d'écraser le travail des autres par accident.

9. Le merge

Un **merge** (« fusion ») consiste à recoudre ensemble deux fils de travail qui ont divergé.

Tu as une branche restée stable, et une branche d'essai qui contient trois nouvelles étapes. Tu veux récupérer ces trois étapes dans la branche stable : c'est un merge.

```
git switch master
git merge essai-mise-en-page
```

Deux cas, très différents

Cas	Ce qui se passe
Avance rapide (fast-forward)	La branche stable n'a pas bougé entre-temps. L'étiquette avance simplement jusqu'aux nouvelles étapes. Aucune complication : on appelle ça une avance rapide.
Vraie fusion	Les deux branches ont chacune de leur côté des étapes que l'autre n'a pas. Git crée alors une étape spéciale, qui a **deux parents**. C'est le commit de merge.

Le conflit : ce n'est pas une erreur

Quand les deux côtés ont modifié **les mêmes lignes du même fichier**, Git ne peut pas deviner qui a raison. Il s'arrête et te le dit : c'est un **conflit**.

Beaucoup de débutants croient avoir cassé quelque chose. C'est faux : Git fait exactement son travail en refusant de choisir à ta place. Il te laisse trancher, parce que toi seul sais quelle version garder.

Dans le fichier concerné, Git insère des repères. Voici ce qu'on voit réellement à l'écran :

```
<<<<<<< HEAD
le texte que TU avais écrit
=======
```

```
le texte que l'AUTRE avait écrit
>>>>>>> essai-mise-en-page
```

La première ligne s'appelle `HEAD` : c'est la version de la branche où tu te trouves. La dernière porte le nom de la branche qui arrive. Entre les deux, la ligne de signes égaux sépare les deux propositions.

Tu gardes ce qui est juste, tu supprimes les trois lignes de repères, tu enregistres, puis :

```
git add fichier-concerne.txt
git commit --no-edit
```

`--no-edit` évite l'ouverture de l'éditeur : Git a déjà préparé un message de fusion, on le garde tel quel.

> **CAS RÉEL**
>
> Une mise à jour de **882 commits** venue de l'extérieur a provoqué deux conflits. Pour le premier fichier, la version d'à côté était la bonne : elle a été conservée. Pour le second, aucun humain ne pouvait trancher proprement — un outil l'a reconstruit tout seul à partir des fichiers de configuration. Deux conflits, deux décisions différentes, et le travail a continué.

> **LA SÉCURITÉ AVANT TOUT**
>
> Avant toute fusion risquée, on crée une branche de secours. Si le résultat ne convient pas, on y revient en une commande. C'est d'ailleurs ce que fait automatiquement le script de synchronisation d'un projet bien outillé — d'où les étiquettes nommées `backup/…`. Une opération risquée sans filet est une opération qu'on regrette.

10. clone et fork

Deux façons de récupérer un projet qui existe déjà. Elles se ressemblent et n'ont pas le même usage.

`git clone` : copier chez soi

Prendre un dépôt existant et s'en faire une copie complète sur sa machine, historique compris.

```
git clone https://github.com/exemple/projet.git
```

Après cette commande, tu as le projet entier, toutes ses étapes passées, et le distant `origin` déjà configuré vers l'adresse d'origine. C'est la commande par laquelle presque tout le monde commence.

`fork` : copier chez soi, sur la plateforme

Un **fork** est une copie du projet créée *sur la plateforme* (GitHub, GitLab, Forgejo…), rattachée à **ton** compte. Elle t'appartient : tu peux y écrire autant que tu veux.

Le scénario typique : tu veux modifier un projet libre dont tu n'es pas propriétaire. Tu ne peux pas écrire chez son auteur, c'est normal. Tu crées donc un fork, tu travailles dedans, puis tu proposes tes changements au projet d'origine.

Action	Où ça se passe	Qui peut écrire
`clone`	Sur ton ordinateur	Toi, localement
`fork`	Sur la plateforme (site web)	Toi, sur ta copie en ligne

En pratique, les deux se combinent : on *fork* sur la plateforme, puis on *clone* son fork sur sa machine pour travailler.

CAS RÉEL

Dans le dépôt évoqué plus haut, il n'y a **pas** de fork : le projet d'origine est public et lisible, mais l'auteur n'a pas besoin d'y proposer de changements. Il a donc simplement *cloné* le dépôt chez lui et déclaré un second distant (`forgejo`) vers son propre serveur, pour y sauvegarder son travail. Aucune demande de permission, aucune dépendance à une plateforme tierce.

11. Les tags

Une étiquette de branche bouge : elle avance à chaque commit. Parfois, on veut au contraire marquer un point fixe. C'est le rôle du **tag**.

Un tag sert à nommer une étape importante — en général une version publiée :

```
git tag v1.0.0
git tag -a v1.0.1 -m "Corrige le calcul de la remise"
```

Contrairement à une branche, un tag **ne bouge jamais**. C'est ce qui permet de dire « la version 1.0.0, c'est exactement ces fichiers-là, pour toujours ».

Pour consulter une version étiquetée :

```
git checkout v1.0.0     # regarder ce qui était livré à ce moment-là
git switch master       # revenir au travail en cours
```

> CAS RÉEL
>
> Le projet dont je parlais est identifié par des tags comme `dsh-v0.1.6-alpha.2`. Le nom se lit de gauche à droite : le projet, un numéro de version, et le mot `alpha` qui signifie « version de test, pas encore stable ». Quand on te dit « on est passé en alpha.2 », cela veut dire qu'un tag a été posé sur une étape précise — et que cette étape ne changera plus.

Pourquoi les numéros de version ressemblent à 0.1.6

La convention la plus répandue se lit `MAJEUR.MINEUR.CORRECTIF` :

Correctif (0.1.**6**) — des réparations, rien de nouveau à apprendre.

Mineur (0.**1**.6) — des nouveautés, mais ton usage habituel continue de fonctionner.

Majeur (**0**.1.6) — des changements qui cassent la compatibilité. Un `0` en tête signale souvent un projet encore jeune, où tout peut encore bouger.

12. Les mots que tu entendras

Tu croiseras ces termes dans des conversations ou des tutoriels. Chacun est expliqué en trois lignes — assez pour ne plus être perdu, pas assez pour les maîtriser : c'est normal, ils s'apprennent par la pratique.

rebase

Une autre façon de combiner deux fils de travail. Au lieu de créer un commit de fusion à deux parents, le rebase *rejoue* tes commits par-dessus ceux des autres. L'historique final est plus lisible, mais tes commits changent d'empreinte. On l'évite sur des branches déjà partagées.

fast-forward

Le merge le plus simple : la branche n'a pas divergé, l'étiquette avance simplement. Aucun commit de fusion n'est créé.

cherry-pick

Prendre *un seul* commit d'une branche et l'appliquer ailleurs, sans rapatrier tout le reste. Utile pour récupérer une correction précise, sans le chantier qui l'entoure.

stash

Mettre ton travail en cours de côté, sans l'enregistrer, pour repartir d'une base propre — par exemple parce qu'on te demande une correction urgente ailleurs. `git stash pop` le ressort ensuite.

worktree

Ouvrir plusieurs dossiers de travail sur le même dépôt en même temps, chacun sur une branche différente. Fini le va-et-vient de `switch` quand deux tâches s'entremêlent.

force-push

Envoyer de force ses commits vers le distant, **en écrasant ce qui s'y trouve**. C'est la commande qui peut réellement détruire le travail de quelqu'un d'autre. À ne jamais utiliser sans savoir exactement pourquoi, et jamais seul sur un projet partagé.

pull request

Sur les plateformes : une demande de fusion. « Voici mes changements dans ma copie, voulez-vous les intégrer chez vous ? » Elle permet la relecture avant intégration. Ce n'est pas une commande Git, c'est une fonctionnalité de site.

13. Mémo : les commandes utiles

Douze commandes couvrent l'immense majorité de l'usage quotidien. Ce tableau est fait pour être relu, pas appris par cœur.

Commande	Ce qu'elle fait	Quand l'utiliser
`git status`	Où j'en suis : branche, fichiers modifiés, fichiers prêts	En permanence. C'est la boussole.
`git init`	Transforme un dossier en dépôt	Une fois, au tout début.
`git clone URL`	Copie un dépôt existant, historique compris	Pour récupérer un projet.
`git add fichier`	Met un fichier dans la prochaine étape	Après avoir modifié quelque chose.
`git commit -m "message"`	Enregistre l'étape	Quand la modification forme un tout cohérent.
`git log --oneline`	Liste les étapes	Pour retrouver un changement.
`git diff`	Montre ce qui a changé, sans rien enregistrer	Juste avant un commit, pour se relire.
`git switch branche`	Change de branche	Pour passer d'un sujet à l'autre.
`git branch nom`	Crée une branche sans y aller	Avant de démarrer un chantier.
`git merge branche`	Fusionne une branche dans la branche courante	Quand le chantier est terminé.
`git fetch`	Regarde ce qui existe en face, sans rien changer	Avant de récupérer, pour voir venir.
`git push`	Envoie tes commits vers le distant	Pour sauvegarder et partager.

CE QUE TU PEUX FAIRE DÈS MAINTENANT

Crée un dossier vide, lance `git init`, écris trois lignes dans un fichier, puis enchaîne `git status`, `git add`, `git commit` et `git log`. En cinq minutes, tu auras manipulé l'essentiel de ce cours. Le reste viendra en s'en servant.

14. GitHub : à quoi ça sert, et quelle différence avec Git ?

Tu as maintenant les commandes. Reste la question que tout le monde se pose à ce moment-là : « et GitHub, dans tout ça ? »

GitHub est un **site web** qui héberge ton dépôt. Tu y envoies tes commits, et ils y restent : accessibles depuis n'importe quel ordinateur, visibles par qui tu veux.

Ce qu'on y fait, concrètement

Sauvegarder ailleurs. Ton travail n'existe plus seulement sur ton disque.

Montrer. Un lien suffit pour que quelqu'un voie ton projet.

Travailler à plusieurs. Chacun envoie ses étapes, le site les rassemble.

Proposer des corrections. C'est là que vivent les *pull requests* dont on parlait plus haut.

Comment on y met son travail

```
# une seule fois : créer un dépôt vide sur le site, puis relier ton dépôt local
git remote add origin https://github.com/ton-compte/ton-projet.git

# ensuite, chaque fois que tu veux publier
git push origin main
```

C'est tout. Le reste — créer un compte, cliquer sur les boutons, comprendre les menus — s'apprend sur le site lui-même en cinq minutes, parce que c'est de l'interface, pas de la technique.

> LA DISTINCTION À RETENIR
>
> **Git** est le logiciel installé sur ta machine. Il fonctionne sans Internet.
> **GitHub** est un site où l'on range une copie. Il a besoin d'Internet, et il n'est pas le seul : GitLab, Bitbucket ou Forgejo font exactement la même chose.

Pourquoi les deux noms se confondent

Parce qu'ils ont été conçus à la même époque, que les noms se ressemblent, et que la plupart des gens entendent parler de GitHub *avant* de savoir que Git existe. On dit « mets-le sur GitHub » comme on dit « passe-moi du scotch » : la marque a remplacé le nom de l'objet.

Ce n'est pas grave. Ce qui compte, c'est de savoir que l'un est un outil et l'autre un endroit.

CAS RÉEL

Le projet dont je parlais plus haut est hébergé sur GitHub, mais son auteur ne s'en sert que pour *lire* : il y récupère les mises à jour d'un logiciel et n'y écrit jamais. Ses propres sauvegardes partent vers un serveur personnel. Deux distants, deux usages — et aucun compte sur le site n'a été nécessaire pour que Git fonctionne.

15. Questions fréquentes

Les questions que tout le monde se pose en commençant. Clique sur une question pour voir la réponse.

À trois choses qui se tiennent. **Revenir en arrière** : chaque étape enregistrée reste accessible — on compare, on restaure, on annule. **Savoir qui a changé quoi** : chaque étape porte un auteur, une date et un message. **Travailler à plusieurs sans s'écraser** : chacun avance sur son fil, puis on recoud.

Et une quatrième, qu'on oublie souvent : mettre son travail ailleurs que sur sa propre machine. Un dépôt qui n'existe qu'à un seul endroit reste fragile.

Parce qu'une copie ne dit rien. `rapport-final-2.docx` ne dit pas ce qui a changé depuis la veille, ni qui l'a modifié, ni pourquoi. Il faut ouvrir les deux fichiers et comparer à l'œil — quand on se souvient encore duquel est lequel.

Git répond à ces trois questions sans effort, et il ne recopie jamais deux fois un contenu identique. Ton dossier ne gonfle pas, et tu retrouves n'importe quelle version en une commande.

Par trois commandes, et rien d'autre : `git init`, `git add`, `git commit`. Crée un dossier vide, écris trois lignes dans un fichier, fais l'aller-retour. Cinq minutes, et tu auras manipulé l'essentiel.

Le reste — branches, merges, distants — s'apprend quand le besoin apparaît vraiment. Vouloir tout maîtriser d'avance est la meilleure façon de ne jamais commencer. Le mémo en fin de page liste ce qu'il faut savoir relire.

Git est le logiciel installé sur ta machine : il garde l'historique, et il fonctionne sans Internet. GitHub, GitLab ou Forgejo sont des *sites* qui hébergent une copie de ton dépôt en ligne, pour la partager et la sauvegarder ailleurs. On peut utiliser Git toute sa vie sans jamais créer de compte.

Presque jamais. Git est conçu pour que l'erreur soit réparable : un commit que tu crois perdu reste récupérable pendant des semaines. Les deux seules choses réellement dangereuses sont la suppression du dossier `.git` et le `force-push` sur un projet partagé. En dehors de ces cas, tu peux expérimenter sans crainte.

Non, et c'est même le contraire qui marche. Trois commandes suffisent pour démarrer : `git init`, `git add`, `git commit`. Les branches, les merges et les distants s'apprennent quand le besoin apparaît réellement. Vouloir tout maîtriser d'avance décourage plus de gens que ça n'en aide.

Non. Git sert partout où un texte évolue et où l'on veut garder trace : un livre, un mémoire, des fiches de cours, des traductions, un dossier administratif. Le fait que ce soit un outil très répandu dans l'informatique ne limite pas son usage. Ce qui compte, c'est d'avoir besoin de revenir en arrière.

C'est le comportement normal. `git commit` enregistre **localement**, sur ta machine. Pour que la copie en ligne reçoive tes étapes, il faut un second geste : `git push`. Le commit et l'envoi sont deux actions distinctes, volontairement séparées — cela permet d'enregistrer souvent, et de n'envoyer que ce qui est prêt.

Non, ce n'est pas une erreur. Cela signifie que tu consultes une étape précise plutôt que de te trouver sur une branche. C'est parfaitement normal pour regarder le passé. Simplement, ne travaille pas à cet endroit : reviens d'abord sur une branche avec `git switch nom-de-branche`.

`git pull` **`git fetch`**

`git fetch` va regarder ce qui existe en face, sans rien changer chez toi : c'est une visite. `git pull` fait la visite *et* intègre les nouveautés dans ton travail. Le second est plus rapide, le premier est plus prudent : il te laisse voir ce qui arrive avant de le mélanger à ce que tu fais.

Cela dépend de ce que tu veux obtenir, et c'est important de faire la différence. Si tu veux **garder** tes fichiers mais annuler l'enregistrement, `git reset HEAD~1` revient d'une étape en laissant tes modifications en place. Si tu veux annuler **un commit déjà envoyé** sans réécrire l'historique, on crée un nouveau commit qui fait l'inverse : `git revert`. En cas de doute, montre la situation à quelqu'un avant d'agir — c'est réparable, mais autant éviter.

Non. Tout le travail local — enregistrer, comparer, changer de branche, fusionner — fonctionne hors ligne. Internet ne devient utile que pour `push`, `fetch` et `clone`, c'est-à-dire tout ce qui touche à une copie distante.

Ce n'est pas une panne, c'est une protection. Quelqu'un a envoyé des étapes que tu n'as pas encore récupérées. Si Git acceptait ton envoi, ces étapes seraient écrasées. La solution est de récupérer d'abord (`git pull`), puis d'envoyer de nouveau.

Non, et c'est le malentendu qui coûte le plus de temps aux débutants. Une branche est une **étiquette** posée sur une étape. Elle ne duplique rien. Quand tu changes de branche, Git remplace simplement le contenu de ton dossier par celui correspondant à l'étiquette choisie — instantanément, et sans consommer d'espace supplémentaire.

Une demande de fusion, propre aux plateformes d'hébergement. Tu as travaillé dans ta copie ; tu proposes d'intégrer tes changements au projet d'origine. Elle permet à d'autres de relire avant d'accepter. Ce n'est pas une commande Git mais une fonctionnalité de site — d'où le nom, qui varie selon les plateformes.

Non. Six ou sept commandes couvrent l'usage quotidien, et elles s'ancreront d'elles-mêmes à force de servir. Le reste se consulte au besoin. Relire ce mémo trois fois vaut mieux que de le réciter.

Oui, si son contenu est déjà intégré ailleurs. `git branch -d nom` la supprime en refusant si des étapes n'ont pas été fusionnées — c'est une sécurité. Une branche supprimée par erreur reste récupérable un moment. En revanche, réfléchis à deux fois avant de supprimer un *dépôt* entier : lui ne revient pas.

16. Quiz : as-tu tout suivi ?

Douze questions pour vérifier ta compréhension. Clique sur la réponse qui te semble juste : la correction s'affiche immédiatement, avec l'explication. Tu peux recommencer autant de fois que tu veux.

1. **Une branche, c'est…**

> Une branche ne duplique rien : c'est une étiquette qui désigne une étape. C'est justement pour cela que changer de branche est instantané, même sur un projet énorme.

2. **Après un `git commit`, ton travail est…**

> Le commit enregistre localement. C'est `git push` qui envoie ensuite vers une copie distante. Les deux gestes sont séparés exprès.

3. **La commande qui répond à « où j'en suis, là, maintenant ? », c'est…**

> `git status` donne la branche courante, les fichiers modifiés et ceux prêts à être enregistrés. C'est la commande la plus utile au quotidien, et celle qu'on oublie le plus d'utiliser.

4. **Un conflit de fusion signifie que…**

> Un conflit est un comportement normal et attendu. Git refuse de choisir à ta place, parce que toi seul sais quelle version doit être conservée.

5. **`git fetch` et `git pull` ...**

> `fetch` est une visite, sans conséquence. `pull` récupère *et* fusionne. Envoyer son travail, c'est `push`.

6. **Un `fork`, c'est...**

> La copie faite sur ton ordinateur, c'est un `clone`. Le fork se passe sur le site web : il crée une version du projet qui t'appartient, où tu peux écrire librement.

7. **Gérer l'historique avec Git nécessite-t-il Internet ?**

> Git est un logiciel local. Internet ne sert qu'aux échanges avec une copie distante : `clone`, `fetch`, `pull`, `push`.

8. **Un *tag* se distingue d'une branche parce que...**

> Une branche avance à chaque commit. Un tag reste collé à une étape précise, pour toujours — c'est ce qui permet de dire « la version 1.0, c'est celle-là, exactement ».

9. **Ton envoi est refusé car « ta branche est en retard ». Que fais-tu ?**

> Le refus est une protection contre l'écrasement du travail d'autrui. Un envoi forcé détruirait justement ce qu'on cherche à préserver : c'est la commande à éviter absolument.

10. `git diff` sert à…

> `git diff` est en lecture seule : il montre. C'est un excellent réflexe juste avant de valider une étape, pour se relire — et repérer une ligne oubliée ou un mot de passe laissé dans le code.

11. Le dépôt d'un projet, c'est…

> Le dépôt, c'est ton dossier de travail *plus* l'historique qu'il contient. Ce qui ajoute la mémoire au dossier, c'est la présence de `.git`. Un dépôt sans copie en ligne est parfaitement valide.

12. Que fait `git add fichier.txt` ?

> `git add` remplit la salle d'attente : il annonce ce qui figurera dans le prochain commit. L'enregistrement lui-même a lieu au moment de `git commit`. D'où l'image du photographe : on rassemble d'abord les personnes, on déclenche ensuite.

Ce quiz corrige immédiatement chaque réponse. Aucune donnée n'est enregistrée ni transmise : tout se passe dans ton navigateur.

Cours rédigé pour Aide IA. Les exemples proviennent de dépôts réels et ont été vérifiés. Une erreur, une imprécision, une question restée sans réponse ? Écris-nous : ce cours s'améliore avec les retours.