SpaceFox

CC BY

Zeste de savoir

Git : la GUI est votre amie !

10 novembre 2022

Table des matières

Introduction

Si vous utiliser Git, il est probable qu'un jour vous vous soyez posé une question existentielle quant à la manière de réaliser telle ou telle manipulation. Et il est tout aussi probable qu'au cours de vos recherches vous soyez tombés sur une réponse du type:

> Facile, il suffit de lancer `git bidule --option` en ligne de commande.

Souvent sans plus d'information. Parfois ça résout le problème, parfois non, et dans bien des cas vous n'avez tout simplement pas compris ce que vous venez de faire.

Le souci ici, c'est que beaucoup d'utilisateurs de Git, en particulier d'utilisateurs avancés (et donc susceptibles d'aider sur Internet) sont des aficionados de la ligne de commande et sont persuadés que celle-ci est *nécessaire* pour utiliser Git, voire parfois *indispensable* pour bien le comprendre.

Je suis quant à moi convaincu que **passer par une interface graphique aide beaucoup à comprendre et utiliser Git**, et je vais vous expliquer pourquoi.

Oh, et un point de vocabulaire:

— CLI = interface en ligne de commandes (*Command Line Interface* dans la langue de Iron Maiden)
— GUI = interface graphique (*Graphical User Interface* dans la langue de Tool)

1. Mon expérience avec Git

Un peu d'historique pour que vous compreniez pourquoi je vais tenir cette position. J'ai réellement commencé à utiliser Git assez tard, en 2014 avec… la création de Zeste de Savoir.

Un an plus tard, dans mon entreprise, j'avais démontré qu'en passant de SVN à Git, on passait d'une demi-journée de *checkout* à l'initialisation du projet géant à moins d'une demi-heure, sans perdre l'historique. J'étais propulsé «référent Git» et chargé de mettre en place la migration SVN → Git sur les projets qui en avaient besoin et sur les nouveaux dépôts. Ceci incluait la formation initiale et le suivi des équipes.

Rebelote à partir de 2018, où ma nouvelle entreprise voulait passer à Git (et avait commencé à le faire), mais était perdue par manque de connaissance des outils.

Donc, ce que je vais vous présenter ensuite n'est pas une pure théorie sortie de mon chapeau. C'est quelque chose qui a servi à des dizaines de développeurs professionnels qui utilisent Git au quotidien maintenant. C'est quelque chose que j'ai *expérimenté* et qui fonctionne avec le plus grand nombre, même si comme toute méthode elle a connu son petit lot d'exceptions.

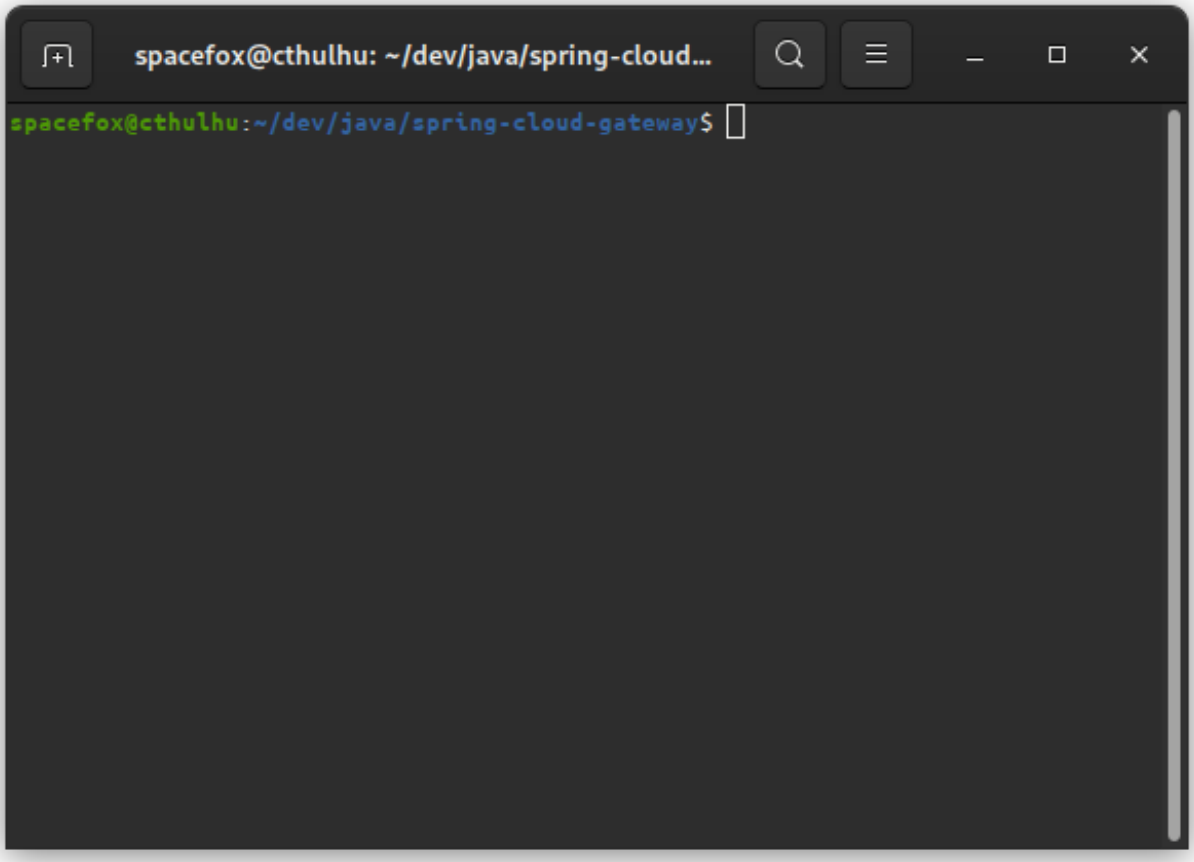
D'autre part, je ne suis pas anti-CLI dans l'absolu. Au contraire, je fais toute mon administration en ligne de commandes (celle de mon poste, celle de mes serveurs personnels et celle des serveurs dont j'ai la responsabilité dans mon entreprise). Mais ça n'implique pas que j'utilise la ligne de commande pour tout et n'importe quoi: elle est plus pratique *dans certains contextes* et non *dans l'absolu*.

2. Les problèmes avec la CLI

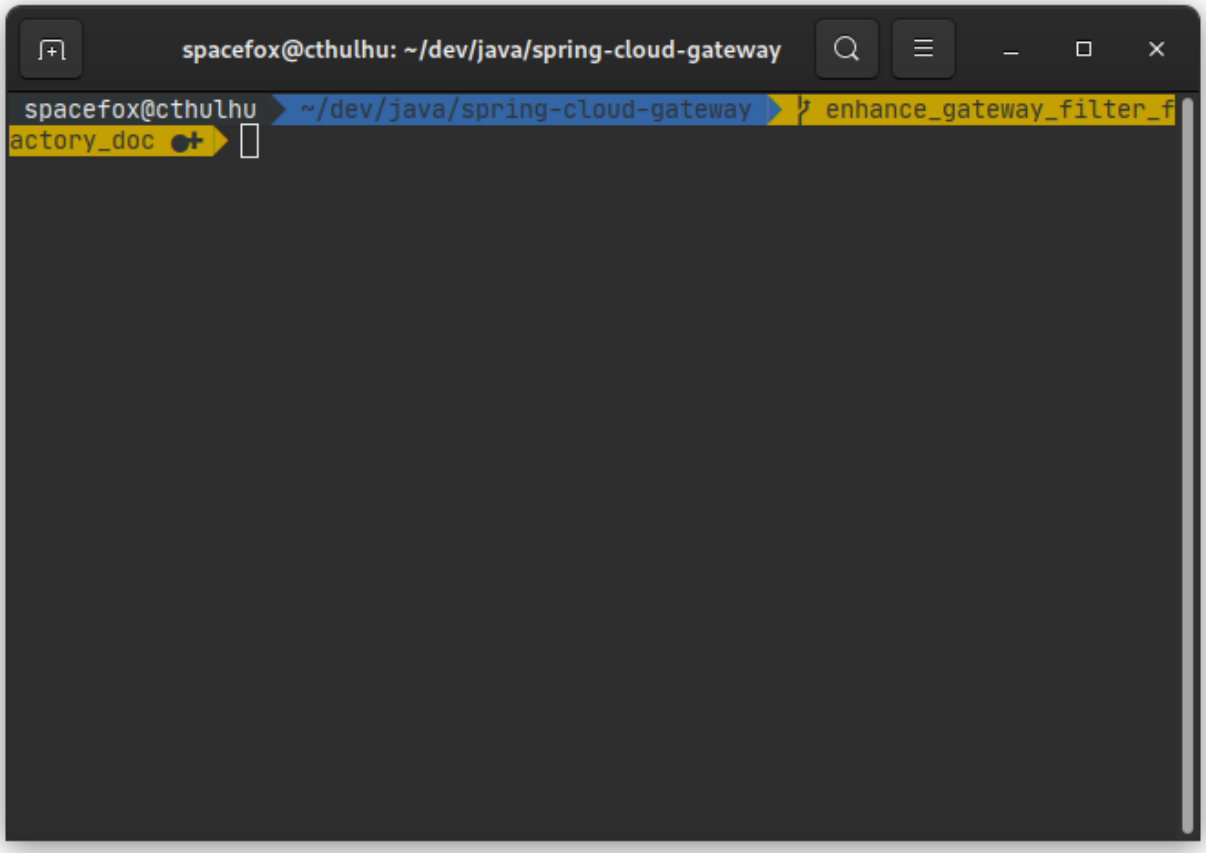
2.1. Un manque d'information

Le problème d'une interface en ligne de commandes, c'est que ça manque de contexte. Genre, beaucoup.

Admettons pour l'exemple que je viens de préparer une évolution sur la Gateway de Spring-Cloud, et que j'aie des retours sur ma *merge request*. Je fais donc mes petites modifications et je veux les commiter. En arrivant sur la ligne de commande, j'ai précisément ceci:



Si je suis un développeur amateur de ligne de commande, j'ai peut-être configuré mon terminal avec `zsh` et un outil pour afficher directement des informations Git dans l'invite de commande, auquel cas j'aurai quelque chose comme ceci:



On a déjà un peu plus d'informations, mais ce n'est pas folichon. En particulier, sans rien faire, je ne sais pas:

— Quels sont les fichiers ajoutés (qu'ils soient ajoutés à l'index Git ou non), modifiés ou supprimés.
— Quel est l'état des branches distantes. Par exemple, aucune de ces captures d'écran ne montre que j'ai *déjà fetch* les dépôts distants, que `master` a avancé et donc que j'ai probablement du *rebase* à prévoir.
— Quelles sont les différences entre les versions commitées et non-commitées à l'intérieur de chaque fichier.
— À quoi ressemble l'arbre de commits, utile pour savoir ce qu'il s'est passé sur le dépôt et surtout quasiment indispensable pour comprendre certaines opérations comme *rebase*.
— Quelles sont mes branches locales (et les branches distantes, mais l'intérêt est souvent moins immédiat).

Et à moins d'avoir spécialement configuré mon terminal, je ne sais pas non plus:

— Quelle est la branche courante.
— S'il y a un dépôt Git dans le chemin en cours.
— S'il y a des fichiers non comités dans ce dépôt.

2.2. Une double connaissance est requise pour aller plus loin

L'autre souci de la CLI, c'est que son utilisation nécessite un double connaissance. L'utilisateur doit savoir:

1. Ce qu'il veut faire fonctionnellement.
2. La syntaxe *exacte* de la commande qui permet d'y arriver.

Or, retenir une syntaxe est généralement plus compliqué que de retenir l'emplacement d'un bouton dans une interface graphique. Et si certaines commandes sont rapides à taper, d'autres n'ont pas cet avantage. C'est d'autant plus complexe avec Git, qui a généralement plusieurs commandes pour arriver au même résultat, avec souvent deux ensembles: les commandes *plumbing* d'assez bas niveau surtout destinées aux scripts, et les commandes *porcelain* destinées plutôt aux humains.

Pire: contrairement à une interface graphique, il ne suffit pas de regarder un peu partout pour espérer trouver si on ne connait pas la syntaxe. Il faut soit aller chercher la documentation, soit de l'aide (sur Internet ou auprès de collègues).

Enfin, la CLI de Git permet par essence de réaliser *toutes* les opérations possibles et imaginables. Mais notez bien qu'elle ne permet **pas** de réaliser **simplement** toutes ces opérations – et n'en a d'ailleurs pas la prétention.

2.3. Les conséquences pratiques

Les conséquences de tout ça, c'est que si la CLI de Git permet littéralement de tout faire, son ergonomie et ses contraintes impliquent:

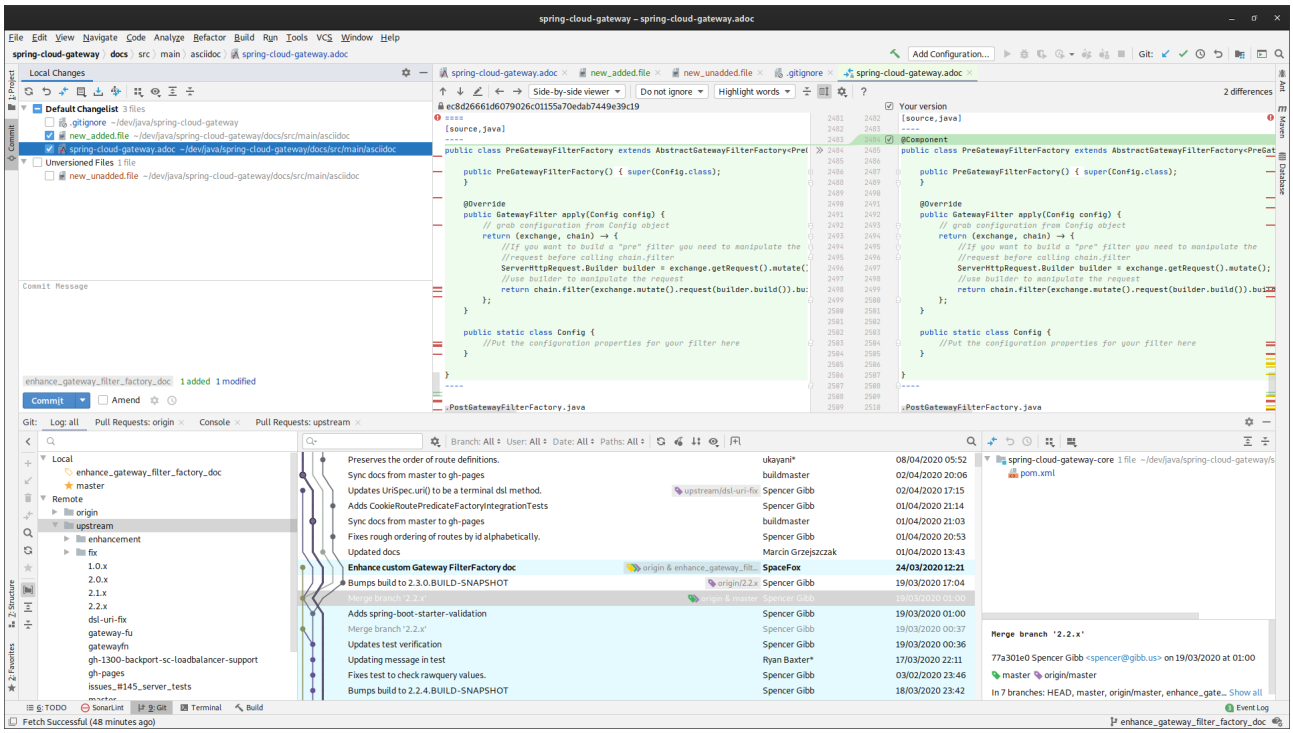
- Un risque accru d'erreurs, principalement par méconnaissance de ce que fait réellement une commande.
- Une perte de temps:
  - Par des opérations rendues complexes par la ligne de commande.
  - À cause du manque d'informations, qui ne permet pas de vérifier en temps réel ce qu'on fait.

3. Pourquoi une interface graphique est l'amie de Git

3.1. Tout, d'un coup d'œil, à portée d'un clic (ou deux)

Le premier avantage d'une interface graphique pour utiliser Git, c'est qu'on a toutes les informations disponibles en même temps, ou au pire à un clic.

Si je reprends mon exemple de la section précédente et que cette fois j'utilise le client Git intégré à IntelliJ 2020.1, j'obtiens ceci:

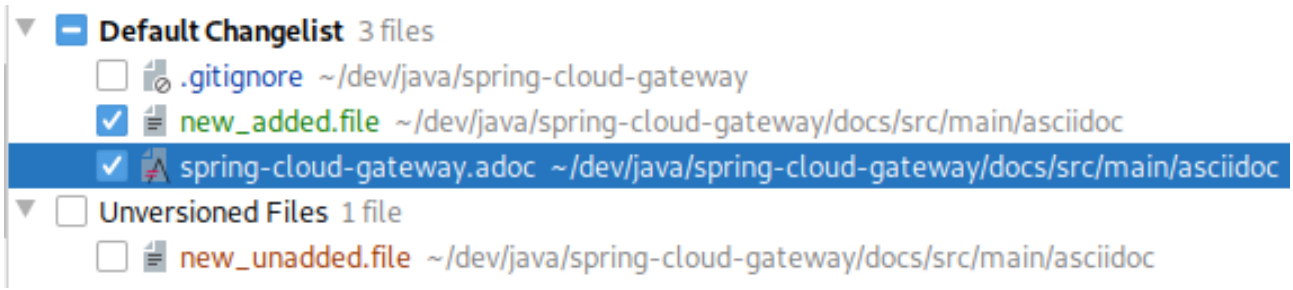


OK, c'est un peu chargé, mais on va détailler. Ici je suis sur un client en particulier, mais n'importe quel bon client graphique Git présentera aussi toutes ces informations, quoique pas exactement sous cette forme.

Je reprends la liste des «informations que je n'ai pas» de la section précédente.

3.1.1. Mes fichiers ajoutés/modifiés/supprimés

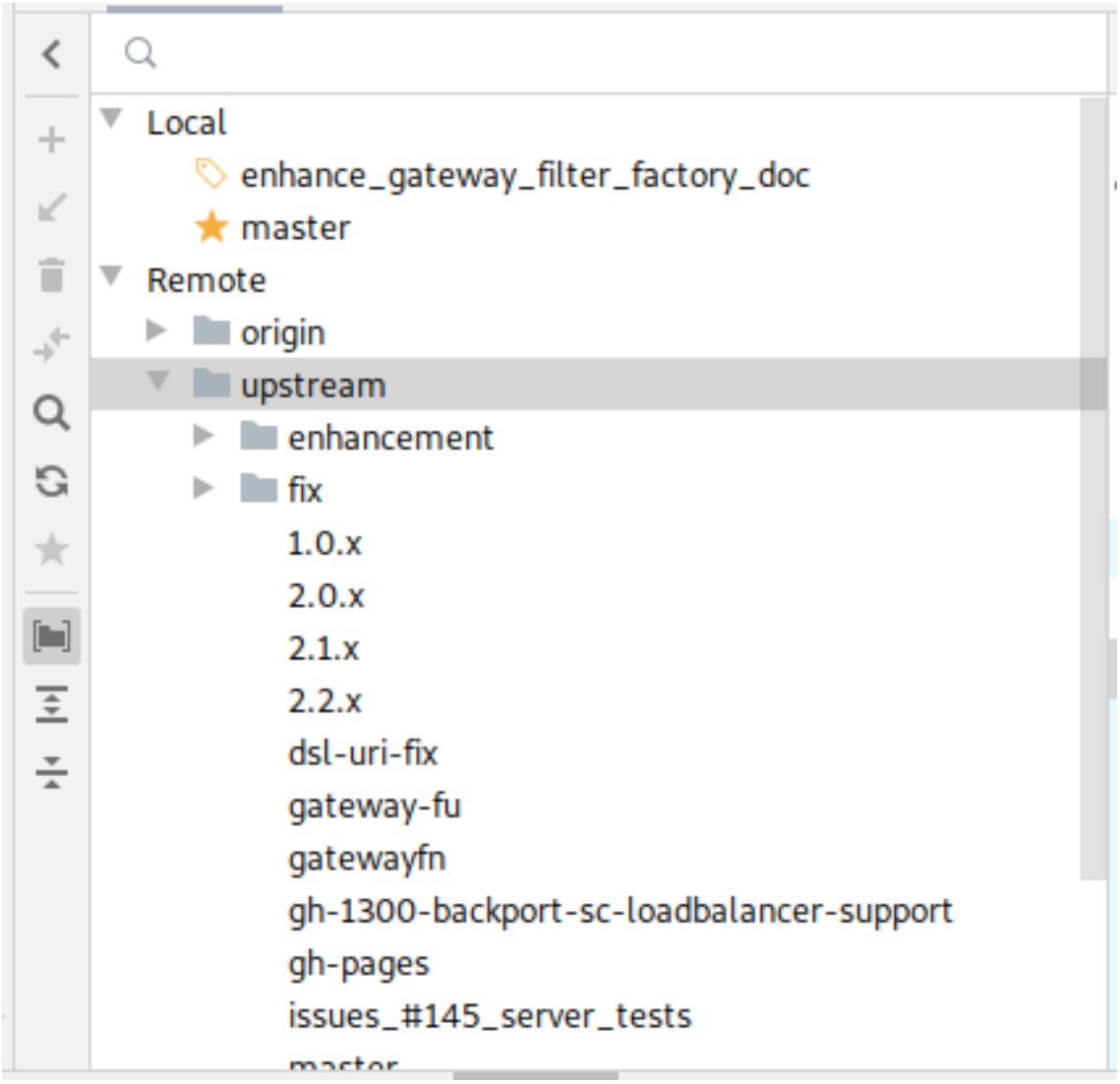
Ils sont présentés ici:



Pour avoir l'information en CLI, il aurait fallu taper:

```
1 $ git status
```

3.1.2. Les branches locales et distantes



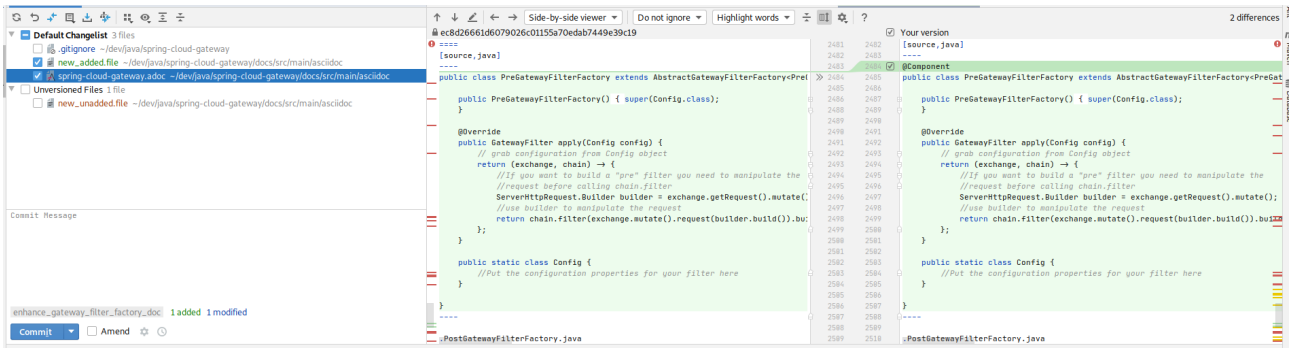
Pour avoir l'information en CLI, il aurait fallu taper:

```
1 $ git branch -a
```

3.1.3. Les différences entre les versions commitées et non-commitées de chaque fichier

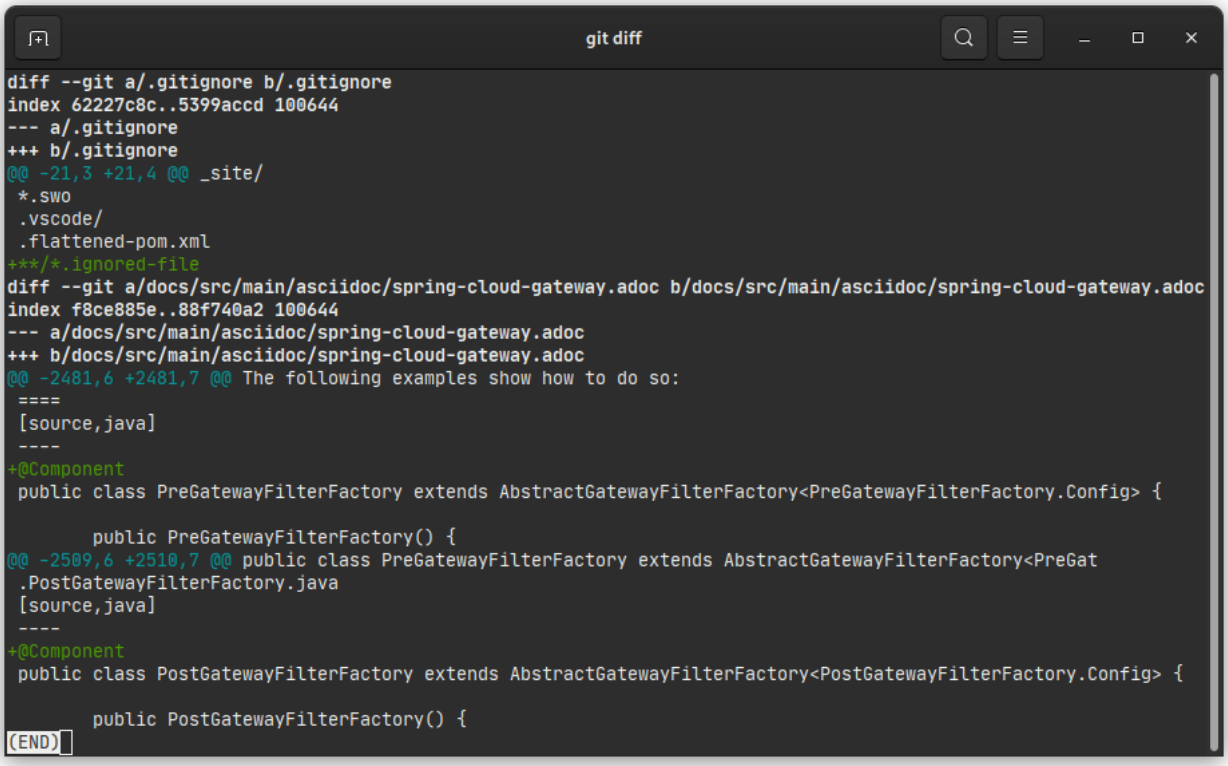
Ici il me suffit de cliquer sur un fichier de la liste pour voir les différences:

3. Pourquoi une interface graphique est l'amie de Git



Pour avoir l'information en CLI, il aurait fallu taper:

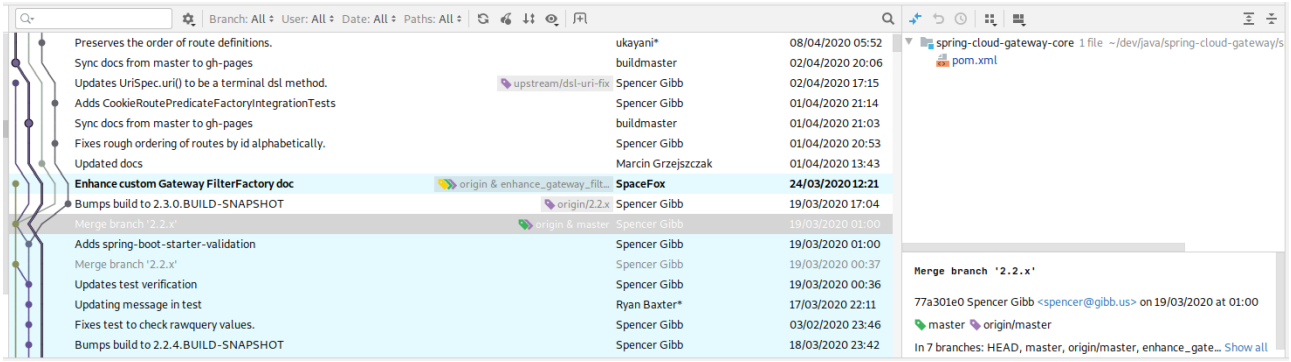
```
$ # Donne le diff d'un seul fichier, il faut taper son chemin :
$ git diff docs/src/main/asciidoc/spring-cloud-gateway.adoc
$ # Donne le diff de tout le commit, avec une ergonomie que je
    vous laisse juger
$ git diff
```



Je ne sais pas vous, mais moi je comprends beaucoup mieux le *diff* de l'interface graphique que celui de la CLI.

3.1.4. L'arbre de commits

Non seulement il est affiché de façon lisible, mais en plus il suffit de cliquer sur un commit pour savoir quels sont les fichiers impactés, et de double-cliquer sur un fichier pour avoir le *diff* exact.



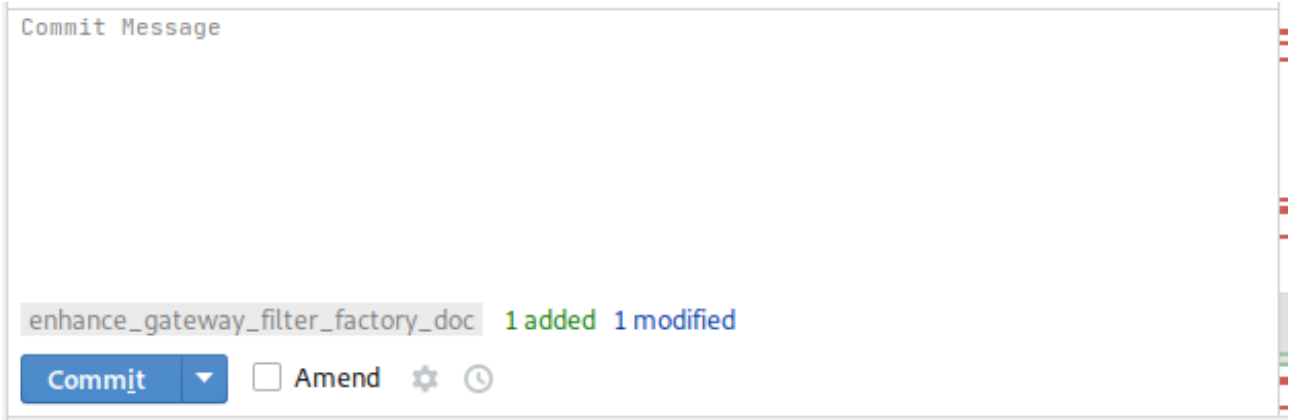
Pour avoir l'information en CLI, il aurait fallu taper:

```
$ git log --graph --oneline --all # Pour l'arbre seulement
$ git show --name-only 77a301e0 # Pour la liste des fichiers
    impactés par un commit
```

On remarque que lister les fichiers impactés par un commit nécessite de connaitre l'identifiant de ce dernier.

3.1.5. La branche courante

Elle est indiquée en bas de la zone de saisie du message de *commit*, avec en prime le résumé des fichiers qui vont être commités.



Pour avoir l'information en CLI, il aurait fallu taper:

```
$ git diff docs/src/main/asciidoc/spring-cloud-gateway.adoc
$ git show-branch                  # Renvoie le nom de la branche courante, entre autres
$ git rev-parse --abbrev-ref HEAD # Renvoie uniquement le nom de la branche courante, la syntaxe n'est pas intuitive
$ git branch --show-current        # Référencé sur Internet mais ne fonctionne pas avec Git 2.20.1
```

Et surtout, j'ai **toutes** ces informations **sur le même écran**, sans avoir besoin de *penser* à les chercher!

3.2. Les actions, mêmes complexes, à portée d'un ou deux clics

Je ne détaillerai pas plus que ça parce que les actions possibles dépendent du client graphique exact que vous utilisez.

Mais quel que soit votre GUI, les principales actions Git seront disponibles en un ou deux clics – ou un ou deux raccourcis claviers, si vous préférez ce mode d'entrée. Je pense notamment, mais pas que, à:

— Sélectionner/désélectionner un fichier pour le commiter.
— Commiter (dont «modifier un commit» avec *amend*)
— Pousser sur un serveur
— Récupérer les informations du serveur
— Merger une branche sur une autre
— *Rebase*
— *Squash*

Les bonnes interfaces graphiques permettent aussi de faire facilement des actions pénibles à réaliser avec la CLI. Celle que j'utilise le plus, c'est le commit partiel d'un fichier. Si les versions de Git récentes ont `git commit --interactive`, les clients graphiques ont tout simplement des cases à cocher devant les morceaux modifiés.

Enfin, la plupart des interfaces graphiques ont des fonctions pratiques, comme le paramétrage de l'affichage par défaut de telle ou telle partie de l'interface, les recherches locales, une bonne interface de résolution de conflit en cas de *rebase* ou *merge*, etc. qui simplifient la vie quotidienne du développeur.

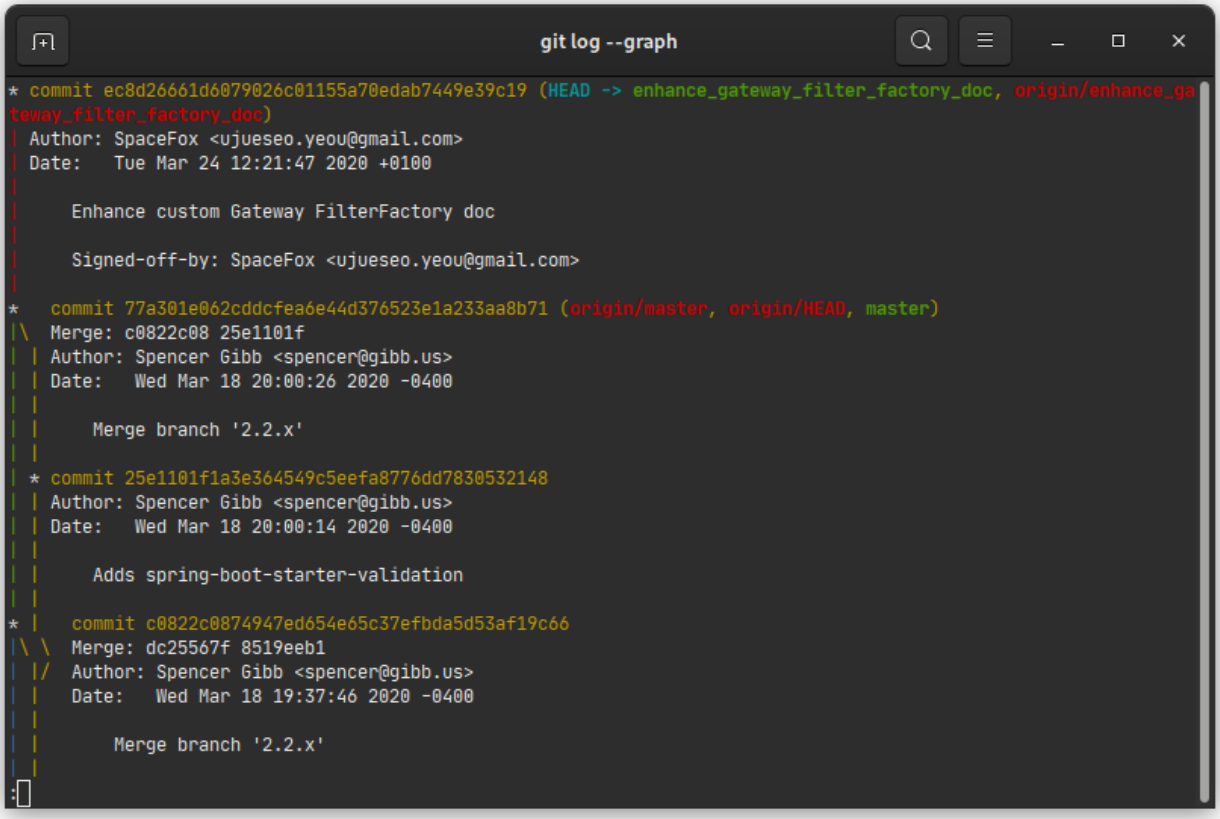
4. La GUI pour comprendre les opérations sur les commits

Un autre avantage d'une interface graphique – et en particulier d'avoir une vue **claire** de l'arbre des commits – c'est qu'elle permet de simplifier la compréhension de certaines opérations, en particulier les manipulations de cet arbre telles que `merge`, `rebase`, `squash`…

4. La GUI pour comprendre les opérations sur les commits

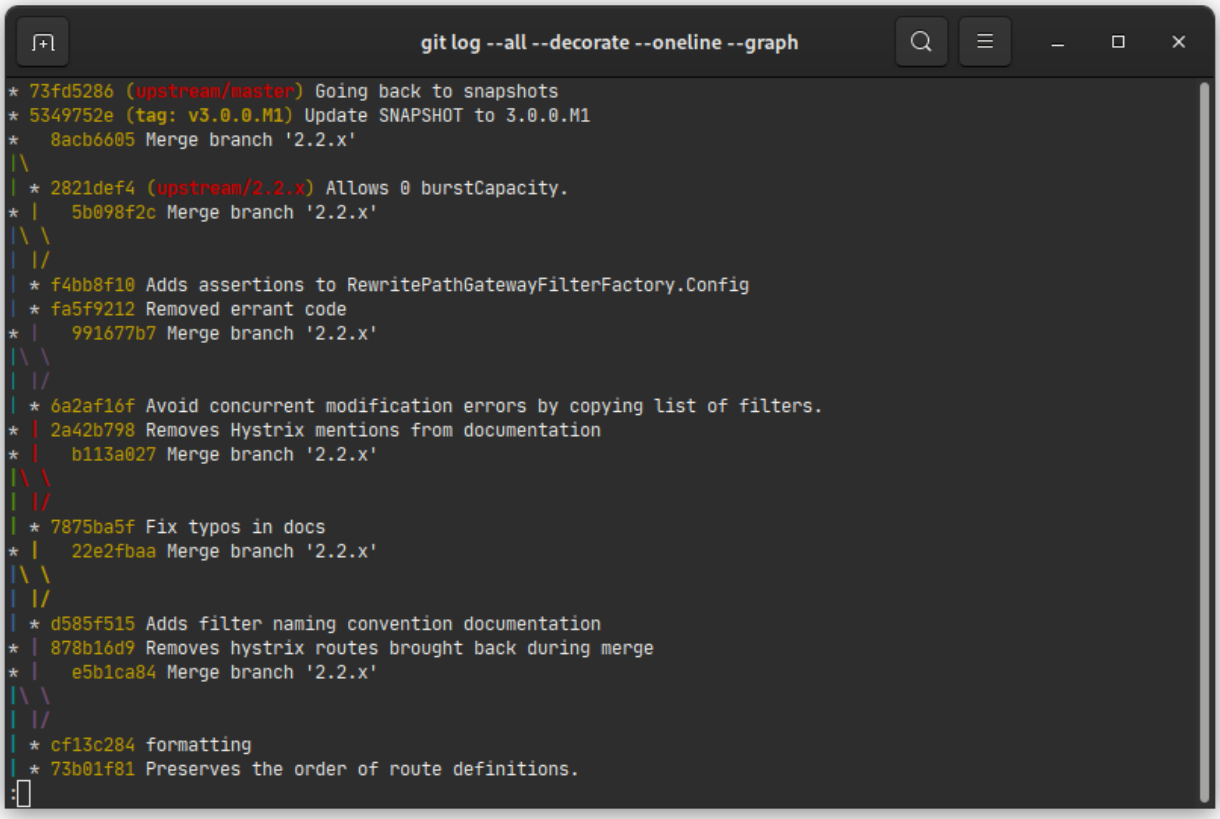
J'insiste sur la vue claire de l'arbre des commits: la CLI permet une visualisation de cet arbre, mais l'affichage par défaut est chiche en informations:

```
1 $ git log --graph
```

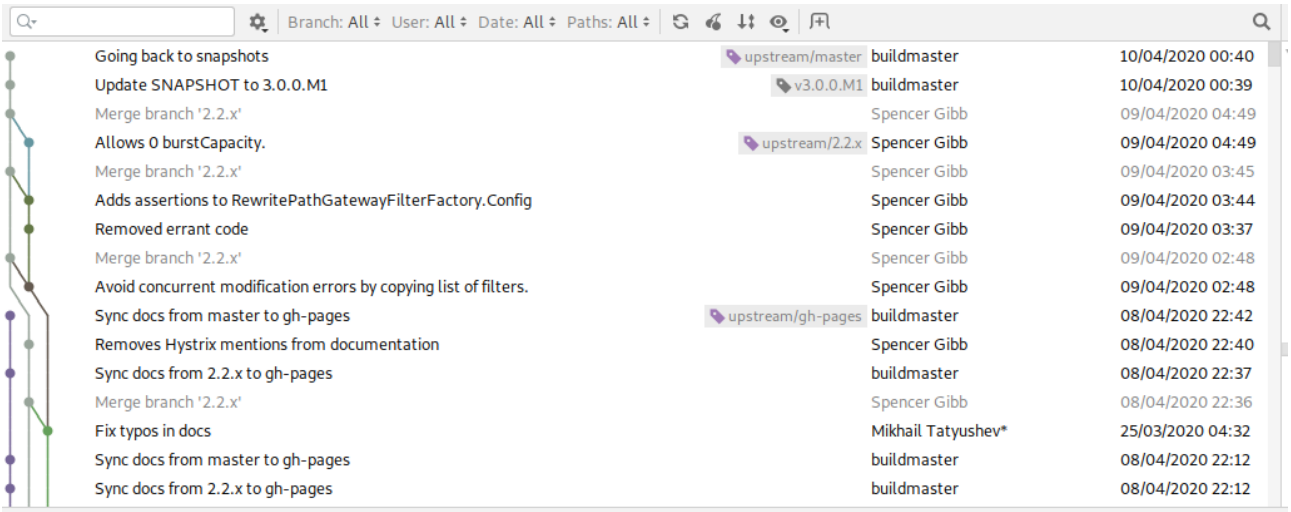


Une version plus complète de la commande permet d'avoir un affichage plus complet et compact:

```
1 git log --all --decorate --oneline --graph
```



Mais force est de constater qu'un affichage graphique du même arbre semble plus lisible et complet:



Mon expérience, partagée par beaucoup de développeurs que j'ai vu apprendre Git, est que ce type d'interface est **nécessaire** pour une bonne compréhension et un usage efficace de ces opérations, en particulier de `rebase`.

C'est d'autant plus vrai que la doc officielle de Git est à la fois très dense (elle veut couvrir beaucoup de cas) et, pour la version française, utilise des choix de traduction parfois étonnants qui obligent à apprendre tout le vocabulaire en double (la version anglaise utilisée dans la vie de tous les jours et dans les commandes; et la version française utilisée dans la documentation).

Mon expérience, c'est que si cette documentation a beaucoup de qualités – en particulier sa complétude est très appréciable quand on connait déjà Git et que l'on cherche un comportement particulier – mais est assez mal adaptée à une formation initiale à l'outil.

5. La CLI ne permet pas de « comprendre Git »

Ici je réponds à un argument que j'ai souvent lu, et qui est:

> Même si on ne l'utilise pas, il faut au moins apprendre à utiliser la CLI de Git, parce que c'est indispensable pour comprendre ce que fait Git et que les interfaces graphiques font de la magie en lançant toutes seules des commandes compliquées.

Pour moi, ce type d'argument est faux, et pour plusieurs raisons indépendantes.

5.1. Bien utiliser un outil c'est d'abord en comprendre les règles fonctionnelles

Git est un outil puissant, mais compliqué, parce qu'il permet énormément de choses.

Ce que j'ai constaté au travers des formations et des moult questions que j'ai vu passer sur diverses plateformes, c'est que généralement:

> **C'est la compréhension fonctionnelle qui pose problème**: indépendamment de l'outil, l'utilisateur ne sait pas *quelles opérations fonctionnelles effectuer* pour résoudre son problème.

Ce n'est pas l'utilisation de la CLI, pas plus que d'une quelconque GUI, qui va permettre à l'utilisateur de mieux appréhender Git. Ce dont il a besoin généralement, c'est de savoir ce que peut faire Git pour l'aider, et comment.

Par exemple, s'il a une branche de développement pas à jour, que va-t-il devoir faire? Un *rebase*, un *merge*, une nouvelle branche et *cherry-pick* certains de ses commits, etc.? Est-ce que l'utilisateur a la connaissance de toutes ces possibilités et de leurs cas d'utilisation? Généralement, non.

5.2. Le mythe du problème des commandes magiques faites par les GUI

Un autre argument qui revient souvent, c'est que les interfaces graphiques utiliseraient des commandes compliquées «sous le capot», que l'on ne maitriserait donc pas, ce qui:

1. serait dangereux,

2. empêcherait de comprendre ce qui est vraiment fait.

Ce à quoi je réponds:

1. On utilise un outil *précisément* pour faire de façon simple ce qui est compliqué sans lui.
2. Si on trouve qu'un outil est dangereux, soit c'est vraiment le cas (et il faut le prouver et changer d'outil ou le corriger), soit c'est de la méfiance à priori et n'a pas de sens.
3. Les GUI ne sont pas magiques: ce qu'elles font est, d'une façon ou d'une autre, permit pas Git.
4. Vous croyez vraiment qu'un utilisateur débutant comprend mieux ce que fait une commande de 50 caractères avec 4 options que ce que fait un appui sur un bouton?

5.3. « Utiliser la CLI » ≠ « Utiliser des commandes de bas niveau ou natives »

5.3.1. Nous ne sommes plus à l'époque de Git < 1.5

Historiquement et surtout avant la version 1.5, les commandes Git étaient très centrées sur le système de fichier et imposaient en effet de comprendre assez bien ce que Git réalisait pour pouvoir être utilisées. En fait, Git était plus proche d'une boite à outils que d'un système de gestion de contenu à proprement parler.

Ça n'est plus le cas depuis, parce que depuis Git a intégré un nouveau jeu de commandes.

Peut-être que *vous* avez dû apprendre des commandes absconses, ça n'est plus *nécessaire* aujourd'hui.

5.3.2. Plomberie et porcelaine

Les commandes proches du système de fichier existent toujours et sont pratiques pour une intégration dans des programmes ou dans des scripts. On les appelle les commandes de plomberie («plumbing» en anglais) et ne devraient pas être tapées à la main dans un terminal (dixit la doc officielle sur le sujet).

Mais il existe aussi tout un tas de commandes dont l'ergonomie a été pensée pour des utilisateurs humains. On les appelle les commandes de porcelaine et vous savez quoi? Elles sont aussi éloignées du fonctionnement interne de Git que peut l'être un appui sur un bouton dans une interface graphique.

Et c'est **normal**, puisque dans les deux cas ce sont des commandes destinées à être utilisées par des humains, et donc sont censé êtres simples, et non refléter au plus près ce qui est fait au niveau du système de fichiers.

5.4. Mais...

5.4.1. ... l'utilisation au clavier...

Toute bonne interface graphique peut être entièrement pilotée au clavier, et ne vous obligera pas à «perdre du temps» à le lâcher pour utiliser la souris.

5.4.2. … ça va plus vite…

Taper raccourci clavier n'est pas plus lent que de taper une commande de plusieurs caractères. Même lâcher son clavier et utiliser la souris est sans doute plus rapide.

Apprendre les raccourcis clavier du logiciel utilisé n'est pas plus long ni plus compliqué que d'apprendre les commandes Git, ni que de paramétrer des alias personnalisés d'ailleurs.

5.4.3. … et c'est plus accessible.

L'accessibilité est probablement la meilleure raison de préférer la CLI à une interface graphique.

6. Quel client graphique ?

À cette étape le lecteur avisé demandera:

? D'accord, mais quel client graphique Git me recommandes-tu?

Eh bien… c'est une bonne question.

Le problème est que de toute évidence, créer un bon client graphique Git, avec une bonne ergonomie, semble être quelque chose de complexe. Beaucoup de clients Git libres sont moches et/ou d'une ergonomie douteuse, et beaucoup de bons clients Git sont propriétaires. Cela dit, il semble y avoir plus de clients Git libres maintenant qu'à l'époque où j'ai fait le tour des principaux clients existants.

Ajoutez à ça que ça fait longtemps que je n'utilise plus que celui intégré à mon IDE, et donc que je ne sais pas ce que vaut le marché actuel.

Si le sujet vous intéresse, Git maintient une liste sur son propre site .

Néanmoins, voici quelques avis personnels de clients que j'ai testés personnellement, n'hésitez pas à compléter avec vos retours en commentaires!

6.1. Clients libres

— **Le client Git intégré aux outils JetBrains** : excellent; si vous utilisez un IDE de cet éditeur, ne cherchez pas plus loin, en plus son intégration au reste de l'IDE est nickel. Il a existé une demande à ce que JetBrains en fasse un outil indépendant, mais a été repoussée *sine die* par JetBrains . Reste à voir si un tiers la réalisera.
— **git gui** : le client officiel, mais minimaliste et pas toujours ergonomique.
— **TortoiseGit** : intégré à l'explorateur Windows (et donc seulement pour Windows), fait le boulot, mais a tendance à copier l'ergonomie de SVN (et de TortoiseSVN). Pas très pratique pour expliquer que Git n'est pas SVN du coup…
— **eGit, le client d'Eclipse** : il y a 3–4 ans, il était notoirement lent et d'une ergonomie très douteuse, trop proche de SVN. Je sais qu'ils ont fait de grosses avancées depuis, mais je ne sais pas à quel point, à tester si vous utilisez Eclipse.

— … plein d'autres que je n'ai jamais testés.

6.2. Clients propriétaires

— **SmartGit** : si vous n'avez rien contre les interfaces un peu spartiates, c'est pour moi le meilleur client indépendant, en particulier grâce à un excellent outil de diff/merge. Gratuit pour une utilisation non commerciale.
— **Sourcetree** : a été un bon client gratuit pour Windows et Mac, mais souffre de choix de développement hasardeux, comme la non-intégration d'outil de diff/merge, ou la suppression sauvage de certaines options (le *force-push* n'a été possible qu'en CLI pendant plusieurs mois à une époque, alors qu'il avait fonctionné avant…)
— **GitKraken** : la bêta était prometteuse, mais le modèle économique (version gratuite qui ne permet l'accès qu'aux dépôts publics des plus grands fournisseurs) me semble pas très sain.
— … plein d'autres que je n'ai jamais testés.

7. Aider et demander de l'aide sur Git

7.1. J'ai besoin d'aide

Indiquez directement le client que vous utilisez, et sachez vous servir de sa documentation.

C'est normal d'avoir des difficultés à comprendre les concepts Git et à en connaitre toutes les possibilités.

Mais c'est à vous de savoir comment transcrire une indication fonctionnelle précise en actions dans votre client particulier.

7.2. Je réponds à une demande d'aide

Ne partez pas du principe que «Git = CLI». En particulier:

Ne donnez **jamais** de lignes de commande à taper, encore moins sans les explications associées.
Une ligne de commande n'est **jamais** une explication en soi. Même si elle fonctionne.

Le but est que la personne en face sache ce qu'elle doit faire *fonctionnellement* pour résoudre son problème, de façon à ce qu'elle ne reparte ni avec une ligne de commande magique, ni avec l'impression que Git c'est encore l'un de ces *«putain d'outils de libriste de merde qui ne s'utilisent qu'en ligne de commande et ne sont utilisables que par des barbus asociaux»* (critique hélas trop fréquente, même si pas forcément verbalisée en ces termes).

Conclusion

La CLI est une interface de pilotage de Git qui peut vous convenir, et qui a ses avantages – en particulier celui d'être *toujours* disponible si Git est installé.

Ça n'en fait pas une interface *universellement pratique*, en particulier elle est plutôt génératrice d'erreurs et de temps perdu. Elle est aussi assez peu pratique pour l'apprentissage de Git, spécialement des concepts avancés de manipulation de l'arbre des *commits*.

Au contraire, les interfaces graphiques, en présentant sensiblement plus d'informations simultanées, permettent d'éviter des erreurs et de mieux appréhender des concepts complexes. À ce propos, il n'y a pas besoin que l'interface soit réellement *graphique*, on pourrait imaginer qu'une interface en `ncurses` fasse très bien le job.

Donc, utilisez la CLI si c'est le meilleur outil pour vous. Mais ne partez jamais du principe que c'est la meilleure interface *pour les autres*, et surtout, **arrêtez de répondre aux demandes d'aide en donnant des commandes Git CLI sans explications**. Parce qu'une ligne de commande n'est *en aucun cas* une explication en soi.

Le logo Git est CC BY 3.0 Jason Long .