Premiers pas en Git

Dans ce TP, nous allons aborder les notions suivantes :

- L'installation de Git.
- La création de dépôt Git et de commits.
- Naviguer dans un historique Git, faire des recherches et corriger des erreurs.

1 Installer Git

Git est un outil standard et est très bien *packagé*. La manière la plus classique de l'installer est d'utiliser le gestionnaire de paquets de votre distribution Linux. Voici quelques exemples courants :

- Arch Linux et dérivés : `pacman -S git`
- Debian et dérivés : `apt install git`
- Nix : `nix-env -i git`

La page officielle de téléchargement de Git recense d'autres commandes selon votre gestionnaire de paquets.

Vous pouvez également compiler Git vous-mêmes si besoin. Le code source de Git est accessible ici. Les instructions pour une installation depuis des sources sont ici.

1.1 Vérifier que votre installation est fonctionnelle

Vous pouvez lancer des commandes Git pour vérifier qu'il est bien installé dans votre environnement — par exemple `git --version`.

Avoir accès au manuel de Git est également très important, ce que vous pouvez vérifier avec `man git status`. Note : `man` appelle votre *pager* (probablement `less` par défaut), quitter la page du manuel se fait donc de la même manière que pour quitter votre *pager* (en appuyant sur 'q' pour `less`).

2 Créer un dépôt et faire des commits

Placez vous dans un répertoire spécifique à ce TP (par exemple `${HOME}/1-git-intro`).

2.1 Créer un dépôt vide

Vous pouvez créer un dépôt Git vide par une des deux manières suivantes :

- `git init <DIR>`, où `<DIR>` est le nom du répertoire que vous voulez créer. Ici, on peut prendre `<DIR>=exo1`.
- OU en créant le répertoire et en vous plaçant dedans (`mkdir -p <DIR> && cd <DIR>`) puis en tapant `git init`.

Afin de vérifier que votre dépôt a bien été initialisé, placez-vous dans son répertoire puis tapez `git status`, qui devrait vous dire que vous êtes sur la branche `master` qui ne contient aucun commit pour l'instant.

2.2 Créer des commits

La création de commits en Git se fait en trois étapes :

1. Créer, modifier ou supprimer des fichiers
2. Enregistrer les modifications à inclure dans le prochain commit
3. Créer un commit à partir de la liste de modifications à enregistrer

Créez un fichier textuel `hello.txt` avec pour contenu `Hello, word!`. Lancez ensuite `git status`, qui devrait vous dire que le fichier `hello.txt` est non suivi (*untracked*).

L'appel précédent de `git status` vous dit comment inclure votre nouveau fichier dans le prochain commit. Suivez son instruction afin d'ajouter `hello.txt` dans votre prochain commit. Relancez ensuite `git status`, qui devrait vous dire qu'il y a un seul changement d'enregistré pour le prochain commit : la création du fichier `hello.txt`.

Vous pouvez regarder le contenu actuel de votre commit grâce à la commande `git diff --staged`. Tout comme la commande `man git` de tout à l'heure, `git diff` peut appeler votre *pager*, qui doit être fermé pour reprendre votre travail (appuyer sur 'q' pour `less`). Le contenu de votre commit doit ressembler à ça :

```
diff --git a/hello.txt b/hello.txt
new file mode 100644
index 0000000..af5626b
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1 @@
+Hello, word!
```

Finalement, vous pouvez créer un commit à partir de ces modifications. Vous pouvez le faire en tapant `git commit -m 'initial commit'`, qui crée un com-

mit avec pour message “initial commit”. Tapez ensuite `git status` pour voir dans quel état vous êtes. Si tout s'est bien passé, `git status` devrait vous dire que votre répertoire de travail est propre (*clean*), ce qui veut dire que votre répertoire de travail correspond parfaitement au commit sur lequel vous êtes.

Modifiez maintenant le fichier `hello.txt` pour écrire `world` à la place de `word`, puis créez un commit qui contient cette modification (en réappliquant les commandes que l'on vient de voir et grâce à l'aide de `git status`).

Supprimez ensuite le fichier `hello.txt` puis créez un commit qui contient cette modification (en vous aidant là encore de `git status`). Effectuez l'appel de `git commit` sans message de commit (sans l'option `-m`), afin de rédiger le message de commit dans un éditeur de texte. Cet usage est plus adapté aux éditeurs de texte en terminal (`vim`, `emacs`, `nano`...) que graphiques. Si votre éditeur est bien configuré dans votre système (variable d'environnement `${EDITOR}` ou `${VISUAL}`), Git devrait s'en servir par défaut. Vous pouvez forcer l'éditeur pour éditer un commit grâce à la variable d'environnement `${GIT_EDITOR}` (par exemple, `export GIT_EDITOR=nano` en bash) — référez-vous à `man git commit` pour plus d'options de configuration.

3 Travailler à partir d'un dépôt existant

La commande principale pour récupérer une copie locale d'un dépôt existant est `git clone`. Tout d'abord, placez-vous à la racine du répertoire spécifique à ce TP (par exemple `${HOME}/1-git-intro`).

Vous pouvez tout d'abord faire une copie du répertoire de travail de l'exercice précédent en appelant `git clone <existing-dir> <new-dir>`, par exemple `git clone exo1 exo1-copie`, ce qui crée un nouveau répertoire `exo1-copie` avec les mêmes commits et branches que le dépôt `exo1`.

On utilise le plus souvent `git clone` pour récupérer un dépôt accessible en réseau avec la syntaxe `git clone <repo-url> [<new-dir>]`. Git supporte de nombreux protocoles pour accéder aux dépôts, les plus courants sont `ssh`, `git` et `http` — référez-vous à `man git clone` pour voir tous les protocoles supportés. Lancez la commande `git clone https://github.com/pytest-dev/pytest.git` pour récupérer le code source de `pytest`.

4 Inspecter et visualiser l'historique

Dans cette section, nous allons inspecter le contenu de commits existants et visualiser l'arbre de commits du projet `pytest`. Placez-vous donc dans le dépôt cloné dans la section Travailler à partir d'un dépôt existant.

4.1 Inspecter le contenu d'un commit

La manière la plus directe de voir un commit est d'appeler `git show`. Lancez `git show` sans paramètre supplémentaire, qui devrait vous afficher le contenu du commit courant (celui vers lequel `HEAD` pointe). Note : tout comme `man` et `git diff`, `git show` peut appeler votre *pager*.

On peut bien sûr inspecter d'autres commits que celui actuel. Lancez pour cela `git show HEAD~1`, qui devrait vous afficher le contenu du commit précédent — `HEAD~1` veut dire "le (premier) parent du commit référencé par `HEAD`" ; similairement, `HEAD~2` est "le (premier) parent du commit référencé par `HEAD~1`". En pratique, on inspecte souvent un commit en particulier (`git show <commit-id>`, par exemple `git show d18cb961cfc57`) ou le sommet d'une branche (`git show <branch>`, par exemple `git show master`). La syntaxe pour vous référer à des commits est assez riche et est décrite ici.

Une option pratique de `git show` est `--stat`, qui permet de résumer les modifications effectuées sur chaque fichier. De manière similaire, l'option `--shortstat` permet de résumer encore plus le commit. Essayez ces deux options.

4.2 Visualiser l'arbre de commits

La visualisation de l'arbre de commits se fait essentiellement grâce à la commande `git log`, qui a de nombreuses options. Lancez tout d'abord `git log` sans paramètres et observez le résultat.

Les options de `git show` peuvent être utilisées dans `git log`. Lancez `git log` avec l'option `--stat` puis avec l'option `--shortstat`. Lancez ensuite `git log` avec l'option `--patch` (ou `-p`). Ces options sont très utiles selon ce que l'on cherche dans l'historique.

Souvent, on veut représenter le lien de parenté entre les commits, ce que permet l'option `--graph`. Cette option est très souvent combinée avec `--oneline`, qui affiche en une seule ligne les informations très importantes du commit, ce qui rend l'affichage plus dense et lisible. Enfin, le format d'affichage de `git log` est complètement configurable via `--pretty` ou `--format`, et de nombreux outils (comme tig) utilisent un format d'affichage personnalisé.

Une fonctionnalité très pratique de `git log` est de pouvoir filtrer les commits de différentes manières. Tout d'abord, on peut se limiter à l'affichage de certains commits. `git log <branch>` va par exemple afficher tous les commits utilisés dans une branche. `git log <commit-id-1>...<commit-id-2>` va lui afficher uniquement les commits situés entre deux commits. On peut aussi se limiter aux

commits affectant certains fichiers ou sous-répertoires en indiquant des chemins en paramètres positionnels de `git log`. Les paramètres positionnels sont en général à la fin des commandes et peuvent être préfixes d'un `--` qui indique le début des paramètres positionnels. Par exemple, `git log -- README.rst` n'affiche que les commits qui affectent le fichier `README.rst`.

4.3 Alias de visualisation

Les formats personnalisés de `git log` pouvant être très longs, ils sont souvent utilisés dans des `alias` qui permettent à l'utilisateur d'appeler une commande longue grâce à un nom plus court. Les alias sont stockés dans la configuration de Git et peuvent être créés grâce à la commande `git config`. Par exemple, taper `git config --global alias.st status` crée un alias `st` qui permet de simplement taper `git st` au lieu de `git status`. Créez l'alias `st` et vérifiez qu'il fonctionne.

La commande `git config` permet de modifier les fichiers de configuration de Git. Vous pouvez également modifier ces fichiers à la main, ce qui est parfois plus pratique que d'interagir avec la ligne de commande. Ouvrez et observez votre fichier de configuration dans `${HOME}/.gitconfig`. Ajoutez-y ensuite un alias `l` qui appelle `git log` avec un format spécial — vous pouvez trouver un exemple de commande à appeler dans mon propre fichier de configuration de Git. Lancez ensuite votre alias pour vérifier que tout fonctionne. Notez que vous pouvez très bien appeler votre alias avec d'autres options de `git log` : `git l --stat` devrait vous afficher un résumé des modifications faites par chaque commit.

4.4 Questions de visualisation

À l'aide des exemples que nous venons de voir et du manuel, quelles commandes faut-il taper pour pouvoir répondre aux questions suivantes ?

1. Quels sont les commits qui affectent le fichier `CHANGELOG.rst` ?
2. Quelles sont les modifications du fichier `src/_pytest/fixtures.py` ?
3. Les commits qui touchent le dossier `doc` sont-ils sur beaucoup de fichiers ?
4. Quels commits ont été faits par "holger krekel" ?
   Depuis quand n'a-t-il pas commité ?
5. Quelles modifications ont été faites dans `src/` entre les versions `6.0.0` et `6.0.2` ? Indice : `man git diff`.

4.5 Outils de visualisation graphique

Il existe différentes interfaces graphiques pour visualiser un historique Git. Si vous le souhaitez, vous pouvez tenter d'utiliser une des suivantes :

- `gitk` est l'outil officiel de visualisation graphique. Il a le gros avantage d'accepter les mêmes options que `git log`, ce qui en fait un outil très puissant pour faire des analyses détaillées.

- `gitg` est un outil développé par GNOME. Il a l'avantage d'être assez intuitif aux débutants, mais n'est pas très pratique pour des usages avancés.
- `tig` est un outil interactif de visualisation d'historique Git en terminal.
- `git cola` affiche l'historique, même s'il se centre sur la création de commits.
- Les interfaces web des forges (GitHub, GitLab...) ont une visualisation sommaire de l'historique.
- Certains éditeurs de texte et IDE permettent de visualiser l'historique Git en plus de créer des commits.

5 Se déplacer dans l'historique

Il est assez courant de se déplacer dans un historique Git, notamment lorsqu'on se sert des branches (ce qui sera détaillé dans le prochain TP). Nous allons dans cette section aller voir le contenu du dépôt de `pytest` dans une ancienne version. Placez-vous donc dans le dépôt cloné dans la section Travailler à partir d'un dépôt existant.

Notons tout d'abord qu'il est possible de voir le contenu d'un fichier à une version précise sans nous déplacer dans l'historique. Par exemple, `git show 3.0.0:README.rst` affiche le contenu du fichier `README.rst` à la version `3.0.0` du dépôt. Lorsqu'on quitte l'affichage de `git show`, notre commit actuel n'a pas changé — ce qui peut se voir en appelant `git status`.

5.1 Se déplacer grâce à `git checkout`

Déplaçons-nous maintenant vers le version `3.0.0` du projet, grâce à la commande `git checkout 3.0.0`. Cette commande devrait vous indiquer que vous êtes maintenant dans un état de tête détachée (*detached HEAD*), et vous indiquer ce que cela veut dire. Elle devrait aussi vous indiquer comment revenir à un état *normal* de HEAD, c'est-à-dire sur une branche, mais nous pouvons ignorer ça pour ce TP. Appelez `git status` et observez ce qu'il vous affiche. Appelez `git l` et observez que les commits au-dessus de votre commit actuel ne sont pas affichés automatiquement (ce qui peut se corriger avec l'option `--all` de `git log`).

Affichez le contenu du fichier `README.rst` et constatez qu'il est identique à ce que vous avait affiché `git show 3.0.0:README.rst`. **Notez donc que `git checkout` modifie votre répertoire de travail pour qu'il corresponde à ce quoi vers `HEAD` pointe.**

Vous pouvez très bien créer des commits dans l'état dans lequel vous êtes. Ajoutez-vous en tant que contributeur dans le fichier `AUTHORS` et commitez cette modification. Lancez ensuite `git l` pour vérifier que votre commit est au bon endroit dans l'historique.

Revenez ensuite à votre point de départ en tapant `git checkout master`, qui remet `HEAD` vers la branche `master` **et modifie votre répertoire de travail**

en conséquence. Si vous inspectez votre historique git grâce à `git l --all`, vous verrez que votre commit n'est pas visible (votre *pager* a très probablement une fonction de recherche, elle est accessible en tapant '/' pour `less`). Votre commit n'est pas visible car il n'est utilisé par aucune des branches de votre dépôt, mais le commit est tout de même stocké par Git et va le rester (sauf si vous demandez à Git de nettoyer ces commits *perdus* en appelant son *garbage collector*).

5.2 Retrouver un commit perdu

Il peut arriver de ne pas retrouver des commits après avoir fait certaines opérations (notamment `git rebase`, comme nous verrons au prochain TP). Retrouver ces commits perdus est tout à fait possible, mais utilise des commandes un peu plus bas niveau de Git. Tout d'abord, `git reflog` permet de voir l'historique de vos manipulations Git et recense le nom des commits que vous avez créé, vous permettant ainsi de vite retrouver un commit perdu récemment.

Dans notre cas précis, `git fsck` peut aussi être très utile. Lancez `git fsck --lost-found` qui va analyser tout l'historique Git et lister les commits *perdus* (*dangling*).

Retrouvez l'identifiant du commit que vous avez créé en section Se déplacer grâce à `git checkout` et déplacez-vous de nouveau vers ce commit grâce à `git checkout <commit-id>`.

5.3 Corriger des erreurs locales

Tout comme `git checkout`, `git reset` permet de modifier votre position dans l'arbre des commits. Autrement dit, ces deux commandes permettent de changer la valeur de `HEAD`. Ces deux commandes n'ont cependant pas la même utilité, puisque `git reset` peut uniquement changer `HEAD` **sans modifier votre répertoire de travail**.

Le cas d'utilisation le plus courant de `git reset` est la réécriture d'historique pour corriger des erreurs. Imaginons que vous ayez fait une erreur dans votre commit de la section Se déplacer grâce à `git checkout`. En tapant `git reset HEAD~1` vous déplacez `HEAD` vers le commit parent mais les modifications que vous aviez effectuées sont conservées dans votre répertoire de travail. Vous pouvez le vérifier grâce à `git status`. Vous pouvez donc ensuite corriger votre modification et faire un nouveau commit.

6 Points clés à retenir de ce TP

Autour des concepts de Git.

- Ce qu'est un commit et comment en manipuler.
- Ce qu'est `HEAD` et comment le manipuler.
- Tout ce qui est commité peut être retrouvé plus tard.
  Sauf `--hard`, `--force` ou appel du *garbage collector*.

Autour des commandes de Git.

- `git status` est un très bon réflexe à avoir.
- `git init` et `git clone` pour créer des dépôts.
- `git add`, `git rm` et `git commit` pour créer des commits.
- `git show` pour voir des commits.
- `git log` pour voir l'arbre des commits et faire des recherches.
- `git diff` pour voir la différence entre versions.
- `git checkout` pour se déplacer dans l'historique.
- `git reset` pour corriger des erreurs locales.
- `man git <command>` est riche en détails.