TP2 : Un historique linéaire

Processus de création de version

Un premier commit

Au cours du TP précédent, nous avons manipulé l'état de notre répertoire de travail. Il est maintenant temps d'en sauvegarder une première version.

```
$ cd ~/Bureau/MonDepot
$ git commit
```

Au moment où l'on exécute la commande `git commit` pour créer une version, git ouvre un éditeur de texte permettant de saisir une description des changements effectués. Je vous laisse le soin d'en saisir une selon votre inspiration.

Le format est libre, mais une convention très répandue dans le monde git est de commencer par un résumé court (une ligne de moins de 80 caractères), éventuellement suivi d'une paire de sauts de lignes et une description plus précise. Cette convention est exploitée par de nombreux outils de visualisation, je vous conseille donc de la suivre.

Notez que l'éditeur de texte ouvert par `git commit` contient un récapitulatif de l'état actuel du dépôt, similaire à celui qui est affiché par `git status`. Ce dernier ne sera pas présent dans la description finale. Il vous permet juste de vérifier rapidement les changements qui seront sauvegardés.

La version sera créée lorsque vous quitterez l'éditeur de texte en sauvegardant la description. Mais si vous vous rendez compte que vous avez tapé `git commit` trop vite, pas de panique : il suffit de saisir une description vide (ou de quitter l'éditeur de texte sans sauvegarder vos modifications) pour annuler l'opération.

Nouveau contenu et différences

Après avoir créé un commit, la sortie de `git status` change :

```
$ git status
```

La commande status décrit en effet toujours l'état de l'index et du répertoire de travail par rapport à la dernière version enregistrée (`HEAD`).

Ici, si vous avez suivi fidèlement le TP1, tous les changements du répertoire de travail qui ne sont pas dans la liste "gitignore" ont été enregistrés dans le premier commit. `git status` devrait donc indiquer qu'il n'y a rien de nouveau.

Effectuons maintenant quelques changements :

```
$ echo 'Un premier changement' > MonDossier/Contenu.txt
$ echo 'Un nouveau fichier' > AutreContenu.txt
```

```
$ git status
```

Comme la plupart des gestionnaires de version, git distingue soigneusement les changements apportés à des fichiers sous gestion de version des créations de nouveaux fichiers dans le répertoire de travail.

Pour visualiser le premier type de changement, on peut utiliser `git diff` :

```
$ git diff
```

Cette commande décrit les différences entre le répertoire de travail et l'index, selon un format similaire à celui de l'utilitaire "diff" des systèmes Unix.

Attention : `git diff` ne tient pas compte des fichiers qui n'ont pas été mis sous gestion de version avec `git add`, et ne peut donc pas remplacer totalement l'utilisation de `git status`.

Les habitués d'autres gestionnaires de version noteront que `git add` combine deux fonctions qui sont en général distinctes dans d'autres outils :

- Mettre des fichiers sous gestion de version
- Ajouter des changements survenus dans un fichier à la prochaine version

Dans le modèle de git, ces deux opérations sont unifiées par la notion d'index, qui rend d'un certain point de vue obsolète celle de fichier sous gestion de version. Cependant, cette seconde notion existe malheureusement toujours dans git, comme le piège ergonomique de `git diff` ci-dessus l'illustre.

Aller plus loin avec git diff

`git diff` décrit les changements du répertoire de travail par rapport à l'index (la version en cours de création). Par conséquent, si on ajoute des changements à l'index, ils disparaissent de la sortie de `git diff` :

```
$ git add MonDossier
$ git diff
$ echo 'Un second changement' >> MonDossier/Contenu.txt
$ git diff
```

Ce mode de fonctionnement aide à créer une nouvelle version progressivement :

- On fait un point sur les changements effectués depuis la dernière version
- On décide de ce qui va faire partie de la prochaine version
- On ajoute ces changements à l'index avec `git add`
- On vérifie qu'on n'a rien oublié d'important avec `git diff`
- On crée la nouvelle version avec `git commit`

Mais parfois, on souhaite un autre comportement de la part de `git diff`. On peut ainsi lui demander la liste des changements effectués au sein du répertoire de travail depuis la dernière version enregistrée :

```
$ git diff HEAD
```

Ou bien on peut lui demander les changements présents au sein de l'index, qui seront inclus dans la nouvelle version lorsqu'on appellera `git commit` :

```
$ git diff --cached
```

Il est aussi possible, et souvent très utile, de n'afficher que les changements survenus au sein d'un certain fichier ou dossier avec `git diff <chemin>` :

```
$ git add AutreContenu.txt
$ git commit -m 'Une deuxième version'
$ echo 'Un ajout de contenu' >> AutreContenu.txt
$ git diff
$ git diff MonDossier/
```

Plusieurs autres choses sont à noter dans cet exemple :

- La commande `git commit` possède une option `-m` pour spécifier directement la description, très pratique quand celle-ci est courte.
- Comme `diff`, `git diff` affiche quelques lignes du fichier autour des changements, permettant de comprendre le contexte desdits changements.

Quelques raccourcis

Outre le raccourci `git commit -m` que nous venons de voir, git possède d'autres moyens d'accélérer des opérations courantes.

`git add --all [<fichiers>]`, que nous avons brièvement rencontrée précédemment, ajoute à l'index **tous** les changements d'un jeu de fichiers et de dossiers, y compris d'éventuelles suppressions.

Sans paramètre supplémentaire, cette commande synchronise complètement l'index avec le répertoire de travail. Cela peut conduire à des ajouts involontaires, il est donc recommandé de cibler cette commande sur un dossier.

```
$ rm MonDossier/Contenu.txt
$ git add --all MonDossier/
$ git status
$ git commit -m 'Nous sommes sur une bonne lancée !'
```

`git commit -a` crée un commit avec tous les changements survenus dans les fichiers sous gestion de version. Ce faisant, elle ignore les fichiers qui n'ont pas été mis préalablement sous gestion de version avec `git add`, imitant ainsi la logique d'autres gestionnaires de version comme SVN ou Mercurial.

Si cette logique familière aidera dans leur apprentissage les habitués de ces gestionnaires, elle est globalement piégeuse, et entraîne souvent la création de commits incomplets. J'encourage donc les débutants de la gestion de version à

considérer l'utilisation de `git commit -a` comme une mauvaise habitude qu'il est préférable d'éviter de prendre.

Exploration de l'historique

Visualisation interactive

Nous avons maintenant créé quelques commits, et il est possible d'afficher l'historique de ces derniers avec `git log` :

```
$ git log
```

Si un commit précis de la liste nous interpelle, nous pouvons ensuite demander à git d'afficher des informations détaillées sur ce dernier avec `git show <commit>`, où le commit peut être identifié par son hash.

Ces deux commandes sont extrêmement configurables : on peut demander à `git log` et `git show` d'afficher précisément l'information qu'on veut, dans le format qu'on veut. Mais force est de constater que malgré tous ces efforts, un affichage en ligne de commande devient rapidement difficile à lire dès lors que l'information affichée est un tant soit peu complexe. Un terminal textuel n'est pas vraiment fait pour ça!

Pour cette raison, je vous recommande fortement d'utiliser une interface graphique pour visualiser votre historique. Au cours de cette formation, nous utiliserons pour cela le logiciel `gitg` :

```
$ gitg
```

Vous pouvez constater que `gitg` vous permet d'observer d'un coup d'oeil votre historique. En cliquant sur chaque commit, vous avez aussi accès à des informations plus détaillées : description détaillée, hash, détail des modifications effectuées...

Il existe de nombreux autres logiciels qui peuvent remplacer `gitg` qui n'est disponible que sous Linux.

Premières requêtes

git est fondamentalement basé sur la notion de *différence*, et chaque commit y est définie par les changements qu'il opère vis-à-vis de ses parents.

Si cette vision "locale" des choses est parfois suffisante, il est souvent nécessaire de raisonner par rapport à des versions relativement éloignées dans le passé. Pour ce faire, plusieurs options s'offrent à nous.

Tout d'abord, il est possible d'afficher les changements survenus dans le répertoire de travail depuis une ancienne version avec `git diff <commit>`. Il est également

possible d'afficher la différence entre deux versions enregistrées du projet avec `git diff <commit1> <commit2>`.

Vous aurez peut-être remarqué que nous avons déjà utilisé la première de ces deux commandes, avec l'identifiant `HEAD` qui désigne le dernier commit enregistré. Pour des raisons de concision, nous parlerons désormais de "commit actif".

A tout identifiant de commit, nous pouvons ajouter le suffixe `~[N]` pour désigner le (N-ième) parent de ce commit. Ainsi, `HEAD~` est le parent du commit actif, et `HEAD~2` est le grand-parent de ce commit.

Avec cette syntaxe, nous pouvons demander à git la différence entre le répertoire de travail et le parent du commit actif...

```
$ git diff HEAD~
```

...ou bien les changements du commit actif par rapport à son parent :

```
$ git diff HEAD~ HEAD
```

Nommer un commit

Désigner des commits par des hashes absolus ou par des chemins relatifs à base de HEAD est fastidieux, et le risque d'erreur est élevé. Lorsqu'on doit souvent faire référence à un commit, il est préférable de lui donner un nom plus lisible pour un être humain.

En gestion de version, cela s'appelle une étiquette (*tag*), et on peut en créer une avec la commande `git tag <étiquette> [<commit>]` :

```
$ git tag important
$ git tag papa HEAD~
$ git tag maman important~
$ git tag ancetre HEAD~2
```

Après l'exécution de ces commandes...

- `important` pointe sur le commit actif (`HEAD`)
- `papa` pointe sur le commit avant le commit actif (`HEAD~`)
- `maman` pointe sur le commit avant `important` (donc `HEAD~`, comme `papa`)
- `ancetre` pointe deux commits avant le commit actif (donc sur le commit `HEAD~2`, qui précède le commit doublement étiqueté `papa`/`maman`).

Vous pouvez visualiser ces étiquettes dans `gitg` en redémarrant ce dernier ou en rafraîchissant son affichage avec le raccourci clavier `Ctrl+R`.

Contrairement à `HEAD` et à la branche active, les étiquettes ne bougent pas. Si vous créez de nouveaux commits par la suite, `important` pointera toujours vers le même commit.

Si vous le souhaitez, vous pouvez ajouter une description à votre étiquette, avec l'option `--annotate` de `git tag`. Ces annotations sont souvent utilisées pour indiquer les points du développement d'un logiciel auxquels une version publique a été distribuée, et documenter les nouveautés de cette version :

```
$ git tag --annotate v1.0 HEAD~
$ git show v1.0
```

Vous pouvez aussi afficher la liste des étiquettes présentes dans le dépôt avec `git tag --list`, qui s'abbrévie en `git tag` :

```
$ git tag --list
```

Et enfin, vous pouvez détruire des étiquettes avec `git tag --delete` :

```
$ git tag --delete v1.0 important maman papa ancetre
```

Revenir dans le passé

git nous permet aussi de remonter le temps, et de charger le contenu d'une version préalablement enregistrée dans le répertoire de travail.

Pour que les choses se passent bien, il faut que ledit répertoire de travail soit *propre*, c'est à dire qu'il n'y ait pas de changements en attente. Pour savoir si c'est le cas, on utilise comme toujours `git status` :

```
$ git status
```

Si vous avez suivi le déroulement prévu du TP, ce n'est pas le cas, il y a des changements en attente. Il va falloir s'occuper de ces changements d'abord, par exemple en les sauvegardant sous forme d'un nouveau `commit`.

```
$ git add --all
$ git commit -m 'La version ultime'
$ git status
```

Maintenant, nous sommes prêts. Revenons donc à un ancien commit, par exemple le parent de celui que nous venons de créer, avec la commande `git checkout` :

```
$ git checkout HEAD~
```

A ce stade, git vous affichera un message pour vous avertir que vous n'êtes plus sur une branche, et que si vous créez des commits dans cet état, vous ne pourrez pas facilement les retrouver par le suite. En terminologie git, on désigne cet état par le qualificatif un peu morbide de `HEAD détachée`.

Le reste du message vous explique comment vous pourriez préserver vos changements en créant une autre branche. Nous étudierons cela ultérieurement, vous pouvez pour l'instant ignorer cette partie.

Si vous regardez les fichiers de votre dépôt, vous pouvez constater que leur contenu a changé pour reprendre la valeur qu'ils avaient dans la version précédente. Une vraie machine à voyager dans le temps !

N'hésitez pas à revenir en différents points de votre historique avec `git checkout`, en utilisant `gitg` pour retrouver les identifiants de vos commits. Par défaut, `gitg` affiche l'historique relativement au commit actif, mais vous pouvez lui faire afficher l'ensemble des commits du dépôt en sélectionnant l'option "Tous les commits" dans le menu de gauche.

Quand vous avez fini d'explorer, il est facile de revenir sur la branche `master` :

```
$ git checkout master
```

Annuler un jeu de commits

Il est possible d'annuler l'effet d'un jeu de commits de façon non-destructive avec la commande `git revert`. Par exemple, pour annuler l'effet du dernier commit effectué, on peut utiliser la commande :

```
$ git revert HEAD
```

A plus grande échelle, il est aussi possible d'annuler l'effet de plusieurs commits avec la variante `git revert <commit1> <commit2>...`, ou celui d'une série de commits avec la variante `git revert <commit1>..<commitN>`.

Cette commande est sans risque, car elle fonctionne en créant un nouveau commit qui annule l'effet du commit choisi. L'ancien commit n'est donc pas perdu, il reste présent dans l'historique, et on ne peut donc pas perdre accidentellement des données en utilisant `revert`.

Cependant, cela signifie aussi que `git revert` n'est pas l'outil approprié pour annuler un ajout incorrect au dépôt, comme par exemple l'ajout d'un secret cryptographique ou d'un gros fichier à la gestion de version. Dans ces cas-là, il faudra avoir recours à des outils beaucoup plus invasifs et destructeurs tels que `git filter-branch` ou le "BFG Repo-Cleaner".

Conclusion

Dans ce TP, nous avons vu comment git permet de...

- Construire soigneusement des versions de son travail (*commits*)
- Comparer des versions de fichiers avec `git diff`
- Visualiser l'historique du dépôt
- Désigner les commits au sein de ce dernier de différentes façons
- Accéder à d'anciennes versions de son travail

Avec ces notions, nous pouvons utiliser git pour construire un historique de fichiers linéaire. C'est tout à fait suffisant pour travailler seul sur un projet, et les anciens gestionnaires de version allaient rarement au-delà.

Mais pour collaborer facilement avec d'autres personnes, il va falloir introduire un outil qui permet à git de représenter la notion d'évolution parallèle : la branche. Ce sera le sujet de la prochaine séquence.

Exercices

1. Créez un nouveau dépôt git.
2. Ajoutez quelques fichiers du contenu de votre choix, puis sauvegardez un commit contenant ces fichiers.
3. Modifiez au moins deux des fichers que vous avez sauvegardés, puis créez un commit ne comprenant que les modifications d'un seul fichier.
4. Affichez les modifications qui n'ont pas encore été sauvegardées.
5. Utilisez un raccourci git pour créer un commit contenant les modifications des autres fichiers sans avoir besoin de les nommer dans les commandes.
6. Affichez la différence entre les deux premiers commits que vous avez effectués au sein du dépôt.
7. Vérifiez que le répertoire de travail est propre.
8. Basculez sur le grand-parent du commit actuel, et attachez-y une étiquette.

Antisèche

Dans toutes les commandes qui suit, l'indication `<chemins>` peut être remplacée par un ou plusieurs chemins d'accès vers des fichiers ou des dossiers.

Ajouter des changements à l'index (pour les inclure dans le prochain commit) :

```
$ git add <chemins>
```

. . . en prennant en compte les suppressions de fichier (rejetées par défaut) :

```
$ git add --all <chemins>
```

Créer un *commit*, une version sauvegardée du projet :

```
$ git commit
```

Saisir la description du commit directement dans la commande :

```
$ git commit -m <description>
```

Afficher les changements des **fichiers versionnés** par rapport à l'index :

```
$ git diff
```

. . . par rapport au dernier commit enregistré (commit actif) :

```
$ git diff HEAD
```

`HEAD` désigne le dernier commit enregistré. Pour tout commit `com`, `com~` désigne le commit parent (celui qui le précède) dans l'historique, `com~2` le commit grand-parent (le parent du parent), et ainsi de suite.

Afficher les changements dans l'index par rapport au commit actif :

```
$ git diff --cached
```

Afficher les changements survenus du commit A au commit B :

```
$ git diff A..B
```

Restreindre l'affichage des changements à certain fichiers (ou dossiers) :

```
$ git diff <chemins>
```

Afficher l'historique des commits en ligne de commande

```
$ git log
```

Lancer le visualiseur d'historique `gitg` sans bloquer le terminal

```
$ gitg &
```

Donner un nom à un commit (par défaut le commit actif) avec une étiquette :

```
$ git tag <étiquette> [<commit>]
```

. . . et compléter l'étiquette par une description :

```
$ git tag --annotate <étiquette> [<commit>]
```

Afficher la liste des étiquettes présentes dans le dépôt :

```
$ git tag --list
```

Détruire une ou plusieurs étiquette(s) :

```
$ git tag --delete <étiquettes>
```

Attention : Les commandes suivantes modifient le répertoire de travail et doivent donc être exécutées dans un répertoire de travail propre.

Charger le contenu d'un commit dans le répertoire de travail :

```
$ git checkout <commit>
```

Revenir sur la branche master :

```
$ git checkout master
```

Créer un commit qui annule l'effet d'un ou plusieurs autres commits :

```
$ git revert <commits>
```

Créer un commit qui annule l'effet de tous les commits entre A et B :

```
$ git revert A..B
```

Informations complémentaires

Contrôler finement les ajouts à l'index

Parfois, il arrive que l'on se soit un peu laissé emporter et que l'on ait effectué de nombreuses modifications sur un fichier en oubliant de créer des versions intermédiaires.

Dans de nombreux projets, créer un grand commit fourre-tout regroupant tous ces changements est mal vu, et il est préférable de décomposer ces modifications en plusieurs commits plus simples. Mais comment faire lorsqu'il y a plusieurs changements qui n'ont rien à voir au sein d'un même fichier ?

git fournit pour ce genre de cas l'outil `git add --patch <fichier>`. C'est une variante plus avancée de `git add`, qui permet d'ajouter sélectivement des modifications du fichier à l'index.

Une description complète de cet outil serait difficile à faire par écrit du fait de sa nature interactive, et sort un peu du cadre de cette formation courte. Mais si vous avez de l'avance sur le TP, nous vous invitons à solliciter le formateur pour une démonstration.

Corriger son dernier commit

Au lieu de créer un nouveau commit, on peut aussi vouloir modifier le commit précédent, par exemple pour ajouter des modifications oubliées ou pour changer la description. Tout cela est possible avec l'option `--amend` :

```
$ echo 'Du nouveau!' > NouveauContenu.txt
$ git add NouveauContenu.txt
$ git commit --amend -m 'Une version avec plus de contenu'
```

Les plus attentifs remarqueront peut-être dans la sortie de `git commit --amend` que cette opération modifie aussi le hash hexadécimal qui identifie le commit. Il y a des causes profondes à cela :

- Le hash d'un commit est une fonction déterministe de son contenu. C'est ce qui fait qu'il est improbable que deux commits indépendants sur le réseau aient le même hash, permettant ainsi la gestion de version distribuée.
- Un gestionnaire de version ne modifie jamais directement un commit. Quand on fait `git commit --amend`, git retire l'ancien commit de l'historique et le remplace par un nouveau commit avec nos modifications. L'ancien commit est en réalité toujours présent dans le dépôt, et il existe des moyens d'y remonter en cas de manipulation accidentelle.

`git commit --amend` est un premier exemple de commande qui modifie l'historique. git dispose d'un vocabulaire très riche de commandes de ce type, que nous aurons l'occasion d'explorer davantage au cours de la formation.

Annuler des manipulations accidentelles

L'une des fonctions fondamentales d'un gestionnaire de version étant de pouvoir revenir en arrière en cas de problème, la plupart des opérations git peuvent être annulées. Voici une première série de commandes d'annulation :

- Ajout de modifications à l'index → `git reset [<fichiers>]`
  - D'anciennes versions de git distinguaient l'ajout de modifications à l'index de ajout de fichiers à la gestion de version. Avec elles, il faut utiliser `git rm --cached <fichiers>` dans le second cas.
- Création d'un commit → `git reset --soft HEAD~`
- Création d'un commit + ajouts préalables à l'index → `git reset HEAD~`
- Création d'un commit + ajouts préalables à l'index + modifications préalables du répertoire de travail (DANGER) → `git reset --hard HEAD~`

Les pouvoirs d'annulation de git se limitent malheureusement aux données qui ont été versionnées. Les opérations sur le répertoire de travail sont en général irréversibles. C'est une des raisons pour lesquelles il vaut mieux créer fréquemment des commits simples que créer rarement des commits complexes.

La seule chose qu'il est facile de faire dans le répertoire de travail, c'est de ramener des fichiers du répertoire de travail à l'état enregistré dans `HEAD`, avec la commande `git checkout -- <fichier>`. Comme toute opération sur le répertoire de travail, cette commande est irréversible, donc dangereuse, et doit être utilisée avec prudence pour éviter des pertes de données.

Mettre des changements de côté

Parfois, lorsqu'on souhaite se promener dans l'historique avec `git checkout`, devoir sauvegarder les modifications de son répertoire de travail est ennuyeux. On n'est pas encore prêt à créer un commit qui représente une version cohérente du projet, et on n'en voit pas la nécessité alors qu'on veut "juste" aller jeter un coup d'oeil à l'histoire du dépôt.

Pour ce genre de cas, git fournit un outil bien pratique appelé `git stash`. Cet outil prend toutes les modifications actuelle et les range dans un endroit à part du dépôt (en fait une branche jetable). Démonstration :

```
$ echo 'Demandez le programme !' >> AutreContenu.txt
$ git status
$ git stash
$ git status
```

La liste des modifications sauvegardées prend la forme d'une pile, que l'on peut consulter avec la commande `git stash list` :

```
$ git stash list
```

Lorsqu'on est prêt à reprendre le travail sur cette version du répertoire de travail, il suffit d'utiliser la commande `git stash pop` pour restaurer la dernière version sauvegardée du répertoire de travail :

```
$ git stash pop
```

Comme avec git checkout, il est préférable de n'effectuer cette opération que sur un répertoire de travail propre.

Autres outils graphiques

Une liste relativement complète d'outils graphiques pour visualiser et manipuler des dépôts git est disponible sur https://www.git-scm.com/downloads/guis .

Lorsqu'il doit développer sous Windows ou macOS, l'auteur y utilise généralement SourceTree. Un collègue utilisateur de macOS lui a également recommandé Fork sur cette plate-forme.