Introduction à Git

Dans ce tutoriel, vous allez apprendre à utiliser Git, un outil de gestion de version décentralisé adapté aux développements logiciels à plusieurs. Les entrepôts GIT sont très utilisés par toutes les équipes de développement. Il est important de maîtriser cette technologie car elle participe à la qualité des développements et à leurs sécurités car il est possible de tracer les versions et de revenir en arrière le cas échéant.
Les trois grands avantages de Git sont :

- Un historique complet du projet (versionning), de revenir à une version antérieure en cas de bug ou d'erreur ;
- Git facilite grandement la collaboration : Chaque développeur peut travailler sur sa propre copie du projet, en parallèle. Git détecte également les conflits si deux personnes modifient la même partie d'un fichier. Il permet aussi de fusionner les changements facilement, en s'assurant que personne ne perd son travail ;
- Git est devenu l'un des logiciels de version les plus populaires et le plus utilisé, il existe donc énormément d'outils et de librairies tournant autour de Git. La plupart des éditeurs de code tel que Netbeans reconnaissent également Git.

Git devient particulièrement puissant lorsqu'il est accompagné d'une plateforme en ligne qui héberge ses dépôts, tel que **GitHub** ou **GitLab**. Ces plateformes permettent de bénéficier de ce que Git offre grâce à l'hébergement sur serveur distant du projet, mais aussi de profiter d'outils spécifiques à la plateforme comme les pull requests ou les actions CI/CD. Dans ce tutoriel, nous verrons comment utilier Git avec la plateforme GitHub, et plus particulièrement leur outil **GiHhub Desktop**. A ce propos l'école est campus GitHub et peut héberger à ce titre vos projets sur nos propres serveurs si vous choisissez ce dépôt.

I Fonctionnement général de Git

Un projet versionné avec Git est appelé un **dépôt** (repository). Ce dépôt contient :

- Les fichiers du projet ;
- un dossier caché `.git/` où Git stocke tout l'historique et les métadonnées.

L'utilisateur travaille sur son **dépôt local**, lui seul a connaissance de ce dépôt. Il est possible de créer à tout moment un dépôt local grâce à la commande `git init`. Ce dépôt local peut être rattaché à un **dépôt distant**, typiquement grâce à une plateforme comme GitHub ou Gitlab. Lorsqu'il n'est pas associé à un serveur distant (comme avec GitHub par exemple), Git permet surtout de garder l'historique des changements de code.

1 Enregistrement dans l'historique : Les commit

Lorsque vous souhaitez sauvegarder vos modifications dans l'historique, vous effectuez un `commit`. Il s'agit en quelque sorte d'une « capture d'écran » de votre code à un instant de son développement. Si vous modifiez plus tard votre code, mais que ces modifications ne vous conviennent pas, vous pourrez alors revenir à une version précédente. Git utilise un système en trois étapes pour enregistrer les modifications :

- **Working Directory** (répertoire de travail) : Il correspond au répertoire de travail, où se trouve le projet ;
- **Staging Area** (zone de préparation) : Il est possible de suivre les modifications uniquement de certains fichiers choisis parmi les fichiers du Working Directory. Cette étape consiste donc à sélectionner les fichiers à inclure dans le prochain snapshot (`commit`). Vous pouvez ajouter vos fichiers directement sur un terminal avec la commande `git add <vos_fichiers>` ;
- **Repository (HEAD)** (historique validé) : Une fois les changements validés, ils sont stockés dans l'historique du projet. Pour valider votre commit, vous pouvez passer par la commande `git commit -m "Votre message de commit"`.

Le message de commit doit être clair afin que vos collègues sachent les modifications que vous apportez. Une bonne pratique consiste à mettre des « tag » pour plus de clarté, par exemple : "[ADD][SQL] Ajout de la classe X pour la base de donnée".

2 Lien entre dépôt local et distant

Lorsque votre dépôt local est rattaché à un dépôt distant (hébergé par une plateforme tel que GitHub), vous pouvez publier vos modifications dans ce dépôt distant, à travers la commande `push`. La commande `pull` permet de récupérer les modifications publiées par vos collègues sur le dépôt distant, vers votre dépôt local. Si vous ne possédez pas de dépôt local et que vous souhaitez intégrer un projet déjà lancé sur un dépôt distant, vous pouvez cloner le dépôt distant vers votre espace de travail local, à travers la commande `clone`. La commande `clone` ne se fait qu'une seule fois pour copier le travail du dépôt distant. Si vous changez d'espace de travail (en changeant d'ordinateur par exemple), vous devez cloner le dépôt à nouveau. Pour cloner le dépôt, la commande suivante doit être exécutée : `git clone URL_depot_distant`. La figure 1.1 montre le lien entre le dépôt local et le dépôt distant avec le sens de déplacement de l'information en fonction de la commande.

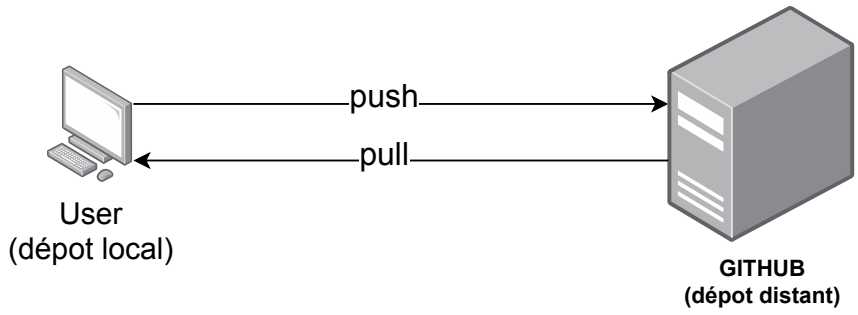


Figure 1.1 – Sens des communications entre le copie locale et la copie distante sur le dépôt Git en fonction de la commande Git.

Pour effectuer un `push`, vous devez d'abord faire un commit. C'est ce commit (qui inclut donc tous les fichiers que vous souhaitez publier) qui sera alors envoyé vers le dépôt distant.
Quant au pull, il se fait en deux étapes :

- Git effectue un **Fetch** : Il observe les modifications effectués sur le dépôt distant. Si aucune modification n'a été faite, Git vous préviendra par le message : "Dépôt à jour". Certains logiciels tel que GitHub Desktop font le fetch automatiquement, en général toutes les 2min.
- Si des modifications ont été détectés, Git vous propose de faire un **pull**. Le pull importera les modifications faites sur le dépôt distant, vers votre dépôt local.

Il peut arriver que Git détecte un conflit entre vos fichiers locaux et ceux du dépôt distant, durant l'étape du pull. Un conflit Git survient lorsque deux modifications incompatibles sont apportées à la même partie d'un fichier entre le travail local et celui d'un dépôt distant. Git ne sait alors pas laquelle choisir automatiquement, et demande de résoudre le conflit manuellement.

3 Gestion de conflit

Admettons qu'il existe un fichier **exemple.txt** contenant :*"Bonjour le monde"*. Deux personnes A et B travaillent en parallèle, tel que présenté sur la Figure 1.2 :

- La personne A modifie le fichier *exemple.txt* et commit sa modification, **sans la pusher tout de suite** : *"Bonjour le soleil"*.
- La personne B fait de même avec le même fichier, mais **push sa modification avant la personne A** : *"Bonjour la lune"*.

Lorsque la personne A va effectuer un push, Git va l'arrêter, effectuer un pull avant, détecter que la personne B a fait une modification avant lui. A ce moment là, Git va détecter que la même ligne a été changée de deux manières différentes, et il va **lever un conflit**, que la personne A devra résoudre.

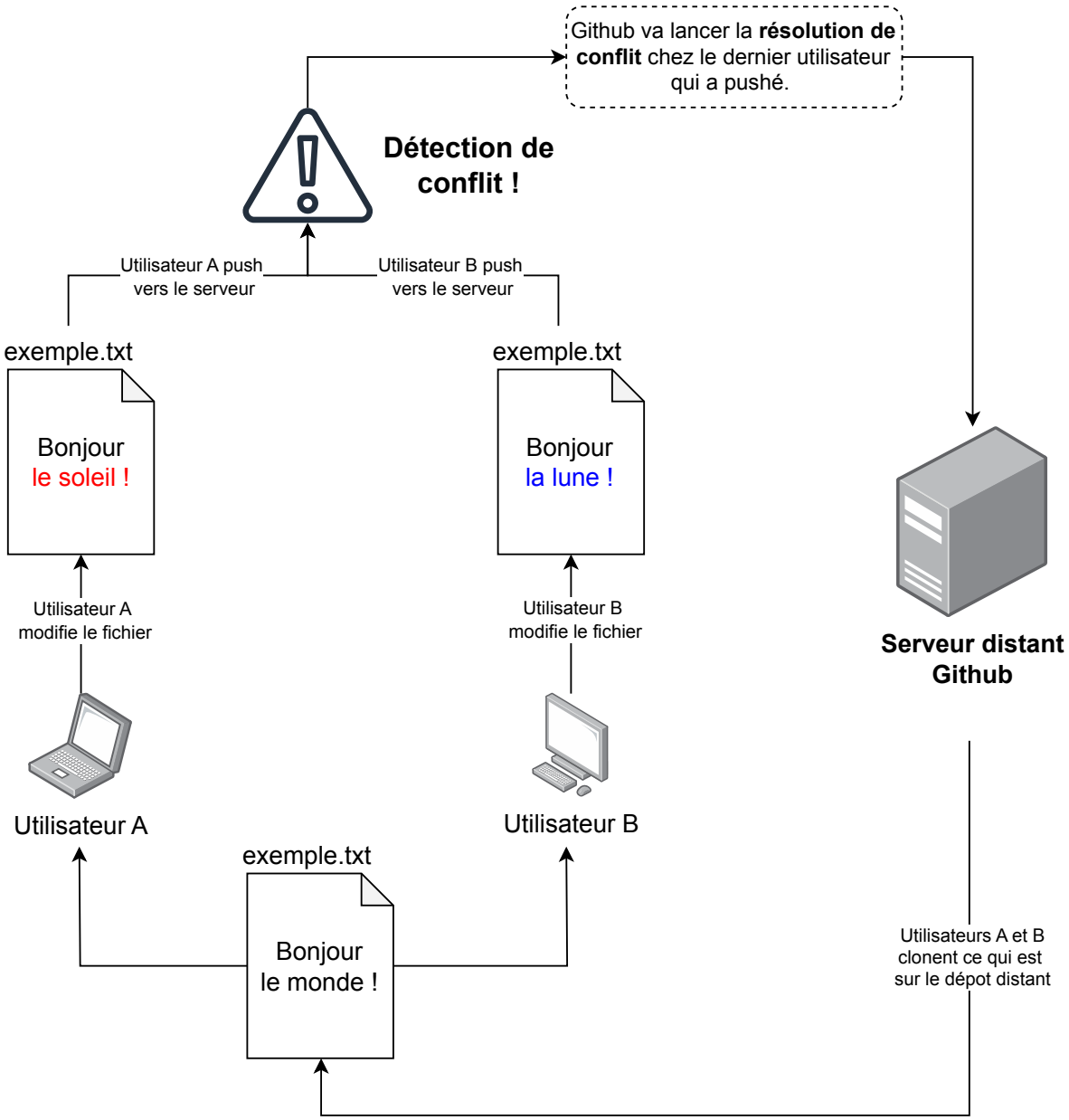


Figure 1.2 – Les utilisateur A et B ont cloné le dépot distant sur le serveur Github, contenant un fichier "exemple.txt". Les utilisateurs A et B apportent chacun une modification, mais en pushant, un conflit est levé. Si B a pushé en premier, c'est A qui devra résoudre le conflit.

Git va marquer le fichier avec des balises de conflit :

```
<<<<<<< HEAD
Bonjour le soleil
=======
Bonjour la lune
>>>>>>> remote-origin
```

Les balises donnent les deux versions du fichier posant problème : La partie en haut est la modification locale (de A), et en bas, la modification du dépôt distant (de B).
Pour résoudre le conflit, il suffira d'ouvrir le fichier avec n'importe quel éditeur (Visual Studio Code et Netbeans peuvent détecter ces balises), de retenir la version que vous souhaitez garder, enregistrer et commit/push cette modification.
Ici, admettons que la personne A ouvre un éditeur et décide de garder les deux versions, il n'aura qu'à faire sa modification en supprimant les balises : "Bonjour le soleil et la lune ". Il ne lui restera qu'à commit et push.

Remarque

Lors de la résolution du conflit, n'hésitez pas à prévenir vos camarades des versions que vous choisissez, ou essayez de prendre une version qui ne risque pas de casser le travail de vos collègues. Cependant, si cela devait arriver, ce n'est pas grave, gardez en tête que vous avez un historique commun, et que vous pouvez donc revenir en arrière.

4 Cas particulier des conflits locaux

Il peut arriver qu'au cours de votre travail, vous souhaitez récupérer les modifications distantes, grâce à la commande `pull`, mais que vous avez déjà effectué des modifications sur votre dépôt local. Le pull peut alors entrer en conflit avec vos modifications actuelles, qui peuvent alors être écrasées si cette situation n'est pas gérée correctement.
La commande `stash` permet de sauvegarder temporairement des modifications en cours (non commitées) pour revenir à un état propre du projet, sans perdre son travail. Il sera ensuite possible de restaurer ces modifications avec la commande `stash pop` (pour retirer la sauvegarde de la pile de sauvegarde) ou `stash apply` (pour restaurer les modifications sans toucher à la pile).
Pour résumer, pour effectuer un pull sans perdre des données, il faudra, dans cet ordre :

1. Sauvegarder ses modifications locales grâce à `stash`.
2. Récupérer les modifications distantes grâce au `pull`.
3. Restaurer ses modifications locales grâce à `stash pop` ou `stash apply`.

GitHub Desktop

GitHub Desktop [1] est une application graphique développée par GitHub qui permet de gérer facilement des dépôts Git sans avoir à utiliser la ligne de commande. Elle simplifie des actions courantes comme cloner un dépôt, créer des branches, faire des commits, résoudre des conflits, et publier du code sur GitHub, le tout via une interface intuitive, tel que le montre la Figure 1.3.
Officiellement, GitHub Desktop est uniquement **compatible avec Windows et macOS**.

Pour les utilisateurs Linux, il n'existe pas de version officielle, mais plusieurs alternatives non officielles et communautaires sont disponibles, comme Shiftkey's fork [2] ou des versions empaquetées via Flatpak et AppImage, qui permettent d'utiliser une version similaire sur les distributions Linux.

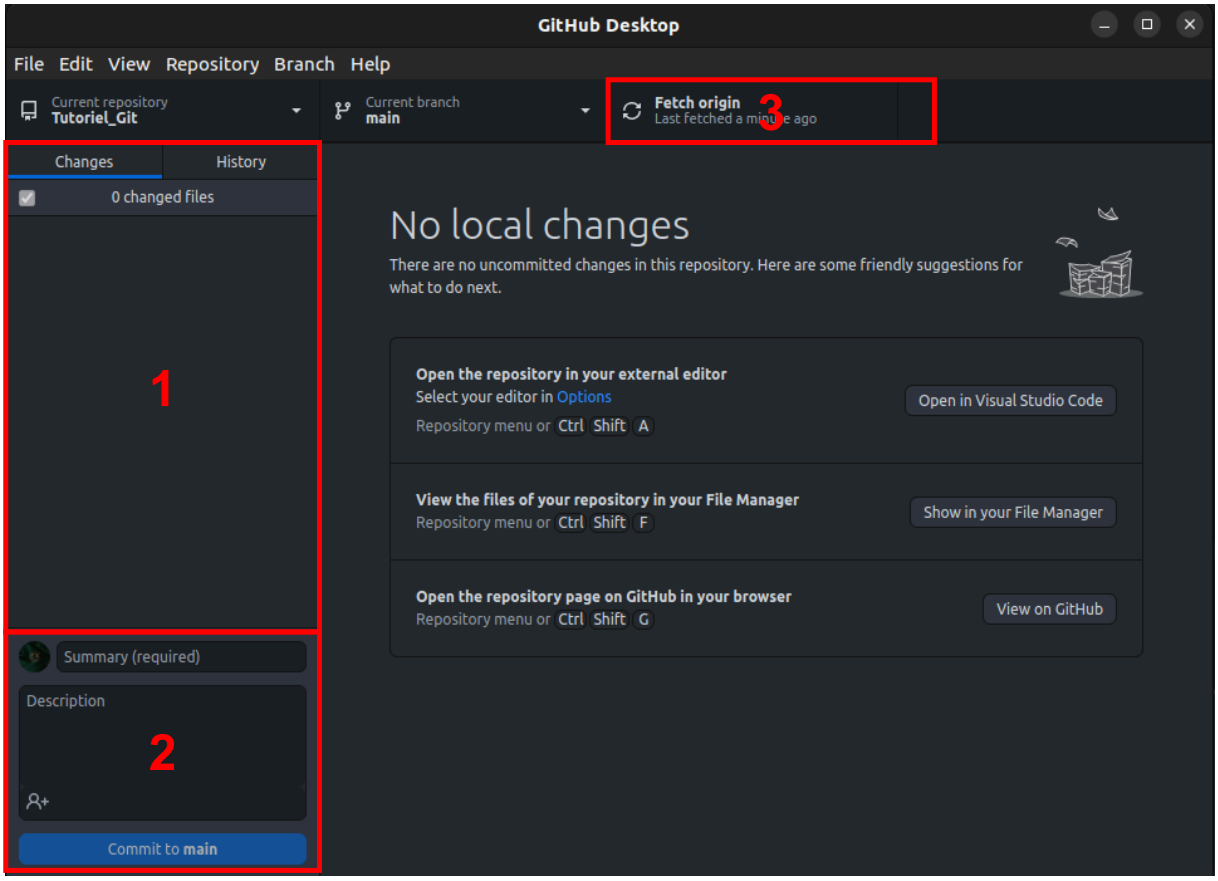


Figure 1.3 – La zone 1 à gauche montrera l'ensemble des fichiers modifiés dans le dépôt local, qui n'ont pas encore été commité. Cocher les fichiers affichés revient à faire la commande *add*. La zone 2, juste en dessous, est la zone de commit. Il suffit de donner un nom au futur commit et de le valider grâce au bouton "Commit to main". Pour finir, la zone 3 permet de push, mais aussi de fetch et pull (le bouton changera de nom selon ce qui est possible de faire).

1. https ://github.com/apps/desktop
2. https ://github.com/shiftkey/desktop

1 Protocole usuel d'utilisation

a Créer et cloner un dépôt distant

1 La première chose à faire lorsque vous souhaitez démarrer un projet de zéro est de créer un dépôt distant sur une plateforme tel que Github ou Gitlab. Pour Github, allez sur votre compte github (vous devez vous inscrire avant), dans l'onglet "Repositories", et créez un nouveau dépôt grâce au bouton bleu "New".

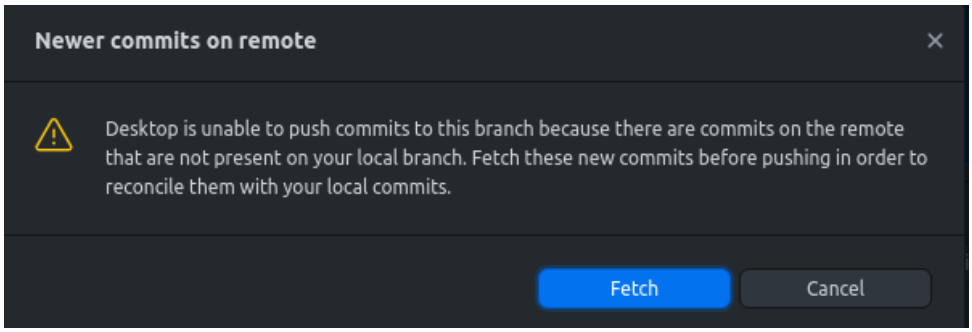
2 Après avoir donné un nom et réglé quelques paramètres, votre dépôt distant est à présent crée, et devrait posséder un premier fichier "README.md". La deuxième étape est de cloner ce dépôt distant localement. Pour cela, ouvrez Github Desktop. En haut à gauche, cliquez sur "File" > "Clone repository...". Une fenêtre s'ouvre, avec un onglet "URL" en haut à droite. Cet onglet vous donne la possibilité de cloner le dépôt distant via son lien URL. Le dépot local sera crée au chemin indiqué lors de sa création, en général dans "Documents" > "Github".

Remarque

Lors de la création du dépôt distant, il est vivement conseillé de cocher l'option "Add a README file", et "Add .gitignore" (avec le langage que vous allez utiliser, comme Java).

b Gestion des conflits (merge)

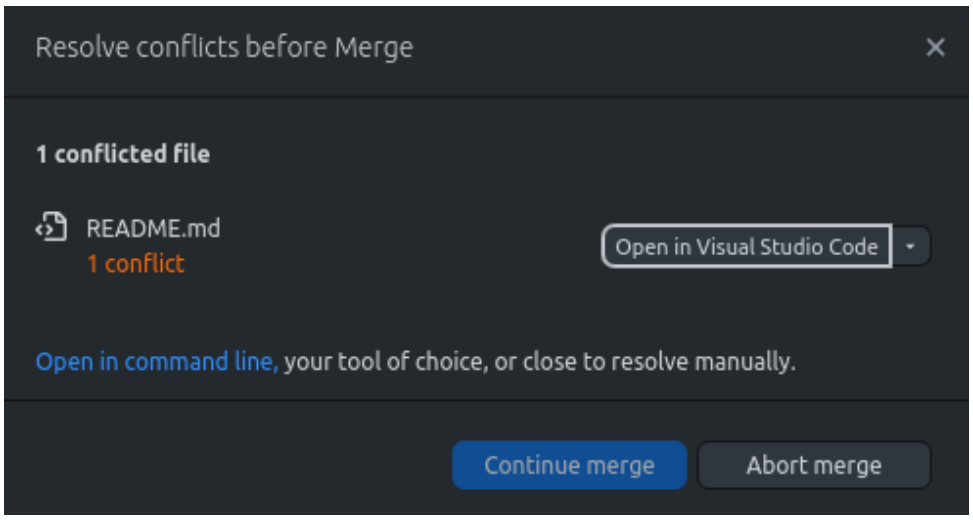
1 Comme expliqué précédemment, il est possible que vous ayez commité des changements alors que vos collègues ont déjà push avant vous, sans que vous le sachiez. La fenêtre de la Figure 1 vous préviendra de ces changements dans le dépôt distant, et effectuera un fetch + pull.

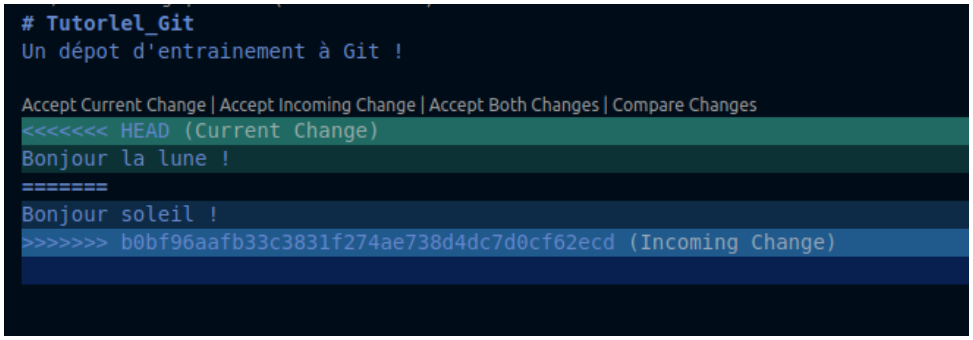


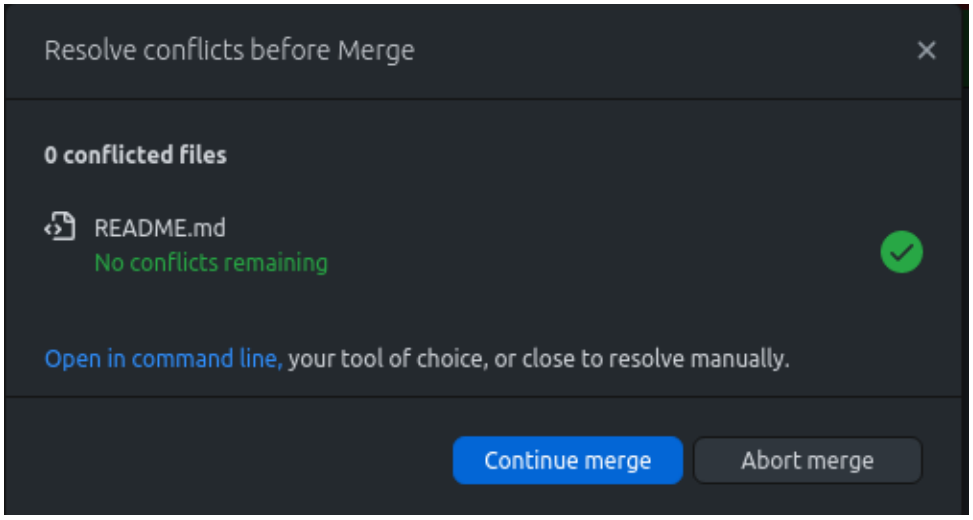
2 Si le changement que vous avez apporté localement entre en conflit avec les changements du dépôt distant, une autre fenêtre tel que affiché sur la Figure 2 vous préviendra et vous proposera de corriger ces conflits sur l'éditeur de votre choix, par défaut, il choisira Visual Studio Code.

3 Ouvrez le/les fichier(s) conflictuels sur l'éditeur de votre choix, les balises présentées dans la section précédente (voir Figure 3) seront présentes, avec la possibilité de choisir votre version (current change), la version venant du dépôt distant (incoming change), ou une version qui fusionne les deux options.

4 Un fois tous les conflits résolus, une nouvelle fenêtre telle que présentée dans la Figure 4 s'ouvre sur Github Desktop pour vous confirmer qu'un commit contenant la résolution de tous les conflits (le merge) est à présent possible, et que vous pouvez à présent pusher vos modifications.



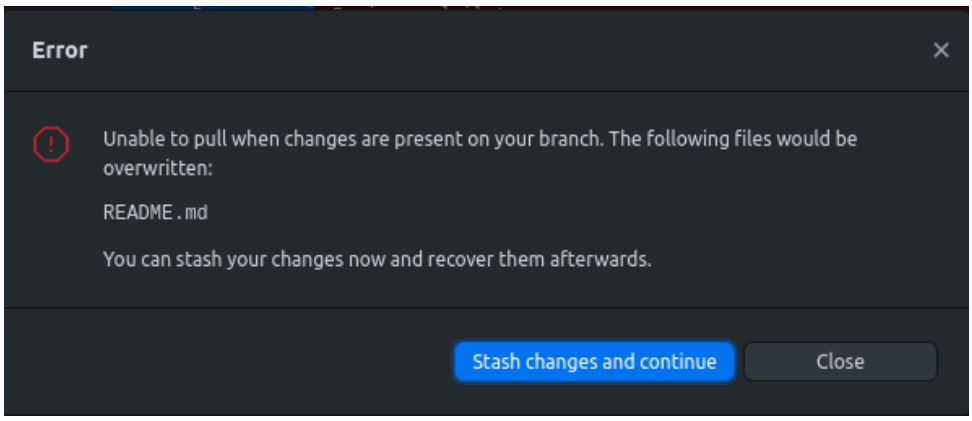




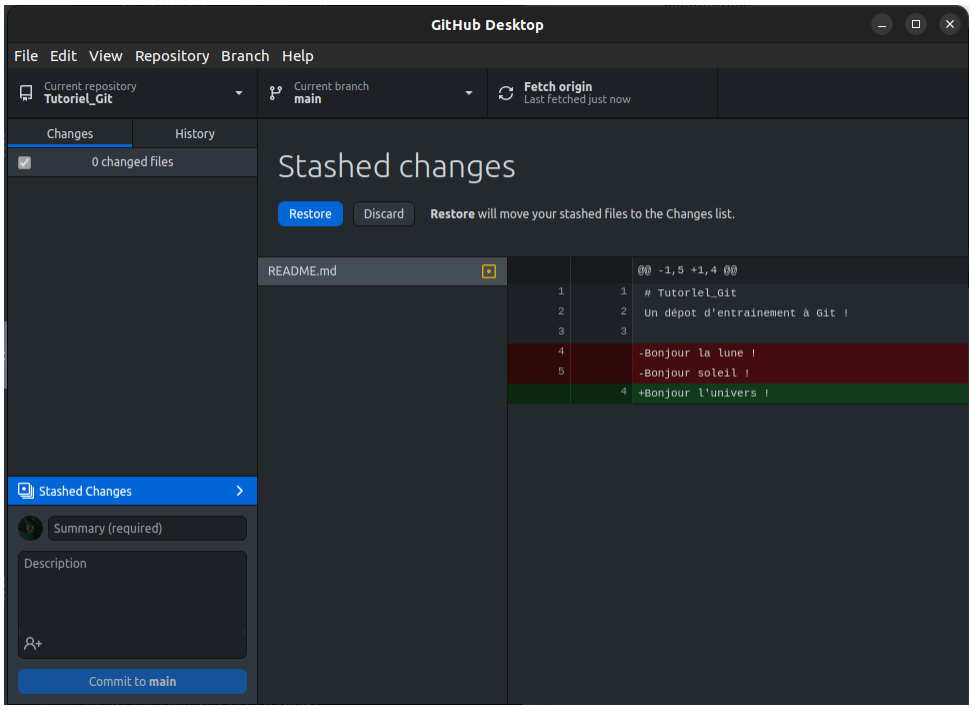
C Gestion de conflits locaux (stash)

1 Dans le cas où vous avez effectué des modifications **non commités**, et que vous souhaitez pull les modifications distantes, il peut arriver que ce pull soit en conflit avec vos modifications locales. Si cela arrive, Github Desktop vous prévient et vous propose de sauvegarder localement vos modifications, via la commande `stash`. (cf Figure 1)

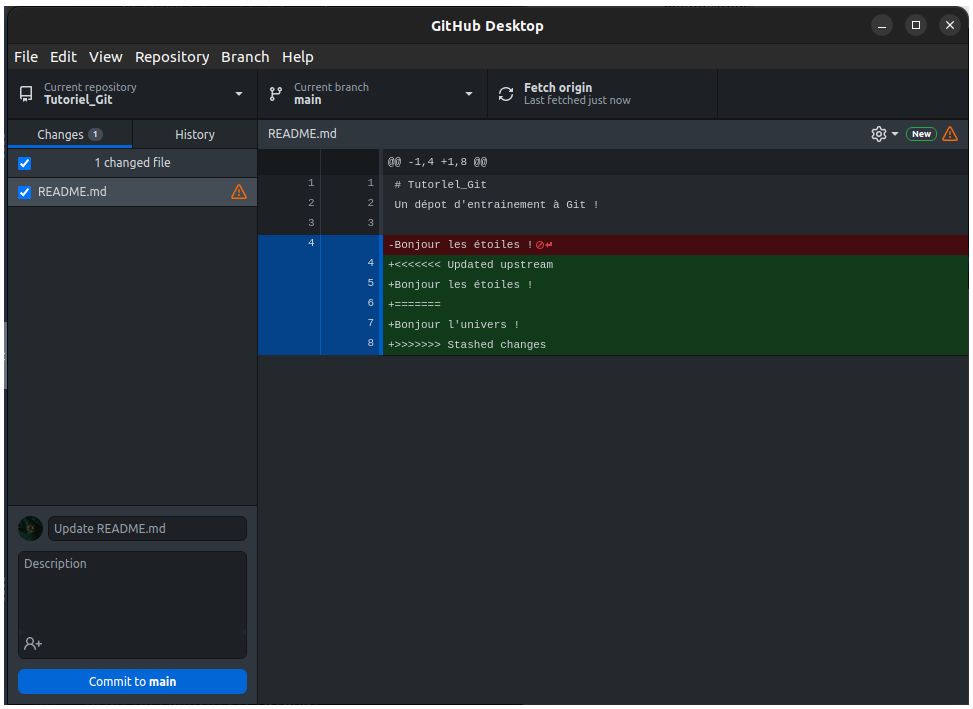
2 Vos modifications sont alors sauvegardées et **mis de coté**, et votre dépôt est revenu à la version du dernier commit (vos modifications locales ont été enlevés). C'est souvent à ce moment là que les utilisateurs novices pensent avoir perdu leur travail, mais n'ayez crainte, vos modifications ont bien été sauvegardés, il ont juste été mis de coté. Github Dsktop vous signale que le `stash` a bien fonctionné en vous montrant les sauvegardes faites, sur la Figure 2. **Faites votre pull**



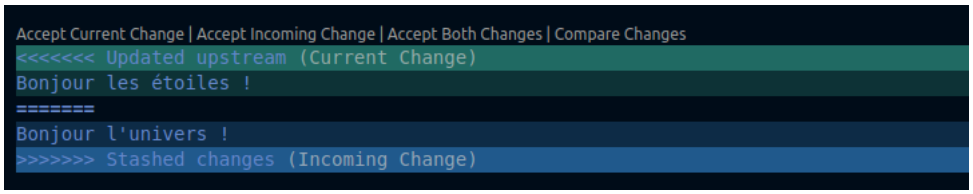
avant de restaurer votre stash ! Sinon votre sauvegarde aura servi à rien.



3 **Après avoir fait votre pull**, et donc chargé les modifications distantes, votre dépôt est à jour. A partir de ce moment, vous pouvez restaurer vos modifications locales (le `stash pop`) en cliquant sur le bouton bleu "Restore" sur la Figure 2. Cela remettra vos modifications locales en place. Cependant, il peut arriver que vos modifications locales et les modifications distantes entrent en conflit. Si cela arrive, Github Desktop vous signalera les fichiers conflictuels (voir la Figure 3).



4 Il suffira d'ouvrir sur l'éditeur de votre choix les fichiers conflictuels : les balises tel que déjà présentés dans le cas d'un conflit classique (merge) seront visibles, vous pourrez alors choisir de la même manière la version de votre choix (cf Figure 4).



5 Une fois que tous les conflits ont été résolus, Github Desktop vous prévient. Vous pouvez à présent poursuivre votre travail ou commiter vos changements comme vous le souhaitez.

Bibliographie