Gestion de branches et de conflits

Dans ce TP, nous allons aborder les notions suivantes :

- La création et le changement de branches.
- L'intégration des modifications, c'est-à-dire la fusion de branches.
- La réécriture d'historique.
- La gestion de conflits de fusion.

Prérequis : avoir fait le TP précédent. En particulier :

- Être à l'aise avec les commandes de base de Git.
- Avoir configuré un alias `git l` similaire à celui du TP précédent.
- Avoir configuré son éditeur Git. Si vous pouvez savez des commits sans l'option `-m` de `git commit`, tout est bon pour vous.

1 Créer des branches

Placez vous dans un répertoire spécifique à ce TP, comme par exemple `${HOME}/2-git-branches`.

Créez un dépôt Git vide `exo1` puis créez-y trois commits — jetez un œil à `man git commit` pour voir les effets de l'option `--allow-empty`, qui peut être pratique ici. Listez les branches locales de votre dépôt grâce à la commande `git branch` sans paramètres, qui devrait vous dire que seule la branche `master` existe pour l'instant.

Créez maintenant une branche nommée `example1` sur votre dernier commit grâce à la commande `git branch <branch>`. Visualisez votre historique Git grâce à la commande `git l` pour vérifier que la branche a été créée au bon endroit. Remarquez que votre branche actuelle est toujours `master` puisque `HEAD` pointe vers `master` — ce que vous pouvez également vérifier avec `git status` ou `git branch`.

Placez-vous maintenant dans votre branche nouvellement créée grâce à `git checkout example1`. Créez un nouveau commit, puis visualisez votre historique Git. Que s'est-il passé pour la branche `example1` ? Et pour `HEAD` ?

Nous venons de voir comment créer une branche puis comment s'y déplacer avec la combinaison de commandes `git branch <branch>` suivie de `git checkout`

`<branch>`. Cette opération est tellement courante qu'elle est faisable en une seule commande : `git checkout -B <branch>`. Créez une branche `example2` et placez-vous y grâce à cette seule commande.

Enfin, il est aussi relativement courant de vouloir créer une branche ailleurs que sur `HEAD`. C'est ce que permet le paramètre optionel `<start-point>` dans `git branch <branch> [<start-point>]` et `git checkout -B <branch> [<start-point>]`. Créez une branche `example3` à partir du commit vers lequel `master` pointe grâce à une de ces commandes, puis visualisez votre historique pour vérifier que votre branche pointe au bon endroit — l'option `--all` de `git log` permet de voir toutes les branches.

2 Intégrer des modifications avec `git merge`

Placez vous de nouveau dans un répertoire spécifique à ce TP (par exemple `${HOME}/2-git-branches`). Dans cet exercice et les suivants, nous allons travailler à partir d'un dépôt Git existant plutôt que de partir d'un dépôt vide.

Récupérez une copie locale du dépôt grâce à la commande suivante : `git clone https://gitlab.com/git-course-mpoquet/exercise-branch1.git`. **Vous pouvez refaire cette manipulation pour repartir d'une copie fraiche du dépôt**, ce qui sera utile pour tester les diverses manières de fusionner des branches ou en cas d'erreur de manipulation.

Jetez un œil aux fichiers de dépôt (*e.g.*, en lançant `tree`) et visualisez l'historique du dépôt avec `git l --all`. Il devrait contenir trois branches :

- `master`, la version maintenue du dépôt à jour.
- `doc-no-conflicts`, qui ajoute une information dans la documentation.
- `usable-as-lib`, qui ajoute la fonctionnalité de se servir de `hello.py` en tant que bibliothèque python en plus d'être un script exécutable.

Ici, j'ai fait tous ces commits mais dans un cas d'utilisation réel, vous pouvez vous dire que les branches `usable-as-lib` et `doc-no-conflicts` ont été faites par différents contributeurs. On souhaite intégrer ces branches dans `master`. Une différence entre ces deux branches est que `doc-no-conflicts` est dite fast-forward par rapport à `master` puisqu'elle est placée directement *au-dessus* de `master`, ce qui n'est pas le cas de `usable-as-lib`.

Fusionnons maintenant `doc-no-conflicts` dans `master`. Pour cela, assurez-vous d'être dans `master` puis lancez `git merge --no-ff doc-no-conflicts`[1]. Observez l'historique résultant après cette commande : les commits de `doc-no-conflicts` ont été intégrés dans master grâce à un commit de fusion

[1] Les branches de ce dépôt ne sont que des branches distantes (*remote branches*) si vous partez d'un clone frais du dépôt. Vous pouvez créer la branche locale correspondante en vous plaçant dedans (`git checkout <branch>`) ou faire directement référence à une branche distante dans les commandes de fusion (`git merge <remote>/<branch>`). Note : `git remote` liste le nom de vos *remotes*.

(*merge commit*). Ce commit de fusion devrait avoir pour parent principal son ancêtre direct dans master (`0aaccd9`) et pour parent secondaire le commit vers lequel `doc-no-conflicts` pointe (`2a69873`) — ce que vous pouvez vérifier grâce à `git l --first-parent`, qui n'affiche que le parent principal de chaque commit.

Nous allons maintenant voir les différents effets possibles de `git merge` selon les cas. En partant à chaque fois d'une copie fraiche de dépôt, fusionnez une des branches avec différentes options de `git merge` (`--no-ff`, `--ff-only`, `--ff`) afin de remplir le tableau suivant. Indiquez dans chaque case si la fusion crée un *merge commit* ("oui"), n'en crée pas ("non") ou échoue ("échec").

	paramètre de `git merge`		
	`-no-ff`	`-ff-only`	`-ff`
fast-forward	oui		
non fast-forward			

Table 1: La fusion crée-t-elle un *merge commit* dans chaque cas ?

Quelle option permet de **toujours** créer un *merge commit* ?
Quelle option permet de ne **jamais** en créer ?

Par défaut, appeler `git merge` sans paramètre pour contrôler la création de *merge commit* est similaire à lui donner le paramètre `--ff`. Ce comportement par défaut peut être changé dans votre configuration Git, par exemple comme indiqué dans cette question stack overflow — voir la partie *configuration* de `man git merge` pour plus d'informations.

3 Intégrer des modifications avec `git rebase`

Une autre commande très importante pour intégrer des modifications est `git rebase`, qui permet comme son nom l'indique de changer la base d'une branche. Repartez d'un dépôt Git frais comme indiqué dans l'exercice précédent : `git clone https://gitlab.com/git-course-mpoquet/exercise-branch1.git`.

Nous allons illustrer le cas le plus courant d'appel de `git rebase` en voyant une autre manière d'intégrer la branche `usable-as-lib` dans `master`. Nous allons pour cela faire passer `usable-as-lib` en fast-forward par rapport à `master`.

Visualisez votre historique Git et notez que le commit dans la branche `usable-as-lib` est `a784b13`. Après vous être assurés d'être dans la branche `usable-as-lib`, lancez la comande `git rebase master`. Visualisez de nouveau votre historique Git. Que s'est-il passé pour la branche `usable-as-lib` ? Le commit précédent (`a784b13`) est-il toujours accessible dans l'historique ? Vous pouvez ensuite fusionner votre branche dans `master` grâce à `git merge`.

`git rebase` est souvent utilisé pour simplifier l'historique d'une branche avant de l'intégrer dans un dépôt, ce qui permet d'avoir un historique final plus lisible.

Ceci est possible en réécrivant l'historique, c'est-à-dire en créant de nouveaux commits qui font les mêmes modifications que les commits d'origine[2]. **Il est cependant fortement déconseillé de réécrire un historique publique**, car cela pose des problèmes de reproductibilité (les commits d'origine avant d'être réécrits peuvent être perdus après un appel de *garbage collection* sur le serveur Git si aucune branche ne pointe vers eux) et est très embêtant pour les utilisateurs qui ont une copie locale de vos branches publiques avant réécriture.

3.1 Rebase interactif

Nous avons vu l'appel le plus simple de `git rebase` dans la partie précédente. Cette commande est très puissante mais a aussi des cas d'utilisation assez complexes — vous pouvez jeter un œil aux schémas de `man git rebase` qui parlent du paramètre `--onto` si vous n'en êtes pas convaincus.

Nous allons ici montrer la puissance de réécriture d'un `rebase` interactif, en montrant des opérations courantes qu'il permet. Placez-vous dans une copie fraiche du dépôt, assurez-vous d'être dans `master` et tapez `git rebase -i --root`, qui permet de réécrire l'historique de manière interactive (`-i`) depuis le début de dépôt (`--root`) jusqu'à `HEAD`. Cette commande va ouvrir votre éditeur de texte (voir TP précédent pour le configurer dans Git) afin de vous demander ce que vous souhaitez faire de chaque commit. Les commits sont triés par ordre dans lequel ils vont être réécrits. Comme indiqué en commentaire du fichier ouvert dans votre éditeur, vous pouvez faire différentes actions sur chaque commit. Ici, nous allons faire les modifications suivantes :

- Simplifier le fichier de licence dans le commit initial `ac1345c`.
- Mettre "doc: initial README" en message de commit de `74d41b0`.
- Fusionner les commits `882c921` et `0aaccd9` en un seul commit.
  Cela se fait via un `squash` sur le dernier des deux commits (`0aaccd9`).

Ce qui devrait se traduire par ce contenu dans votre éditeur de texte :

```
edit ac1345c initial commit
reword 74d41b0 doc: README
pick 882c921 code: first version
squash 0aaccd9 code: fix shabang
```

Après avoir enregistré le fichier et quitté votre éditeur de texte, le `rebase` interactif commence. À chaque étape, `git rebase` devrait vous dire ce que vous êtes en train de faire et ce qu'il reste à effectuer. Notez que `git rebase --abort` permet d'annuler toute l'opération de `rebase`, ce qui est très utile en cas de panique.

1. Tout d'abord, il devrait lancer l'édition du commit initial. Il va pour cela vous replacer dans votre terminal dans un état spécial de rebase interactif — lancez `git status` pour voir l'état dans lequel vous êtes. Puisque vous êtes

[2] C'est le rôle de `git cherry-pick`. Référez-vous à son manuel pour en savoir plus.

en mode édition d'un commit, vous pouvez librement modifier des fichiers et ajouter ces modifications au commit. Ne gardez que la première et la dernière ligne du fichier `UNLICENSE` et sauvegardez le fichier. Réferez-vous à `git status` pour savoir comment enregistrer votre modification puis pour savoir comment continuer vos opérations de `rebase`. (Git va probablement appeller à ce moment-là votre éditeur pour vous proposer de modifier le message de commit. Dans ce cas, laissez le contenu inchangé, enregistrez le fichier et fermez votre éditeur de texte.)

2. Ensuite, il devrait appeler votre éditeur de texte pour changer le message de commit de `74d41b0`. Écrivez donc "doc: initial README" puis enregistrez et quittez le fichier.
3. Il devrait ensuite appeler votre éditeur de texte pour choisir un message de commit pour les deux commits à fusionner. Vous pouvez garder seulement le message de commit du premier fichier, pour faire comme si on avait directement écrit ce fichier avec le bon shabang du premier coup. Supprimez donc les autres lignes non commentées du fichier, enregistrez-le et quittez votre éditeur de texte.

Une fois le `rebase` interactif terminé, jetez un œil à votre historique Git pour vérifier qu'il comprend bien les modifications souhaitées.

4 Comment gérer un conflit ?

Normalement, les appels précédents dans ce TP de `git merge` et `git rebase` n'ont pas créé de conflits. Dans la pratique, des conflits arrivent lorsque plusieurs modifications ont été faites sur les mêmes bouts de code, ce qui arrive relativement fréquemment lorsqu'on travaille à plusieurs en parallèle sur un projet. Nous allons voir les deux manières principales de gérer un conflit en Git.

Tout d'abord, placez vous de nouveau un répertoire spécifique à ce TP, comme par exemple `${HOME}/2-git-branches`. Récupérez ensuite une copie locale du dépôt qui va faire apparaître des conflits en lançant `git clone https://gitlab.com/git-course-mpoquet/exercise-branch2.git`.

Observez le dépôt : jetez un œil à son historique et au contenu des différents fichiers dans chaque branche — rappel : `git show <branch>:<file>` affiche directement de contenu du fichier `<file>` dans la branche `<branch>`.

Essayons maintenant de fusionner `function-politeness` dans `master` grâce à `git merge`. La commande devrait échouer et vous dire qu'un conflit a été détecté, en plus de vous dire sommairement comment le résoudre. Lancez `git status` pour voir quelles commandes effectuer en détails, en plus de vous donner la commande pour annuler la tentative de fusion et revenir dans l'état précédent.

4.1 Résoudre le conflit directement

Résoudre un conflit est tout à fait faisable sans autre outil qu'un éditeur de texte en Git. Lorsque un conflit est détecté, Git vous fait passer en mode résolution de conflits, qui est très proche du mode édition de commit que l'on vient de voir avec un `rebase` interactif.

Par défaut, Git va modifier les fichiers qui ont des conflits pour y faire apparaître les lignes conflictuelles, entourées de guides permettant de voir quelles lignes ont été modifiées dans quelle branche. Ici, un conflit est apparu sur le fichier `hello.py`, affichez donc son contenu. La partie conflictuelle devrait ressembler aux lignes suivantes :

```
<<<<<<< HEAD
    print(f'Hello, {stuff}!')
=======
    print(f'Hi {someone}')
>>>>>>> function-politeness
```

Cet affichage indique que votre version courante (`HEAD`) du fichier utilise `print(f'Hello, {stuff}!')` tandis que celle de la branche function-politeness utilise `print(f'Hi {someone}')`. Ces informations, avec l'aide du code autour du conflit, vous permettent un peu de comprendre ce qui s'est passé et d'éventuellement résoudre le conflit. Cependant, il manque ici une information qui peut être cruciale pour savoir rapidement comment résoudre un conflit sans parcourir l'historique Git : quel était le contenu des lignes conflictuelles dans la base commune de ces deux branches ? Git permet d'afficher cette information, mais il ne le fait pas par défaut.

Recommençons la résolution de conflit en affichant cette information manquante. Annulez la fusion en cours (`git status` devrait vous dit comment faire, `man git merge` le fera sinon), puis configurez le style de conflit de fusion en `diff3` grâce à la commande suivante : `git config --global merge.conflictstyle diff3`. Recommencez ensuite la fusion, qui devrait échouer comme précédemment. Le fichier `hello.py` devrait par contre avoir changé et contenir les lignes suivantes :

```
<<<<<<< HEAD
    print(f'Hello, {stuff}!')
||||||| merged common ancestors
    print(f'Hi {stuff}')
=======
    print(f'Hi {someone}')
>>>>>>> function-politeness
```

L'information ajoutée est ici importante, puisqu'on voit en un coup d'œil quelle modification a été faite dans `HEAD` et quelle modification a été faite dans `function-politeness`, pas uniquement le contenu final des lignes dans chaque branche. Dans `HEAD` le mot `Hi` a été remplacé en `Hello,` et un `!` final a été ajouté, alors que dans `function-politeness` la variable `stuff` a juste été

renommée en `someone`. Une manière efficace de résoudre le conflit est donc de prendre la version de `HEAD` pour conserver son format, en remplaçant la variable `stuff` par `someone`. Modifiez le fichier `hello.py` pour faire cette modification, en supprimant également toutes les lignes de guide (`<<<`, `|||`, `===` et `>>>`) afin que le fichier contienne du Python valide — vérifiez que le fichier `hello.py` marche comme prévu quand vous l'exécutez. Sauvegardez le fichier puis finissez la fusion via des commandes Git (`git status` vous dit comment faire).

4.2 Utiliser un outil de résolution de conflit

Les conflits Git sont souvent résolus directement *à la main* comme dans la section précédente. Git permet aussi d'utiliser un outil de résolution de conflit à la place, ce que nous allons voir maintenant.

Nous allons ici nous servir de kdiff3, un des seuls outils libres de gestion de conflit permettant d'afficher la version de base de fichier. Tout d'abord, installez `kdiff3` s'il n'est pas déjà accessible sur votre machine. Il devrait être *packagé* dans votre distribution Linux favorite. Configurez ensuite Git pour lui dire qu'il doit se servir de `kdiff3` comme outil de résolution de conflits via la commande `git config --global merge.tool kdiff3`.

Repartez d'une copie fraiche de dépôt précédent grâce à la commande `git clone https://gitlab.com/git-course-mpoquet/exercise-branch2.git`, puis tentez de nouveau de fusionner `function-politeness` dans `master`, ce qui devrait encore échouer.

Lancez maintenant `git mergetool`, qui va vous lancer l'interface graphique de `kdiff3`. Cette interface vous affiche simultanément les différentes versions du fichier (la version de base, la version courante et celle à fusionner), en colorant les différences au sein de chaque ligne de chaque version par rapport à la version de base. Le dernier fichier (en bas) est le fichier résultant de la résolution de conflits.

Vous pouvez résoudre le conflit dans `kdiff3` en faisant un clic droit sur les lignes conflictuelles restantes pour lui dire quelle version prendre, puis éditer cette ligne à la main si nécessaire. Si vous jetez un œil aux fichiers de votre dépôt pendant que le `mergetool` s'exécute, vous pouvez voir que les différentes versions du fichier ont été copiées dans des fichiers temporaires, ce qui peut être pratique pour faire différentes opérations sur des fichiers (comme appeler `diff`). Résolvez le conflit dans `kdiff3`, vérifiez que `hello.py` s'exécute comme prévu, puis finissez la fusion via des commandes Git (`git status` est encore là pour vous aider).

5 Points clés à retenir de ce TP

Autour des concepts de Git.

- Comment créer une branche, comment en changer.
- Comment fusionner une branche.
- Comment réécrire un historique.
- Comment résoudre un conflit de fusion.

Autour des commandes de Git.

- `git branch` et `git checkout -B` pour créer des branches.
- `git checkout` pour changer de branche.
- `git merge` et ses options de contrôle de création de *merge commit*.
- `git rebase` pour réécrire une branche.
- `git merge|rebase --abort` pour annuler une fusion/réécriture en cours.