Tuto

pour une utilisation

Propre

de Git

(Parce qu’on est pas des sagouins)

(Et que j’en ai marre de reprendre des bizuths qui font 3 pull par an)

Sommaire :

Ce tuto va expliquer tout ce qu'il y a à savoir sur git pour travailler proprement au club robotique.

Si vous découvrez l'outil, on part de zéro, donc faîtes-vous un bon thé, ça va être un peu long.

Bonne lecture !

1. Kézako ?

A) En général

Git est un logiciel permettant de partager du code facilement entre plusieurs utilisateurs. Il est largement utilisé dans tous les domaines de l'informatique. De nombreux projets, librairies, et autres sont partagés publiquement ppar leurs auteurs à travers cette plateforme.

Si vous ne voulez pas passer pour un con, ça se prononce /git/, donc [guitte] et pas [jite].

Les avantages de ce logiciel sont la tracabilité des modifications, le système de versions, et surtout sa capacité à accueillir plusieurs modifications à la fois sur des machines différentes.

Encore faut-il savoir l'utiliser correctement.

B) Et nous alors ?

Au Supaero Robotik Club, on utilise (comme probablement partout ailleurs) git pour élaborer le code des robots. Tout cela se passe sur «dossier» commun, que l'on appelle dépôt, ou que l'on surnomme ici vulgairement «le git». Entre autres, on pourra retrouver aisément le code des années précédentes (jusqu'à l'année 2019 et même 2018 avec les dossiers R1 dans celui de 2019).

Vous aurez ainsi le loisir de découvrir ce système génial et de vous énerver contre vos collègues qui refusent d'utiliser leurs outils correctement. Dans ce cas, envoyez-les relire ce magnifique tuto.

Si vous êtes le collègue susmentionné, bon, vous l'avez sûrement pas fait exprès mais sincèrement ça énerve. Donc on respire, et on se concentre pour utiliser correctement git. Promis c'est pas si difficile.

2. Comment on l'utilise ?

A) Installation

Pour l'installation de git, cela se passe par GitLab, un site qui permet de gérer les dépôts. Tout cela est expliqué dans un magnifique tuto que vous trouverez parmi les tutos sur les logiciels.

B) Première utilisation

Bravo, vous avez créé un compte GitLab, et copié le dépôt sur votre machine. À partir de là, vous pouvez utiliser git, c'est théoriquement suffisant.

Sauf que bon, on va pas se mentir, tout faire dans le terminal avec des commandes de 3km qu'on doit sans cesse rechercher sur internet, c'est chiant. Au SRC, on a choisi une solution : SublimeMerge. C'est probablement mentionné dans le tuto logiciel d'ailleurs.

Donc, pour commencer, on va ouvrir le git du club dans SublimeMerge (que je vais appeler SMerge à partir de maintenant) en faisant
file → open repository → <dossier Robotik> → ouvrir

Panique pas en voyant tout ça, t'inquiète pas ça va bien se passer.

C) Conseils

D'une manière générale, faut éviter de faire des trucs au pif en espérant que ça va marcher.

On a pas choisi cette interface là au hasard, c'est parce qu'elle est complète et pratique. Et gratuite. Sauf pour le thème sombre, qui lui est payant. On peut pas tout avoir :(

Honnêtement, on a longtemps utilisé ce qui est présenté dans «la base» dans le club. Sauf que c'est pas hyper propre, et ça peut régulièrement mener à des conflits. Donc on veut utiliser des branches, ce qui rend le truc beaucoup plus pro, sans effort. Et ça rend tout plus simple une fois qu'on a compris.

Si vous avez un doute, demandez à quelqu'un qui saura vous renseigner. Au pire, vous aurez toujours Google et StackOverflow.

3. L'interface SublimeMerge

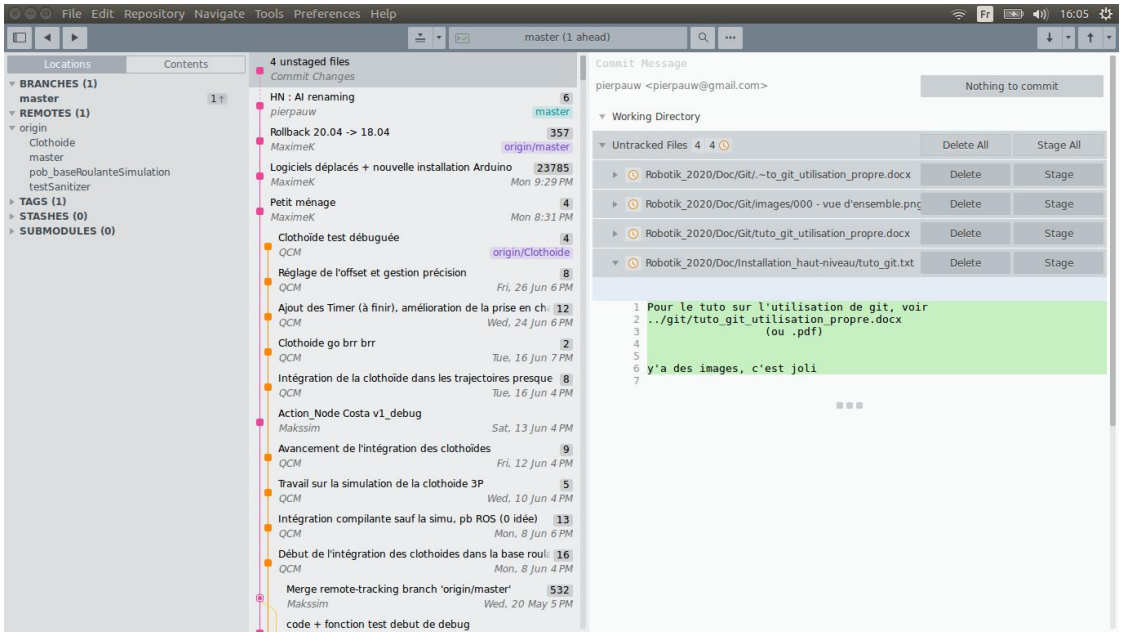
A) Introduction

Je vais tout présenter à travers l'interface de SMerge. Je sais pas sur combien de temps va perdurer ce tuto, mais actuellement on est en 2020, donc peut-être que vous aurez quelques différences. Peu importe.

Je dis peu importe pas parce que je suis un connard qui pense pas à vous, mais parce que c'est aligné avec du vocabulaire de git, qui est reconnu et utilisé depuis des années, et qu'en changeant ça y'a plein de trucs automatisé qui se casseront la gueule alors ça risque pas de changer[1].

Bref, faites pas chier si le bouton est plus au même endroit. Mieux, changez l'image pour que les suivant râlent pas.

Normalement, vous devriez arriver sur un truc qui a cette gueule là :



IMG #00 : Un aperçu classique

En fait, ça ressemblera plutôt à ça quand vous aurez commencé à bosser, et que vous serez en train de faire des modifications. Enfin voilà, ça me permet d'expliquer le principe.

Pour les parties suivantes (n>3), je montrerai sur un dépôt simplifié par souci de simplicité (et pour pas faire n'importe quoi sur le dépôt Robotik).

B) L'arbre

[1] En fait je dis ça mais y'a l'appellation master/slave qui est en train d'être modifiée alors que ça fait quand même partie des trucs qu'on imaginait pas changer. Bref.

Référez-vous à l'image précédente (IMG #00).

C'est la partie au centre. Un arbre est composé d'au moins une mais très souvent plusieurs *branches*. Ces branches sont les lignes de couleur verticales (1 couleur = 1 branche). Chacune a un nom, affiché en haut de la branche dans un rectangle de couleur. On reviendra sur le principe de branche dans la partie dédiée.

Chaque point carré le long d'une branche est un *commit*. Ignorez la première ligne «*4 unstaged files*» pour l'instant, on y revient en D). C'est une mise à jour du code. Le commit fait apparaître l'auteur de la modification, la date et l'heure, le nombre de fichiers modifiés, et un titre censé être expliquatif quant à la modification. Ainsi on voit l'historique des modifications lors du développement.

C) La comparaison origin/local

Référez-vous encore à l'image précédente (IMG #00).

Dans la partie de gauche, vous pouvez voir des sortes de dossiers *BRANCHES* et *REMOTES*. Ce sont les branches respectivement locales et globales, ou origine, ou serveur, ou partagées, enfin en ligne quoi t'as compris.

Là, dans *BRANCHES*, il y a une petite flèche vers le haut avec un 1 à côté de *master* «1↑». Ça veut dire que la branche locale *master* est «en avance», comme indiqué en haut : *master (1 ahead)*. Ceci, par rapport à la branche globale *origin/master* qui apparaît dans *REMOTES*.

Dans l'arbre, on observe cette différence : la branche *master* est plus à jour que *origin/master*, il suffit de remarquer les étiquettes en bas à droite des deux derniers commits.

D) Les modifications à approuver

Référez-vous encore à l'image précédente (IMG #00).

Vous êtes sûrement en train de vous demander ce que c'est que ce bordel qui prend la moitié droite de l'écran. Ce sont les modifications à approuver.

Dans l'arbre, la première ligne indique «*4 unstaged files*», ce qui signifie que 4 fichiers ont été modifiés. Ces modifications doivent être approuvées (*staged*) et consignées dans un commit.

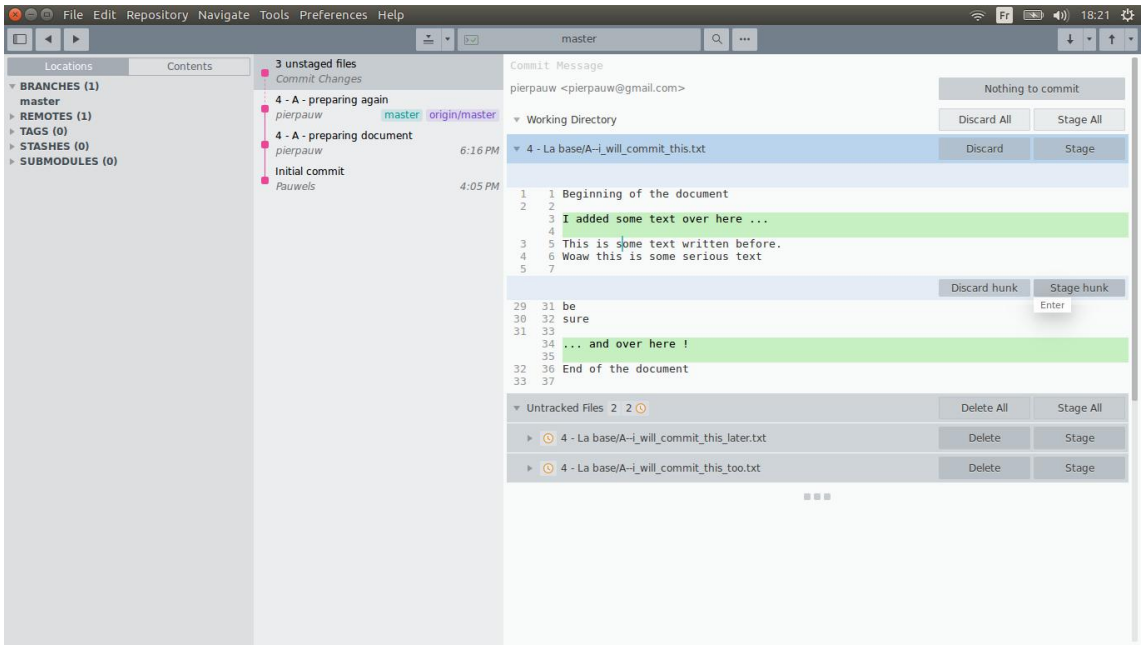
Sur la droite de l'écran, vous avez les susdits 4 fichiers modifiés, et en ayant cliqué sur le 4ème vous voyez en vert le texte ajouté dans ce fichier. C'est grâce à cette interface que l'on peut *commit*, donc réaliser un commit.

4. La base

A) Commit

Le *commit* permet de sauvergarder des modifications apportées sur des fichiers. Cela permet de garder facilement une trace de l'évolution des fichiers, et de sauvegarder ce qui fonctionne avant de tout casser.

Vous voici dans la situation suivante : vous avez effectué des modifications sur deux sujets différents. Observons la partie droite de l'écran sur l'image *IMG #01*.



IMG #01 - Avant de commit

Le fichier en bleu est un fichier modifié, les fichiers gris sont des nouveaux fichiers. Les fichiers supprimés, absents ici, sont rouges. En cliquant sur le fichier, on observe en vert les modifications faites.

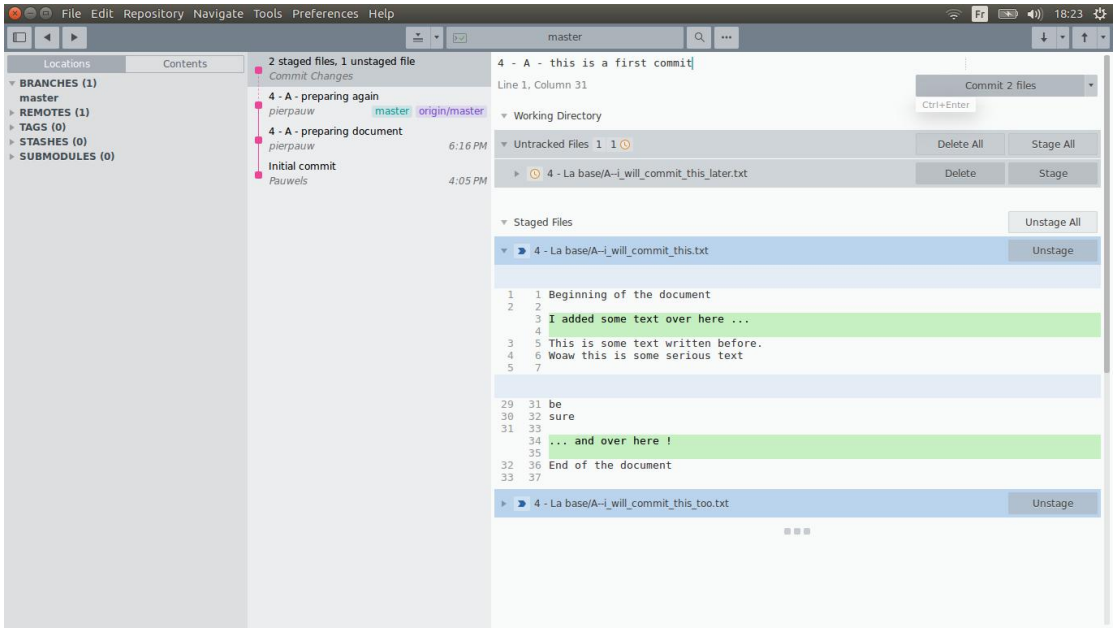
Lorsque vous voulez sauvegarder des modifications, il faut les commit. Pour cela, il suffit de valider les modifications désirées en cliquant sur *stage*. Notez que l'on peut stage uniquement une partie d'un fichier, avec *stage hunk* !

Après commit, ces modifications seront alors sauvegardées sur VOTRE machine. Vous pourrez également vous en servir pour revenir à une version antérieure, ou garder trace de l'évolution, mais elles ne sont pas accessibles depuis une autre machine tant que vous n'avez pas *push*.

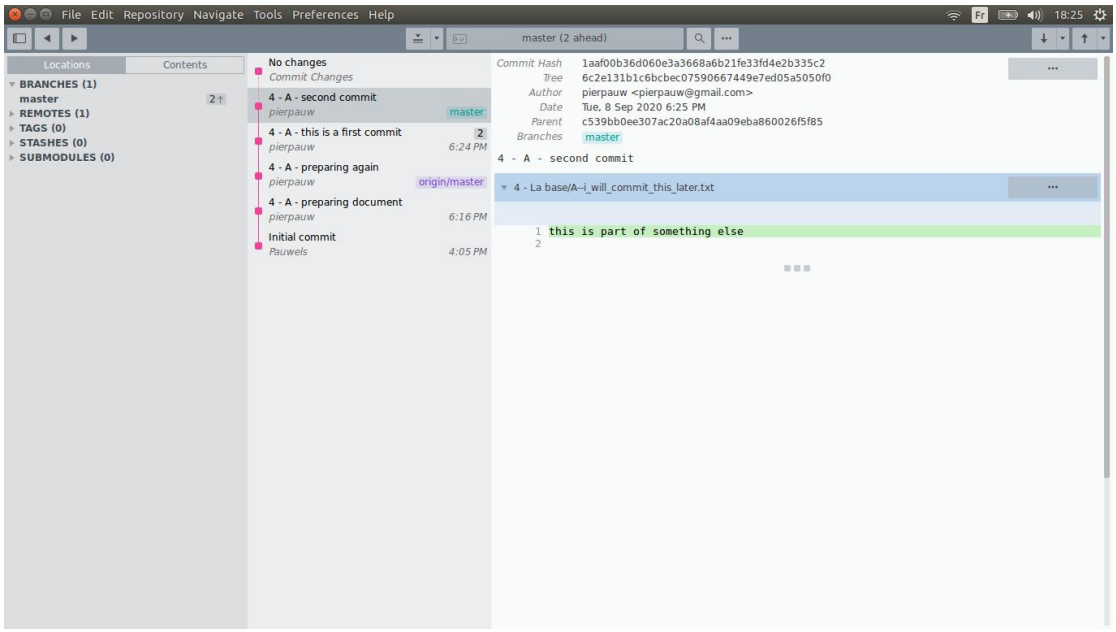
Il est souvent plus approprié de séprarer les commits selon les modifications. Des modifications sur deux sujets différents ne doivent pas se retrouver dans un même commit !

N'oubliez pas de donner un nom cohérent (et de préciser quel domaine ou partie du code est concerné(e)) dans le titre du commit. Merci.

Dans les images suivantes, on fait 2 commits distincts.



IMG #02 - Juste avant le premier commit



IMG #03 - Juste après le second commit

On se retrouve donc bien avec 2 commits différents qui présentent les modifications opérées.

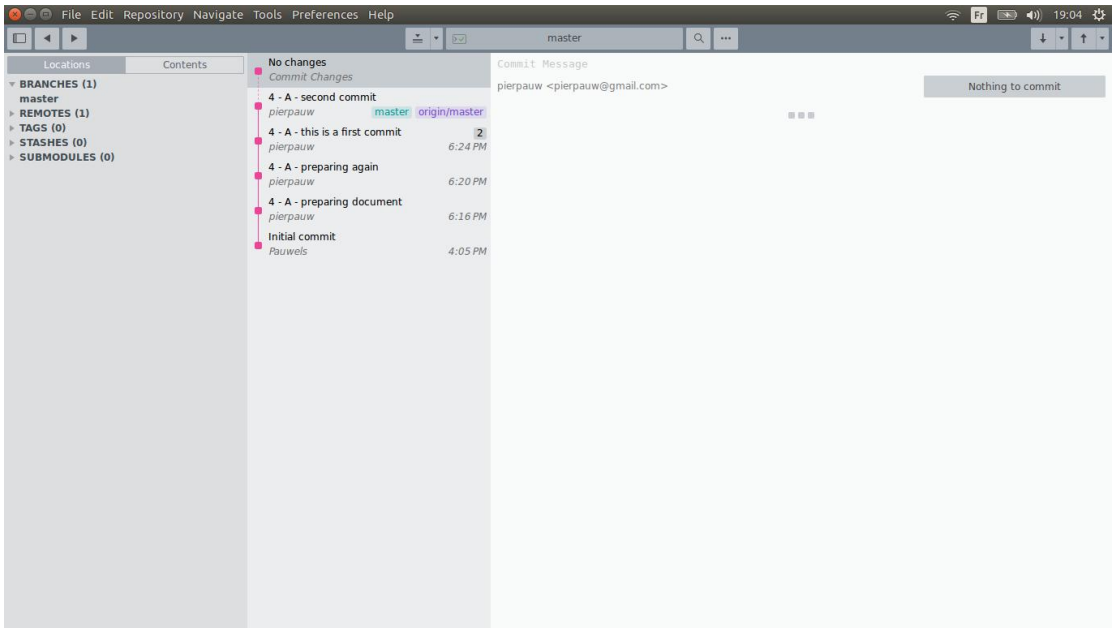
B) Push

Le *push* permet de «mettre en ligne», de partager les modifications que l'on a commit. Les modifications qui n'ont pas été commit ne sont pas partagées. Pour partager son code, il est donc nécessaire de push, sans quoi les modifications resteront sur votre ordinateur jusqu'à ce que vous les perdiez.

Pour cela, rien de plus simple (en théorie), il suffit d'appuyer sur le bouton push.

Regardez l'image *IMG #03*. À gauche, on voit à côté de *BRANCHES: master* le rectangle avec «2↑». Cela veut dire que la branche partagée *origin/master* a deux commits de moins que la branche locale (sur votre ordinateur).

En cliquant sur l'icône en haut à droite «↑», dite «push», vous partagez ces deux commits de différence et la branche *origin/master* est maintenant au même niveau que la branche *master*, ce qui apparaît encore une fois avec les étiquettes sur l'arbre.



IMG #04 - Après avoir push

Tout ceci se passe très bien, jusqu'à ce qu'il y ait conflit. On verra ça dans une autre partie, si vous vous débrouillez bien y'en aura jamais.

Avant de push, il faut cependant toujours *pull*. Je vous explique ça dans la partie juste après ↓↓.

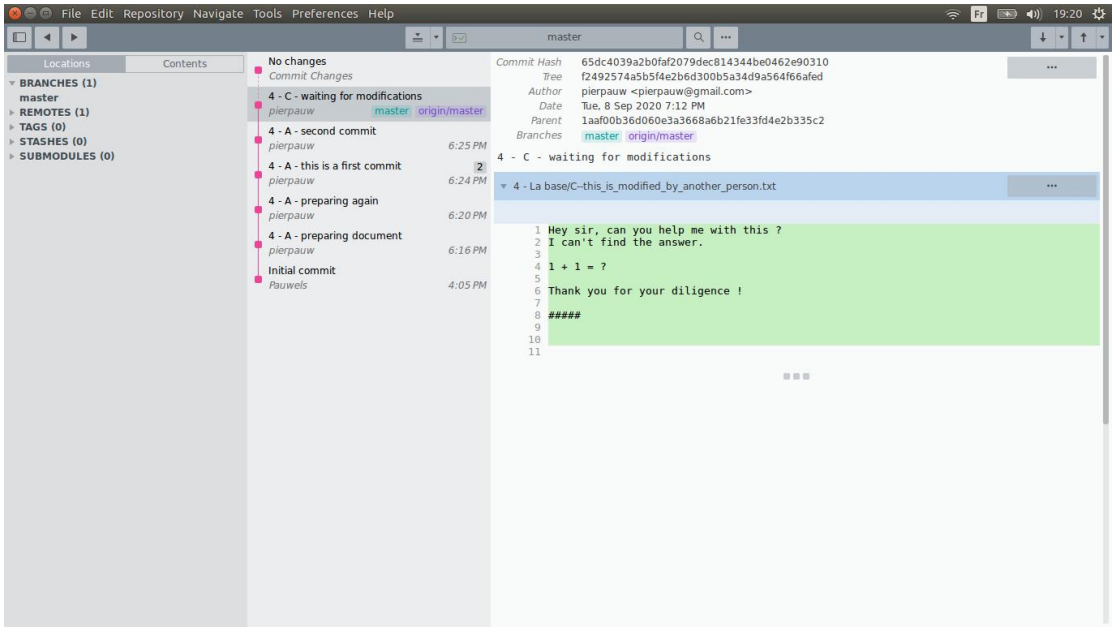
C) Pull

Faire un *pull*, ça n'a rien à voir avec le tricot. C'est mettre à jour le git sur sa machine. Ça permet de récupérer les modifications apportées par les autres.

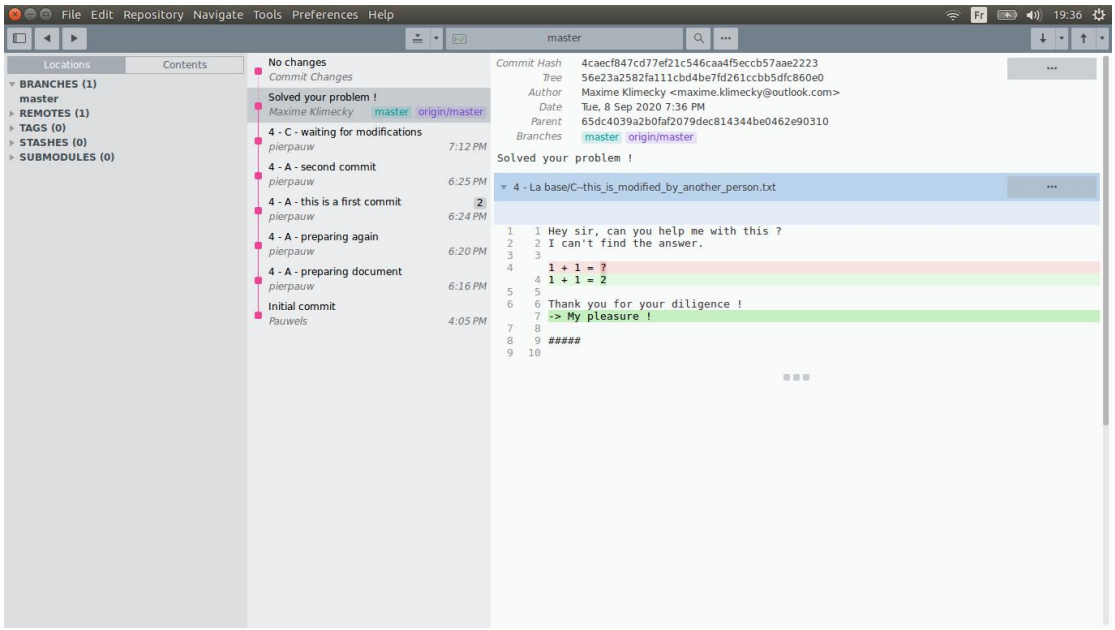
Il n'y a rien à faire si ce n'est appuyer sur le bouton «↓» en haut à droite, juste à côté de push.

Les modifications sont apportées automaiquement aux documents sur votre machine.

Voici l'état du git d'après ma machine, avant et après pull. Des modifications effectuées par d'autres utilisateurs sont apparues ! Elles sont également apparues dans mes propres fichiers par la même occasion.



IMG #05 - Avant le pull



IMG #06 - Après le pull

Il peut y avoir conflit lors d'un pull. Si c'est le cas, se référer à la partie sur les conflits.

Il faut **pull régulièrement** (aussi souvent que possible en fait) car cela permet d'avoir entres autre la dernière version de la branche principale, et de détecter des potentiels problèmes lors du développement.

Voilà, vous avez maintenant le nécessaire pour savoir utiliser git. C'était pas si terrible ?

Bon alors maintenant, pour rendre les choses plus propres, on va apprendre à se servir des *branches*.

5. Les branches

A) Qu'est-ce que c'est ?

Vous connaissez l'expression «Tranquille comme un singe seul sur sa branche»[2] ? Non ? La branche, c'est LE moyen de tout casser de son côté sans faire chier les autres.

Les branches, vous en connaissez déjà une (et même deux). De manière générale, c'est un axe de développement du code, et ce qui est marrant c'est qu'en disant «axe» on voit bien la «route» sur l'arbre dans smerge. L'exemple le plus parlant, c'est la branche *master* que vous avez utilisé jusqu'ici. La variante, c'est *origin/master*, qui n'est rien d'autre que *master* partagé (du coup ça compte quand même que pour un).

Mais on parlerait pas de branches s'il n'y en avait qu'une. Et c'est là tout l'intérêt d'en parler (et la vocation principale de ce tuto). On peut très bien se débrouiller avec une seule branche - et c'est ce qui a longtemps été fait - mais c'est bien mieux de faire plusieurs branches pour l'organisation.

Concrètement, vous allez chacun modifier des trucs un peu partout, et ça ne posera pas de problème. MAIS il y aura vraisemblablement un moment l'un de ces deux cas :
- 2 personnes travaillent sur le(s) même(s) fichier(s) ;
- vous avez une version qui fonctionne.

Et là, ça pourra causer un sacré bordel pour apporter des modifications de façon claire et sûre. Faire une branche secondaire, c'est s'assurer de développer tranquillement sans rien casser. Et à la fin, quand on sait que ça fonctionne et qu'on a testé, on peut l'intégrer dans la branche principale.

B) Créer une branche

Une branche part d'un certain état du code (donc d'un commit, si vous me suivez). Donc pour créer une branche, on va simplement partir d'un commit et créer une branche avec clic droit. (cf *IMG #07*)

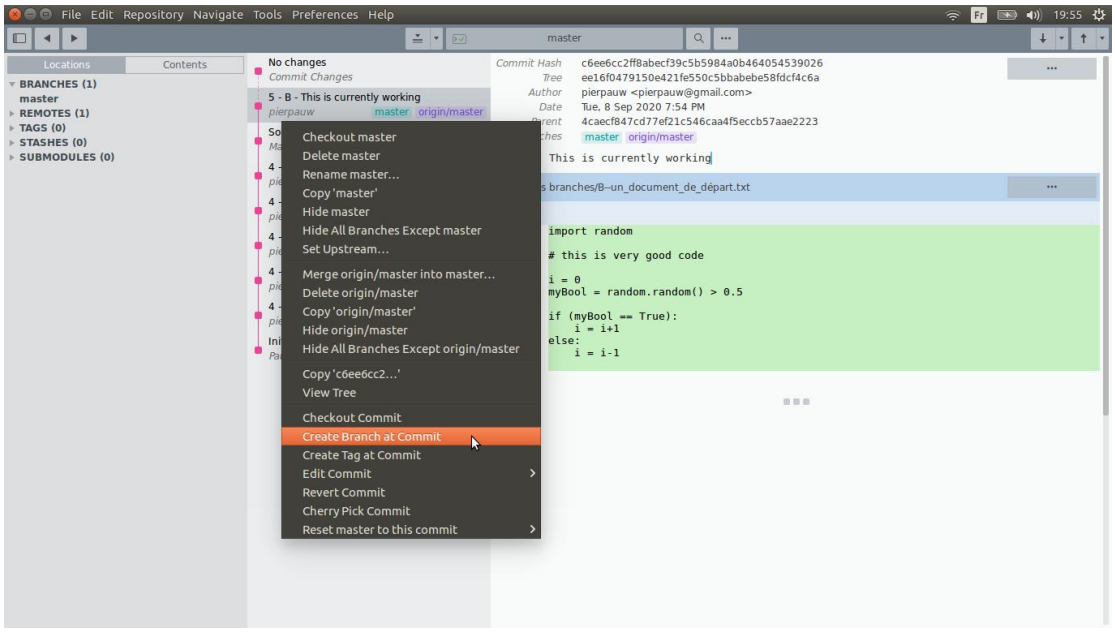
Après l'avoir nommée, vous la voyez apparaître au niveau du commit susmentionné. (cf *IMG #08*)

Ensuite, vous pouvez tout à fait faire vos modifications sur cette branche. Les modifications que vous apportez sur votre PC s'appliquent, lors d'un commit, à la branche en gras dans le panneau à gauche (par défaut la nouvelle). Faire des commits fait grandir cette branche, et vous permet de développer des nouvelles choses au fur et à mesure, en parallèle (cf *IMG #09*). Cela permet d'effectuer toutes les modifications que vous voulez sans jamais impacter la branche principale qui, elle, est censée toujours être en état de marche.

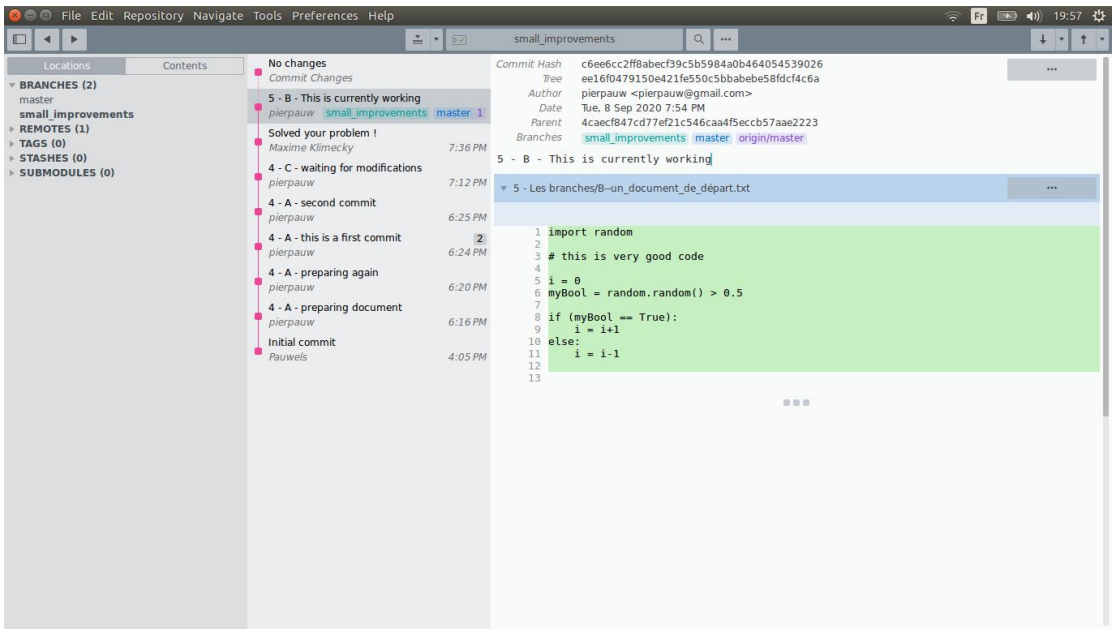
---

[2] Pierre Pauwels, 2020. (Je viens de l'inventer)

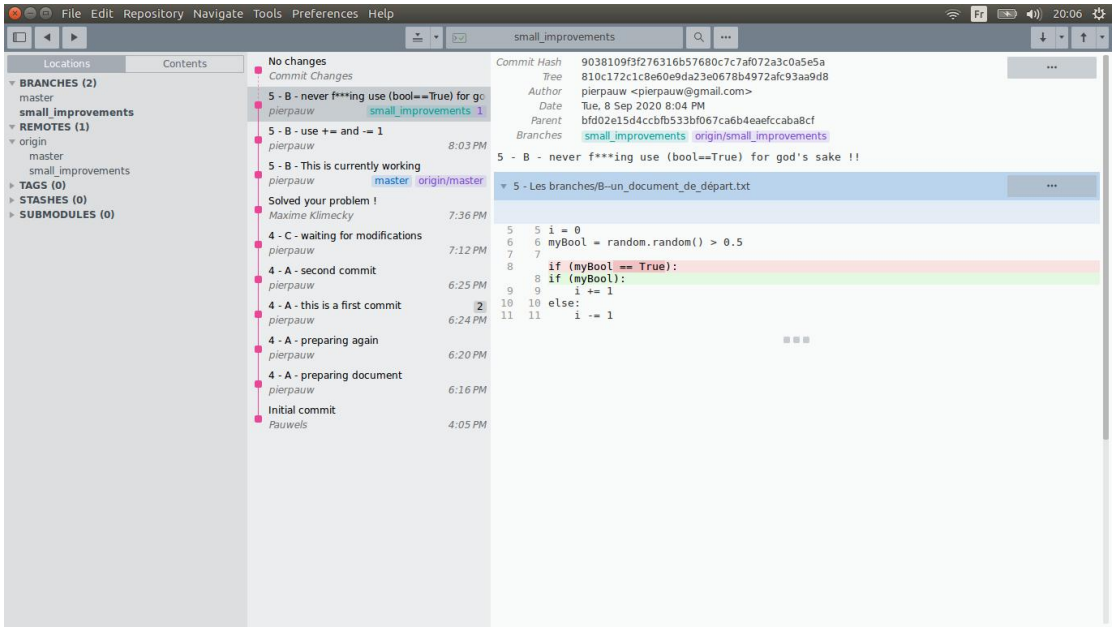
Bref, votre collègue qui se dit qu'il a envie de refaire tout son code peut le faire tranquillement alors que tout vos tests en ont besoin. Ça permet notamment de faire des mini-programmes provisoires (bidons) pour que les autres puissent avoir de quoi tester, et de construire de son côté un vrai truc qui fonctionne bien avant de le fournir aux autres.



IMG #07 - Création d'une branche



IMG #08 - Après la création de la branche



IMG #09 - Modifications apportées à «small_improvements»

C) Changer de branche

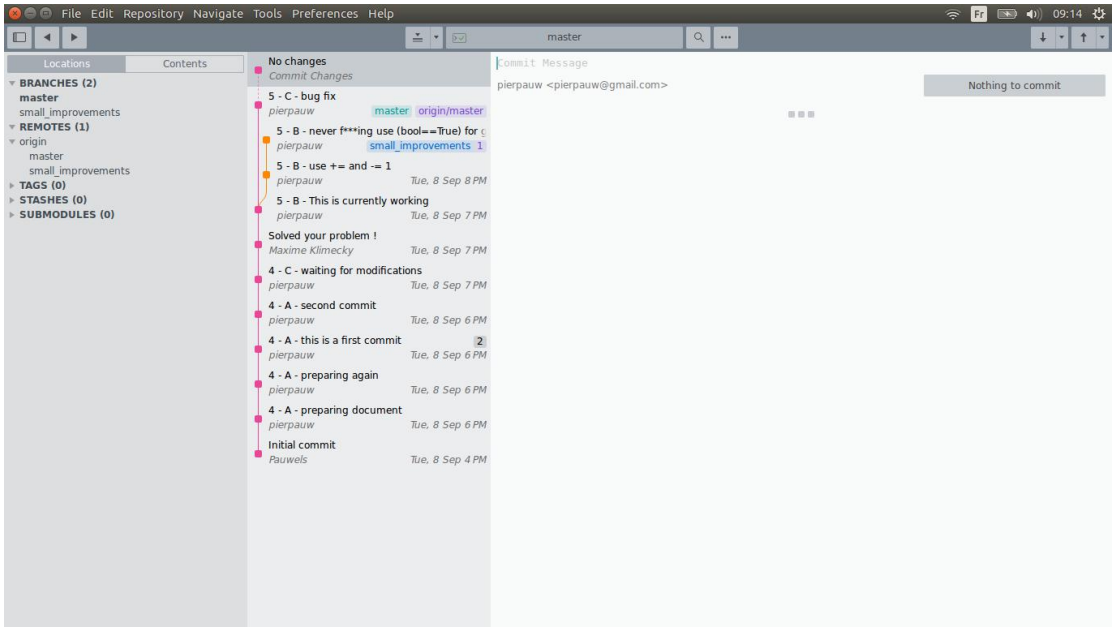
Pour changer de branche, rien de plus simple, il suffit de double-cliquer sur la branche voulue. Elle passe en gras, et la branche selectionnée est la branche rose sur l'arbre. Les autres branches sont d'autres couleurs.

Il suffit de changer de branche pour qu'automatiquement, tous vos fichiers du git[3] sur votre PC soient remplacés par ceux de ladite branche. À partir de là vous pouvez développer ce que vous voulez dessus sans impacter les autres branches (c'est le principe me direz-vous), et vous pourrez reprendre l'autre branche à tout moment.

Pourquoi c'est important ? Parce que si vous êtes en train de bosser sur quelque chose, et que quelqu'un vous demande d'ajouter un petit programme pour faire des tests, il faudra l'ajouter à *master* et pas attendre que vous ayez fini votre bordel. Un autre cas (plus courant) est celui d'un bug dans *master*, et donc reprendre la branche permet directement de récupérer le code en l'état, et de fixer ça au plus vite (cf *IMG #10*).

Bref, si quelqu'un change son code sur le git pour quelque chose qui ne fonctionne plus, vous pouvez à loisir l'engueuler en lui disant qu'on n'a pas que ça à faire que d'attendre qu'il corrige ses conneries, lui servir un bon thé et l'envoyer vers ce tuto.

[3] Plus exactement, tous les contenus des fichiers à l'exception des modifications qui n'ont pas été commit.

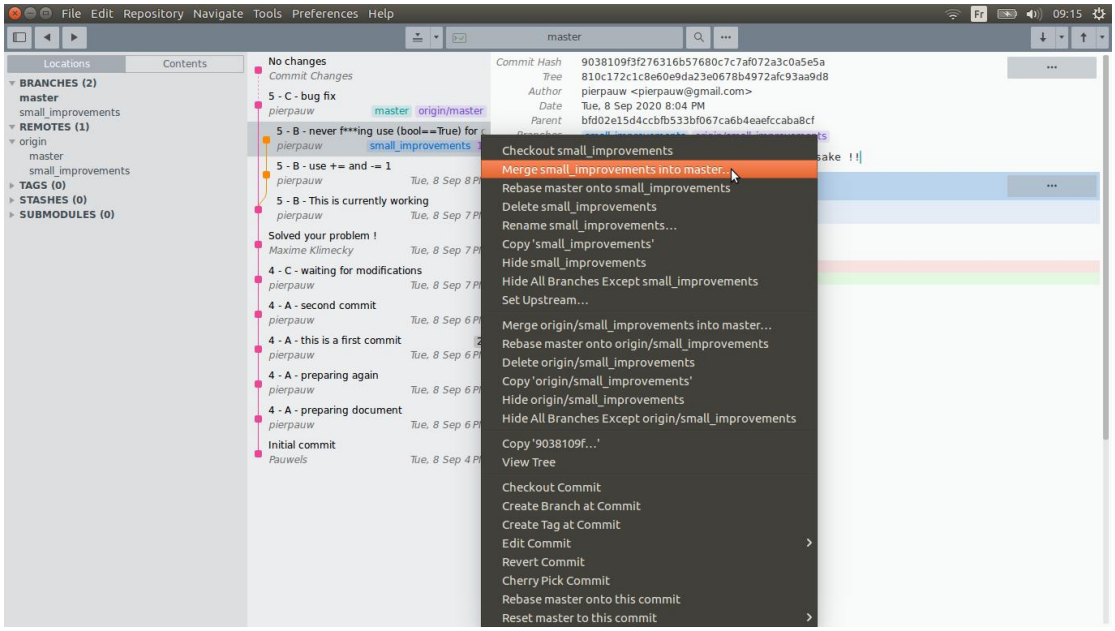


IMG #10 - Bug fix sur master

D) Fusionner des branches

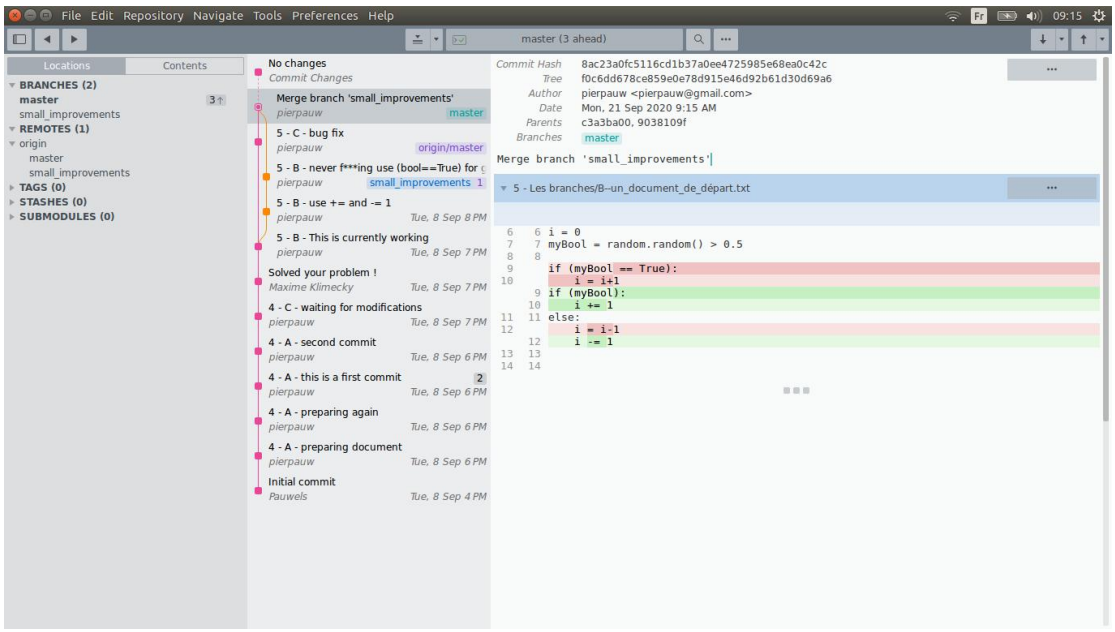
Alors créer des branches c'est bien, mais pouvoir appliquer les modifications c'est mieux. Pour ça, on fusionne les branches. On sit souvent aussi «merge» les branches.

La plupart du temps, il suffit de sélectionner la branche R «receveuse»[4] sur laquelle on veut importer les modifications (typiquement *master*). On fait un clic droit sur le commit de la branche D «donneuse» que l'on souhaite importer. Et on selectionne «merge D into R»), comme dans *IMG #11*. Et on obtient un nouveau commit dans la branche R qui applique les modifications apportées par D, comme dans *IMG #12*.



[4] Ces surnoms sont arbitraires.

IMG #011 - Avant fusion



IMG #12 - Après fusion

J'ai dit la plupart du temps. Parce qu'à force de modifications sur les deux branches, vous risquez de rencontrer un conflit. Si c'est le cas, vous pouvez vous rendre ici.

Ce dont il faut se rappeler, c'est que les modifications sont assez intelligentes pour se faire sans souci si elles ne sont pas liées. C'est même possible au sein du même fichier ! Mais les ennuis arrivent si des modifications proches ont été apportées des deux côtés ...

Notez également que l'on peut importer des modifications de la branche principale vers une branche secondaire, ce qui est notamment très utile (et évite des problèmes futurs) dans le cas d'une correction de bug.

6. Les conflits

A) Pourquoi ça marche pas ?

Là, on va attaquer un gros morceau. Il y a plusieurs raisons possibles, mais elles sont toutes plus ou moins liées.

En général, quand vous ne pouvez pas réaliser une action dans git, c'est que votre version et celle que vous essayez d'importer ne sont pas d'accord. Il s'agit juste de les mettre d'accord.

Je sais que c'est la base, mais surtout, LISEZ LE MESSAGE D'ERREUR. C'est pas compliqué et souvent c'est assez parlant, et c'est un peu la base, si ça se trouve la solution est dedans bordel.

Si votre problème n'est pas dans cette liste, demandez de l'aide ou allez vous renseigner sur internet, et vous pouvez ajouter ce problème et sa solution à cette liste.

i. Je ne peux pas push

En général, vous avez un message du type «cannot push before pull». Alors là, soit vous avez pas du tout lu ce qu'il y a d'écrit juste au-dessus parce que la solution est littéralement dans le titre, soit vous ne pouvez pas pull non plus. Et le problème vient en réalité de ceci, donc référez-vous à la prochaine section.

ii. Je ne peux pas pull

Il n'y a en général qu'une raison pour laquelle on ne peut pas pull (mis à part ne pas être connecté à internet) : vous avez apporté des modifications sur (au moins) un segment de code qui a été également modifié par quelqu'un d'autre sur cette branche. Bref, votre branche (locale) a évolué différemment de la branche globale.

J'espère que c'est pas parce que vous avez 237 commits de retard, mais si c'est le cas sachez que je vous engueule très fort. Il faut pull régulièrement, c'est pourtant pas compliqué !

Comment éviter cette erreur ? Il est très facile de l'éviter en faisant ses modifications sur une branche dédiée. Ainsi personne ne viendra modifier vos trucs en même temps que vous.

Dans un premier temps, si vous n'avez pas commit vos modifications, vous pouvez le faire, c'est jamais une mauvaise idée. Vous pouvez même le faire dans une nouvelle branche, ça peut vous permettre de pull et d'ensuite merge, ce qui résultera probablement dans le cas d'erreur suivant mais sera beaucoup plus lisible au niveau des modifications. Rendez-vous dans la partie suivante pour comprendre comment résoudre ce problème.

iii. Je ne peux pas merge

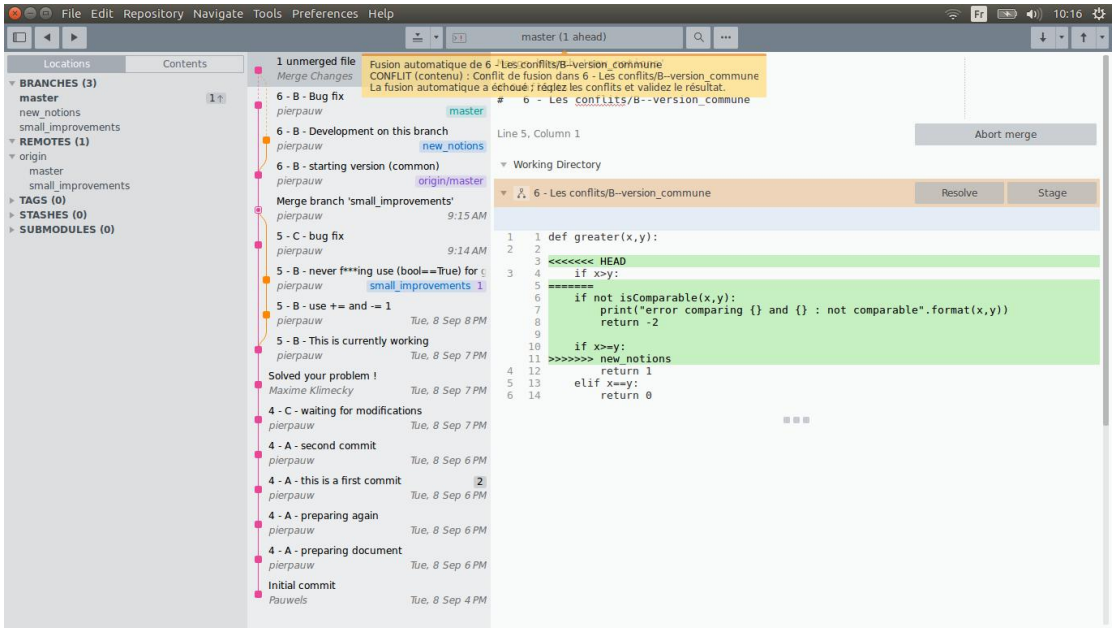
L'impossibilité de merge est relativement courante, puisqu'elle arrive dès qu'un segment a été modifié de façon différente sur les deux branches à la fois.

Déjà, il faut savoir que ça peut être dû à une mauvaise utilisation des branches, SAUF dans un cas où c'est tout à fait normal. Si vous avez corrigé un bug dans les deux branches, ou si vous avez corrigé dans une branche et remplacé par autre chose dans l'autre (cas typique d'un code de test dans master), ce problème apparaît.

Pas de panique, l'éditeur smerge est vraiment super pour corriger ce genre de problèmes. RDV partie suivante.

B) Comment on fait pour que ça marche ?

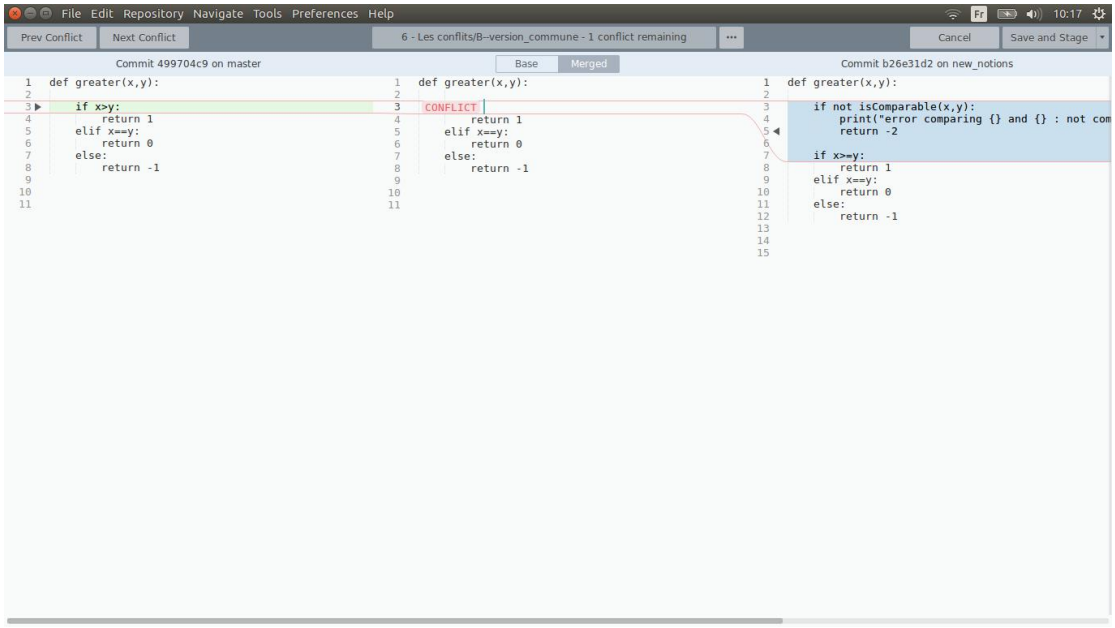
Normalement, avec ce qui précède, vous êtes rendus à un problème de merge. Donc on va apprendre à se servir du magnifique système de résolution de conflits de smerge.



IMG #13 - Situation de conflit

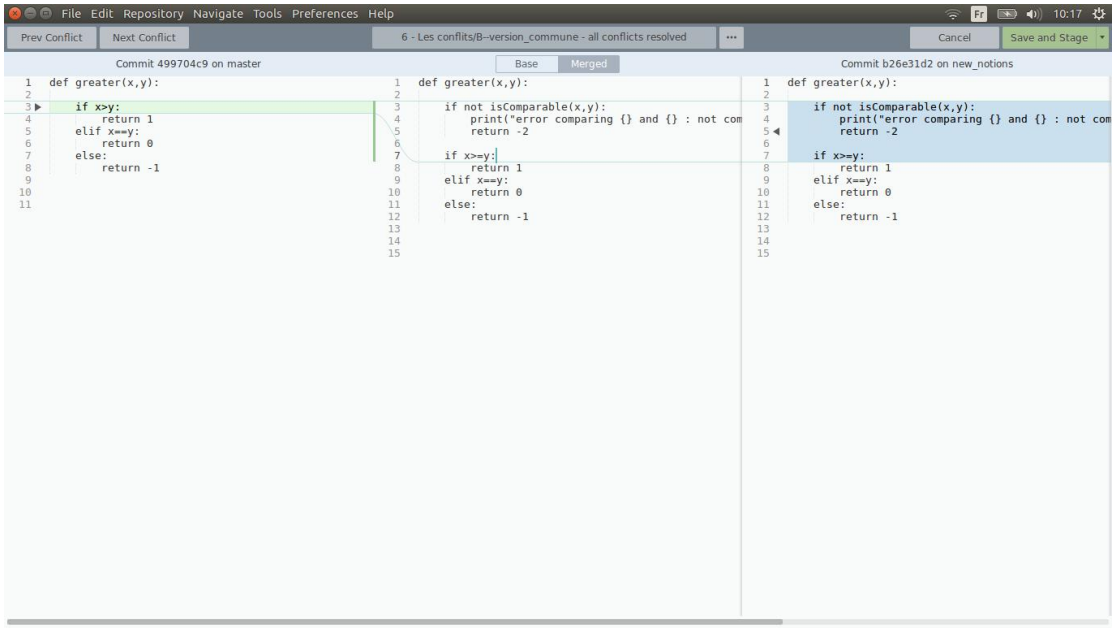
La situation devrait ressembler à celle de *IMG #13*. Cliquez sur *resolve* pour ouvrir le gestionnaire interactif de conflits. Vous arrivez sur cette interface *IMG #14*. Notez que s'il y a plusieurs conflits sur ce fichiers, chacun apparaîtra, et est visible en rouge dans la barre de défilement à droite si le fichier est long.

Vous y voyez le lieu du conflit (en rouge), avec les versions proposées dans chacune des deux branches (en vert et en bleu). Les branches corrsepondantes sont d'ailleurs indiquées dans les onglets au-dessus du code. Le code au milieu représente le code au sortir du conflit, c'est lui qu'on va modifier.



IMG #14 - Gestionnaire de conflits

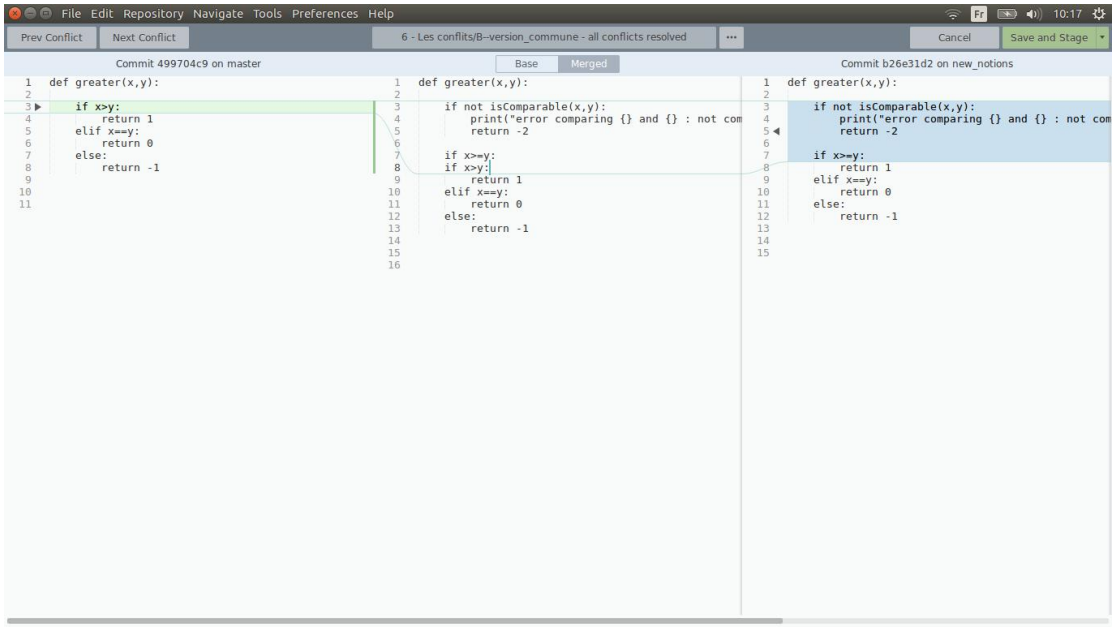
En cliquant sur la flèche ← du fichier de droite, on sélectionne cette solution pour résoudre le conflit. Il apparaît dans l'espace précédemment rouge comme en *IMG #15*.



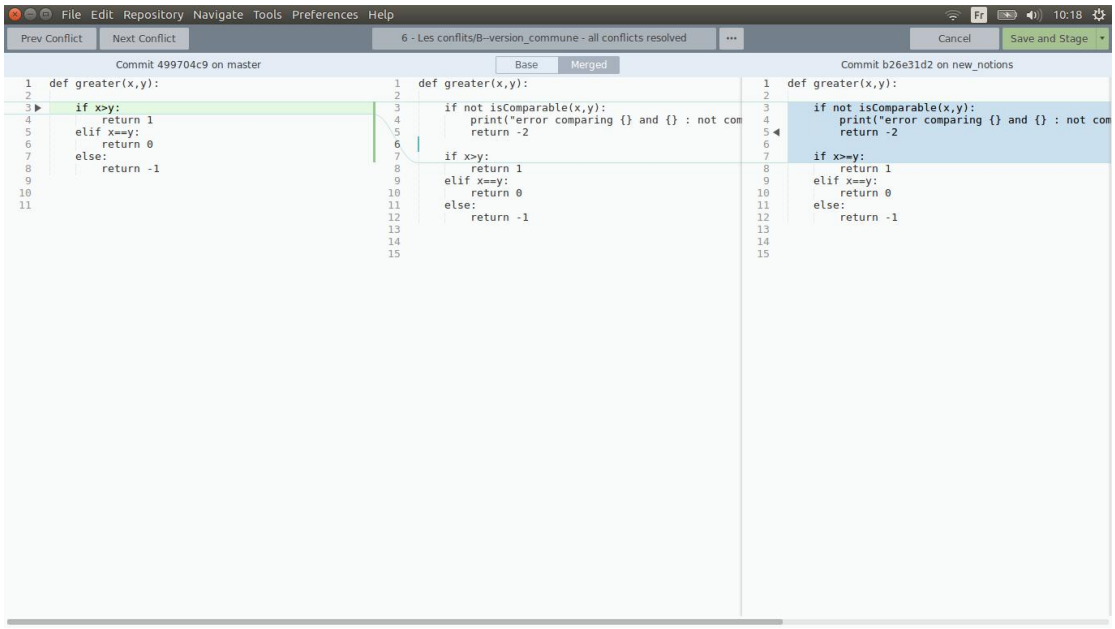
IMG #15 - Choix d'une solution

Mais ça ne s'arrête pas là, car on veut ici aussi un peu de l'autre solution pour remplacer le morceau bugué dans le code de droite. En cliquant sur la flèche → du fichier de gauche, la solution de gauche s'ajoute à la suite dans le code du milieu, comme dans *IMG #16*.

Vous vous rappelez que j'ai dit gestionnaire **interactif** de conflits ? Et bien oui, il est complètement interactif, et on va même modifier en direct et au clavier le code au milieu pour obtenir une solution propre (*IMG #17*).



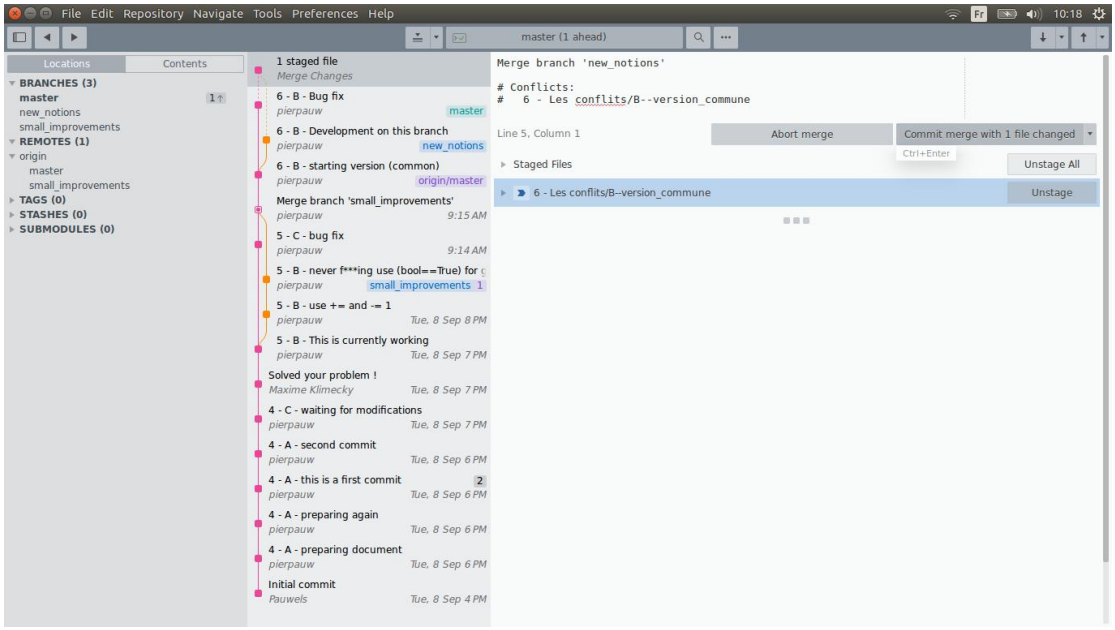
IMG #16 - Ajout de l'autre solution



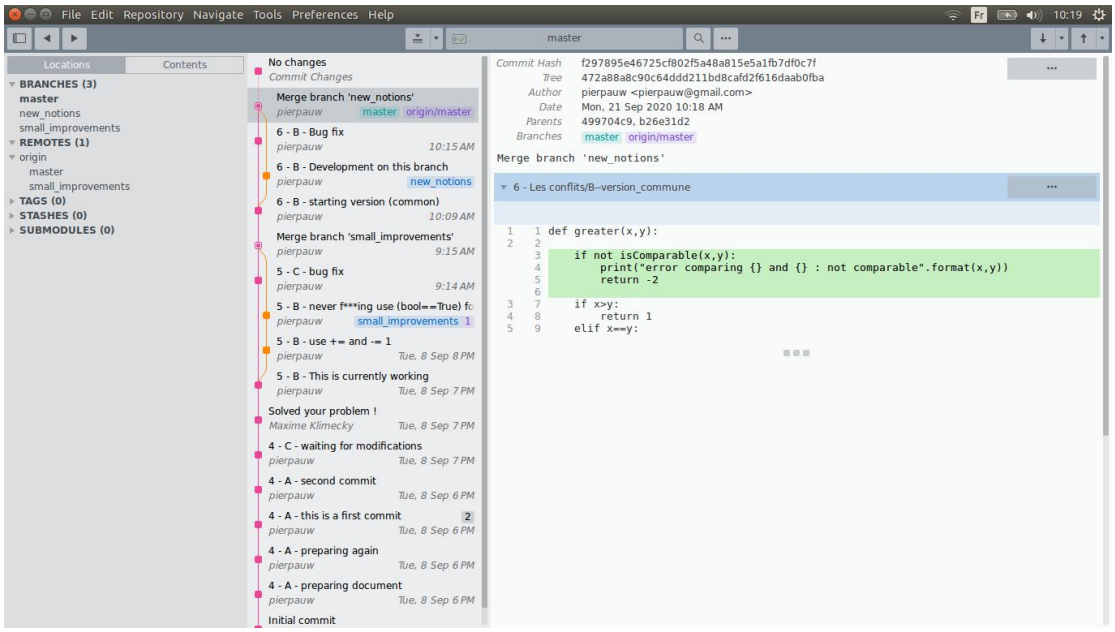
IMG #17 - Correction dans l'éditeur interactif

Maintenant que la correction est apportée (il faut le faire pour chaque conflit), on peut sauvegarder et enregistrer les modifications dans le commit, en cliquant sur *Save and Stage*. Comme sur *IMG #18*, on voit le fichier modifié prêt à être commit. Faites de même pour chaque fichier avec conflit (ou pas si vous voulez commit les modifications une par une mais c'est bizarre, vous voulez merge à l'origine non ?).

Et vous pouvez commit pour terminer la fusion des branches et vous retrouver, enfin, dans la situation de *IMG #19*. Génial non ?



IMG #18 - Le nouveau fichier est staged



IMG #19 - Merge terminé

C) Important

Le gestionnaire de conflits a beau être incroyable, vous n'avez pas réellement envie d'avoir à gérer des conflits. Mais bon, la première fois on panique, la deuxième on s'améliore, et très vite, on sait ce qu'on fait, on devient bon et c'est juste du plaisir[5].

N'hésitez pas à demander à vos collègues l'origine des modifications que vous n'avez pas faites, elles ont très probablement une raison d'être ici, en particulier si elles sont sur la branche principale.

[5] Tout parallèle lubrique serait purement fortuit.

7. Ego feci stercore

A) Avant-propos

Cette section est dédiée à la gestion des problèmes causés par l'utilisateur lui-même (oui toi), et à la réparation, la restauration à un état précédent qui permettra de corriger les erreurs précédentes pour parvenir à conserver une utilisation simple, cohérente et lisible de git.

Notamment, faire «ctrl+z» n'est pas si simple et dépend de la situation. Il faut notamment déterminer si les modifications à annuler sont pour l'instant uniquement sur votre machine, ou si elles ont déjà été push sur le git.

(spoiler : j'espère pour vous que vous n'avez pas push)

B) Erreur locale

Il n'y a pas grand-chose à corriger tant que c'est sur votre machine, donc ça va aller vite :

- J'ai fait un commit et je n'aurais pas dû / je n'ai pas inclus toutes les modifs / j'ai inclus trop de modifs / je souhaite changer le nom du commit :

Si vous avez push, il faut se référer à la partie Erreur sur le git, car la modification est déjà partagée. Sinon, pas de souci, vous allez dans l'onglet *Repository* et vous faites *Undo: commit*. C'est tout, vous pouvez le refaire proprement.

- J'ai apporté une modification à un endroit et je voudrais retrouver ce qu'il y avait avant (v.1) :

Si les modifications n'ont pas été commit, dans smerge, à coté de *stage* dans le volet des modifications, on trouve *discard*. Ceci permet d'annuler la modification et de retrouver le code du commit précédent sur ce fichier ou tronçon (*discard hunk*).

S'il y a eu commit (mais pas push), il est possible d'annuler le commit comme précédemment et de discard (et éventuellement re-commit).

Attention, discard une modification la fera disparaître complètement et à tout jamais, donc enregistrez-la ailleurs si vous voulez la conserver d'une quelconque manière.

- J'ai apporté une modification à un endroit et je voudrais retrouver ce qu'il y avait avant (v.2) :

Dans le cas de grosses modifications, ou si l'on veut complètement revenir à un état précédent, il existe en cliquant droit sur un commit, *reset <branch> to this commit*, ce qui permet de faire revenir les fichiers à l'état de ce commit.

Attention, les modifications non commit disparaîtront complètement, enregistrez-les ailleurs si vous souhaitez les conserver. Notez que le *soft reset* annule les modifications effectuées, alors que le *hard reset* «supprime et remplace» tous les fichiers par leurs prédécesseurs. Je crois que ça veut dire que les nouveaux fichiers qui trainent ici sont perdus lors d'un hard reset. Je crois. Souvent le soft ça suffit, mais si vous

avez un doute et rien à perdre dans le git, un hard reset ça marche bien aussi.

C) Erreur sur le git

On va pas se mentir, si vous avez push, c'est tout de suite plus compliqué. Ctrl+z et les techniques précédentes ne sont d'aucune utilité. On va devoir faire des commits qui annulent des commits précédents, bref c'est pas terrible, donc faites gaffe svp.

- Je veux renommer un commit :

Si c'est un vieux commit, typiquement il existe une branche qui est postérieure à ce commit, c'est pas vraiment la peine, ça fait des trucs moches parce que la branche partira encore de l'ancien commit et non du «nouveau» commit renommé (bref c'est le bordel). Regardez la fin si vraiment ça vous tient à coeur.

S'il est récent, vous pouvez faire clic droit, *edit commit*, *edit commit message*. Ça va faire un nouveau commit (et les commits suivants seront copiés). Sauf qu'à partir de là, la branche locale et la branche globale auront divergé, donc on a un conflit, on peut pas push et ça aurait servi à rien. Sauf qu'on peut forcer le push (option --force, accessible notamment par la flèche ↓ du menu déroulant de push). Il va sans dire qu'il faut savoir ce que l'on fait, et pas tenter des modifications à la volée en espérant que ça marche.

Une solution simple (mode système D) est de faire un nouveau commit (presque) vide, avec un nouveau message.

- Je veux éditer un commit :

Si c'est le plus récent, cf le cas **renommer** en remplaçant *edit commit message* par *edit commit content*. S'il y a eu des commit ultérieurs, je conseille pas. Regardez plutôt **annuler un commit** et recommencez.

- Je veux annuler un commit :

Il est possible à tout moment d'annuler les modifications effectuées lors d'un commit. D'où l'intérêt de faire des commits réguliers, distincts, avec des messages cohérents. Sur le commit à annuler, clic droit, *revert commit*, et dans un nouveau commit vous annulez ces modifications (et uniquement celles-ci, dans le code présent aujourd'hui et donc modifié depuis). N'hésitez pas à compléter (mais pas remplacer) le message du commit de revert.

- Je veux annuler les derniers commits :

Personnellement, je préfère annuler les commits un à un, mais pour des modifications simples et abandonnables voilà une technique. Attention, tous les commits annulés seront définitivement perdus.

Commencer par reset vos fichier au commit auquel vous voulez revenir (voir Erreur locale, modification v2 pour plus d'infos). Ensuite, faites un push --force, ce qui remettra la branche distante au niveau de votre branche locale. Je rappelle que tous les commits qui suivent celui auquel vous avez reset seront définitivement perdus, donc prenez garde et n'annulez pas des commits de quelqu'un d'autre.

Moralité : prenez toujours le temps de vérifier le contenu et le message des commits avant de commit déjà, mais une autre fois avant de push.

8. Remarques

A) Ce tuto

Ce tuto est assez sommaire et fait, il faut l'avouer, un peu à la volée. Dans sa première version, il repose sur ce que j'ai pu acquérir de mon expérience encore jeune, mais j'espère qu'il sera assez compréhensible pour acquérir facilement les bases d'une utilisation propre et saine de git.

Les résolutions de problèmes, comme beaucoup d'autres choses mais celles-ci en particulier, sont loin d'être exhaustives, mais elles devraient permettre de résoudre la plupart des situations dans une utilisation simple de git. Mais bon, si la réponse n'est pas là, elle est cependant trouvable sur internet (où croyez-vous que j'ai appris ça, moi ?).

Le point positif, c'est que maintenant que vous avez ce tuto entre les mains, vous pouvez vous aussi participer à le rendre meilleur ! S'il y a quelque chose qui vous semble imprécis voire incorrect, s'il y a des changements importants dans l'interface smerge (ou si vous changez d'interface), ou simplement si vous souhaitez compléter ce tuto, n'hésitez pas, ça profitera toujours à vous et aux suivants qui vous liront.

N'oubliez pas qu'une des meilleurs façons d'apprendre reste de pratiquer, donc vous aurez l'occasion de découvrir par vous-même ce que vous avez commencé à apercevoir ici (et dans une situation bien plus intéressante que mes exemples bidons).

B) Communiquez !

Une chose primordiale est que git est un outil **collaboratif**. Ça signifie que git est à son plein potentiel lorsqu'il est utilisé en groupe[6], avec d'autres gens du véritable monde réellement réel du vrai dehors. Et donc, vous aurez beau être extrèmement doués avec git, si les autres savent pas l'utiliser, ça rend les efforts de propreté nuls.

Ce tuto est probablement beaucoup trop long pour quelqu'un qui ne s'y attend pas ou qui utilise déjà vite fait git (quoique vous pouvez renvoyer vers les sections «bases» ou «branches» respectivement), mais une façon bien plus efficace est de leur montrer directement[7].

Et puis, quand ils feront des push ignobles de code foireux sur la branche principale, vous pourrez leur dire d'aller voir le tuto en disant que vous leur avez déjà expliqué et que maitenant ils se démerdent.

Que le dieu Jon Skinner t'accompagne dans ton code.

---

[6] Ce qui ne vous empêche absolument pas de l'utiliser pour vos projets perso, c'est d'ailleurs super pour des projets un peu longs. Et personnellement je m'en sers même pour pull automatiquement des maj à distance depuis mon Raspberry. Bref c'est génial, utilisez git.
[7] Bon du coup si vous lisez ça c'est que vous passez par le tuto, mais au moins c'est plutôt complet, vous êtes plus compétents, et en plus si vous oubliez le tuto est là.

*** FIN ***

Voilà, c’est la fin de ce tuto.

Bravo d’avoir lu jusqu’ici.

Tu gagnes un point tuto.