TP `git`

Préparation

Installation de `git >= 2.23`

Nous utiliserons les commandes `restore` et `switch` introduites dans la version `2.23` de `git`, ce sont des alternatives aux commandes historiques `checkout` et `reset` qui sont surchargées et causent trop de confusions.

1. Installez `git` et vérifiez que la commande `git --version` retourne une version supérieure à `2.23`.

Installation de `tig`

De nombreuses interfaces graphiques permettent de visualiser l'historique et l'état du dépôt.

Le paquet `tig` fournit une interface curses permettant une telle visualisation dans votre terminal.

1. Installez le paquet `tig`

Configuration

Les commits contiennent le nom et l'adresse mail de la personne qui l'a crée (cette personne est appelée *committer*).

1. Pour que les commits vous identifient correctement comme étant l'auteur et/ou le committer, tapez les commandes suivantes :

   ```
   $ git config --global user.name "<Prénom> <Nom>"
   $ git config --global user.email "<adresse_mail>"
   ```

   Utilisez votre adresse mail universitaire.

2. Si votre version de `git` est au moins `2.28` et si vous voulez que la branche par défaut des dépots que vous allez créer s'appelle `main` plutôt que `master`, tapez la commande :

   ```
   $ git config --global init.defaultBranch main
   ```

3. Observez le résultat dans le fichier `~/.gitconfig`.

Solo linéaire : création du dépôt et premier commit

1. Sur votre machine locale, dans votre répertoire de travail d'administration système, créez un répertoire nommé `git-test/` et initialisez-y un dépôt git.
2. Dans ce répertoire, créez un fichier nommé `plop` dont le contenu est la chaîne `hop`.
3. Ajoutez le fichier `plop` à la staging area (index).
4. Créez un commit dont le commentaire est "mon premier commit".

Solo linéaire : deuxième commit et observation

1. Modifiez le fichier `plop` en remplaçant la chaîne de caractères `hop` par `hap`.

2. Observez ce que retourne chaque commande du slide "Informations".
3. Ajoutez le fichier `plop` à la staging area (index).
4. Observez ce que retourne chaque commande du slide "Informations".
5. Créez un fichier `toto` dont le contenu contient 4 lignes de votre choix.
6. Ajoutez le fichier `toto` à la staging area (index).
7. Observez ce que retourne chaque commande du slide "Informations".
8. Modifiez la 2e ligne du fichier `toto`.
9. Observez ce que retourne chaque commande du slide "Informations".
10. Ajoutez le fichier `toto` à la staging area (index).
11. Observez ce que retourne chaque commande du slide "Informations".
12. Commitez vos changements.
13. Observez ce que retourne chaque commande du slide "Informations".

Solo linéaire : troisième commit et observation

1. Modifiez la 2e ligne du fichier `toto`.
2. Ajoutez le fichier `toto` à la staging area (index).
3. Observez ce que retourne chaque commande du slide "Informations".
4. Commitez vos changements.
5. Observez ce que retourne chaque commande du slide "Informations".
6. Baladez-vous dans vos 3 commits en utilisant `tig` (cliquez sur entrée pour voir les diffs).

Solo linéaire : oups

1. Éditez le fichier `toto`, supprimez sa première ligne et sauvegardez-le.
2. C'est une boulette, récupérez le contenu de la version précédente du fichier `toto` grâce à `git`.
3. Éditez le fichier `toto`, modifiez sa 4e ligne et sauvegadez-le.
4. Ajoutez le fichier `toto` à la staging area.
5. C'est encore une boulette, nettoyez la staging area ainsi que votre répertoire de travail de sorte à ce que le fichier `toto` soit identique à celui du 3e commit.

Solo programmation

Cette partie n'est pas prioritaire (vous pouvez passer au prochain paragraphe et y revenir plus tard).

1. Lisez la page du slide intitulée "Ne versionner que le source, ignorer les sous-produits".
2. Créez un répertoire nommé `programmation/` dans votre répertoire de travail.
3. Créez-y un fichier de type "Hello world" nommé `hello.c`, et commitez-le.
4. Compilez-le et exécutez-le.
5. Que donne la commande

   ```
   $ git status
   ```
6. Créez un fichier `.gitgnore` à la racine de votre répertoire de travail (worktree) de sorte à éviter de commiter par inadvertance le fichier compilé.
7. Que donne la commande

   ```
   $ git status
   ```
8. Si les fichier compilés n'apparaissent plus, commitez le fichier `.gitgnore`.
9. Modifiez le fichier `hello.c` de sorte à ce qu'il fasse plutôt "Hello git".
10. Compilez-le et exécutez-le.

11. Commitez-le.

Solo branches

1. Créez une branche nommée `essai`
2. Allez sur cette branche (*i.e.* faites de `essai` votre branche courante).
3. Créez un fichier `test.txt`, ajoutez-le à la staging area et commitez.
4. Créez un tag nommé `milieu-test`
5. Modifiez le fichier `test.txt`, ajoutez-le à la staging area et commitez.
6. Regardez le contenu du fichier `.git/HEAD` et des différents fichiers et sous-répertoires du répertoire `.git/refs/`
7. Essayez de comprendre comment `git` gère ses références (tags, branches, branche courante).
8. Retournez sur la branche `main`.
9. modifiez le fichier `plop`, ajoutez-le à la staging area et commitez.
10. Mergez la branche `essai` dans la branche courante `main`.
11. Observez le résultat avec `tig`.

Solo branches dessine moi un DAG

À faire avec un·e camarade (vous pouvez passer au prochain paragraphe et y revenir plus tard).

1. Demandez à votre camarade de dessiner sur une feuille un DAG un peu complexe avec des tags et des branches à plusieurs endroits.
2. Amusez-vous à réaliser ce DAG comme le DAG des commits de votre dépôt avec les commandes `add, commit, branch, switch, tag, merge`.

   Si vous ne voulez pas perdre de temps à créer du contenu à commiter mais vous concentrer sur la gestion du DAG, vous pouvez au choix (ou successivement) :

   - dessiner directement votre DAG dans le simulateur learngitbranching
   - utiliser les alias suivants (le premier crée un fichier vide avec un nom aléatoire, l'ajoute dans la staging area et le commite avec un message aléatoire, le second dessine le DAG des commits en ASCII-art) :

     ```
     $ alias Commit_un_truc='R=$RANDOM;touch f$R;git add f$R;git commit -m c$R'
     $ alias DAG='git log --graph --oneline --all --decorate --color'
     ```

     Ainsi, il suffit de taper la commande `Commit_un_truc` (utilisez l'autocomplétion !) pour créer un nouveau commit et `DAG` pour observer le DAG obtenu.
3. Demandez à votre camarade de choisir une branche et un de ses ancêtres lointains.
4. Faites de l'ancêtre choisi le commit courant sans utiliser son hash, mais avec l'adressage relatif à la branche choisie (cf slides).
5. Cherchez ensemble des commits qui possèdent des ancêtres joignables par plusieurs chemins orientés dans le DAG. Faites un `git diff` entre ces deux adresses du même commit pour vérifier qu'il n'y a pas de différence (et valider votre hypothèse sur les chemins).
6. Imaginez d'autres challenges de ce type de sorte à devenir à l'aise dans la navigation dans le DAG des commits. Amusez-vous. Si vous découvrez un exercice instructif, n'hésitez pas à le partager sur le salon `sysadmin_git` du mattermost.
7. Inversez les rôles.

Collectif : constituer un groupe et cloner un dépôt commun

1. Lorsque vous avez fini avec les exercices précédents, constituez un groupe d'au moins 5 étudiant·es pour travailler ensemble sur un dépôt distant commun, par exemple votre groupe de projet. Vous pouvez utiliser le salon mattermost `sysadmin_git` pour chercher des camarades.
2. Lorsque votre groupe est constitué, choisissez un nom de dépôt distant dans la liste suivante, et reservez-le en l'annonçant sur le salon `sysadmin_git` pour éviter les collisions : `aqua`, `aquamarine`, `azure`, `beige`, `bisque`, `black`, `blue`, `brown`, `chartreuse`, `chocolate`, `coral`, `cornsilk`, `crimson`, `cyan`, `fuchsia`, `gainsboro`, `gold`, `goldenrod`, `gray`, `green`, `honeydew`, `indigo`, `ivory`, `khaki`, `lavender`, `lime`, `linen`, `magenta`, `maroon`, `moccasin`, `navy`, `olive`, `orange`, `orchid`, `peru`, `pink`, `plum`, `purple`, `red`, `salmon`, `seashell`, `sienna`, `silver`, `snow`, `tan`, `teal`, `thistle`, `tomato`, `turquoise`, `violet`, `wheat`, `white`, `yellow`.
3. Voici quelques informations sur la machine hébergeant le dépôt distant. Lorsque nous serons plus avancé·es dans le cours de sysadmin, toutes ces informations devraient vous être familières et compréhensibles.
   - la machine distante est un conteneur accessible via le nom de domaine `yologit.netlib.re`. Il possède sa propre adresse IPv6. Comme vous ne savez pas encore comment joindre une machine IPv6-only par SSH depuis un réseau IPv4-only, ce conteneur hérite de l'adresse IPv4 de la machine hôte qui l'héberge, grâce à une redirection de port. Ainsi, le port de connexion SSH ne sera pas le port 22 (défaut), mais `2271`.
   - les fingerprints des clefs publiques du serveur SSH de ce conteneur sont :

     ```
     RSA     : SHA256:uwcsZ4oiL19jikWYFA2S6u3DhF/NUSnnHRP1wJjeBzk
     ECDSA   : SHA256:bAQxBoag6nOvmmt657sMzP45ktbe/lTKnVsHSNAJnyE
     ED25519 : SHA256:0eoJohNR3VCQBP7OxHORYdQ8720cXobNJNTr32FocbA
     ```

   - l'user qui héberge les dépôts distants sur ce conteneur s'appelle `gituser`.
   - le shell de l'user `gituser` a été configuré pour n'accepter que des commandes `git`. En effet, le fichier `/etc/passwd` de la machine `yologit.netlib.Re` contient la ligne :

     `gituser:x:1000:1000:,,,:/home/gituser:/usr/bin/git-shell`

   - comme vous n'avez pas encore manipulé de clefs SSH, l'authentification se fera exceptionnellement par mot de passe, celui-ci se trouve dans l'en-tête du salon mattermost `sysadmin_git`, il est n'est pas public.
4. Retournez dans votre répertoire de travail d'administration système (quittez `git-test/`) et clonez le dépôt que vous avez choisi :

   `$ git clone ssh://gituser@yologit.netlib.re:2271/~/<nom_du_depot_distant>`
5. Un message d'alerte s'affiche. Afin d'être sur·e que la connexion n'a pas été interceptée par une entité malveillante, vérifiez que le fingerprint du serveur SSH de la machine distante correspond bien à l'un des fingerprints cités plus haut (point 3).
6. Une fois que vous avez vérifié le fingerprint, vous pouvez taper `yes`, et `git` clonera le dépôt via SSH.

Collectif : travailler en parallèle sur des fichiers disjoints

1. Dans le répertoire de travail de votre nouveau dépôt local se trouve un répertoire nommé `perso/`. Créez-y un fichier dont le nom est votre `prenom.nom` et ajoutez-y des informations (non-compromettantes) sur vous (n'hésitez pas à baratiner, pensez que la confidentialité de ce dépôt est très faible).
2. Commitez ces changements et poussez-les sur le dépôt distant.
3. Recommencez l'opération en rajoutant des détails dans le fichier `perso/<prenom.nom>` et repoussez votre nouveau commit.

4. Recommencez jusqu'à devoir merger les changements effectuées par un·e autre étudiant·e de votre groupe.
5. Continuez jusqu'à ce que tout le monde ait pu faire des `pull` des `push` et des `merge`.
6. Observez l'historique et le DAG des commits.

Collectif : travailler en parallèle sur des parties distinctes d'un même fichier, avec un·e release manager

Le but de ce paragraphe est de jouer au jeu du *cadavre exquis* inventé par les surréalistes, mais avec `git`.

1. Désignez 5 joueu·ses et un·e maitre·sse du jeu ("release manager"). Dans tout le jeu, le ou la release manager est la seule qui peut modifier la branche `main`.
2. Dans le répertoire de travail de votre dépôt local partagé se trouve un répertoire nommé `cadavre_exquis/`. Dans ce répertoire se trouve un fichier nommé `template.txt`.
3. Le ou la release manager copie le template en un fichier `jeu.1.txt`, commite ce fichier dans la branche `main` et pousse cette branche sur le dépôt commun que tout·es les joueu·ses récupèrent.
4. Répartissez les 5 lignes à remplir parmis les joueu·ses (1. substantif, 2. adjectif, 3. verbe, ...).
5. Une fois que vous êtes d'accord sur la ligne que chaque joueu·se doit remplir, créez chacun·e une branche ayant pour nom votre prénom.
6. Chaque joueu·se modifie en secret le fichier `jeu.1.txt` en ajoutant au niveau du tiret qui correspond à sa ligne un ensemble de mots admissibles (par exemple un verbe pour la 3e ligne). Évitez les termes offensants ou discriminatoires.
7. Chaque joueu·se commite ses changements sur sa branche et pousse sa branche sur le dépôt commun.
8. Le ou la release manager récupère toutes les branches sur son dépôt local, les merge dans la branche `main` et pousse cette branche sur le dépôt commun. Comme les lignes modifiées sont différentes, il ne devrait pas y avoir de collision à gérer.
9. Tout le monde récupère la branche `main` du dépôt commun et découvre le résultat du cadavre exquis.
10. Observez qui a effectué les changements sur le fichier avec la commande :

    ```
    $ git blame jeu.1.txt
    ```

11. Recommencez une partie (en remplaçant `jeu.1.txt` par `jeu.2.txt`, etc) de sorte à ce que tout le monde ait pu jouer le rôle de release manager.

Collectif : travailler en parallèle sur des parties communes d'un même fichier en étant sur une même branche

1. Dans le répertoire de travail de votre nouveau dépôt local, créez un fichier nommé `baston`, commencez à y écrire un texte, commitez-le et poussez vos changements sur le dépôt distant.
2. Recommencez l'opération en rajoutant des détails à votre texte dans le fichier `baston` et repoussez votre nouveau commit.
3. Modifiez des lignes existantes pour ajouter des précisions.
4. Recommencez jusqu'à devoir intégrer les changements effectués par un·e autre étudiant·e de votre groupe.
5. Essayez de maintenir la cohérence du texte (le sens peut évoluer mais il doit toujours avoir du sens et être gramaticalement correct) sans discuter entre vous.
6. Si un `merge` doit être résolu manuellement, décidez seul·e de la version à garder pour que le texte reste cohérent.

7. Continuez jusqu'à ce que tout le monde ait pu faire des `pull` des `push` et des `merge` en rapport avec le fichier `baston`.
8. C'est facile ou on s'organise ensemble ?

Collectif : choisissez votre workflow

Si vous êtes arrivé·e à ce paragraphe dans les 3h, félicitations !

1. Discutez avec votre équipe, et répartissez-vous des tâches avec des fichiers à modifier, des noms de branches, et éventuellement des rôles pour savoir qui merge quoi (ou pas).
2. Essayez de les réaliser et amusez-vous.
3. Gardez en tête que git est un outil flexible qui n'impose pas d'organisation collective particulière, que le meilleur workflow est celui que vous déciderez ensemble.