TP – Introduction à git

Ce TP est à faire en binôme (ou trinôme ou plus). Les différents utilisateurs·trices seront désignés par *U1* et *U2* dans la suite du sujet de TP.

Objectifs

Ce TP vise à vous faire prendre en main les commandes de base pour utiliser un dépôt `git`. L'outil `git` est un puissant outil de gestion de versions que vous utiliserez pour le projet.

Préparation

Connectez vous sur le serveur GitLab de GRICAD avec vos identifiants étudiants. L'URL du serveur est https://gricad-gitlab.univ-grenoble-alpes.fr/.

Indiquez à votre enseignant votre identifiant pour qu'il puisse vous attribuer un dépôt avec lequel travailler.

Avez-vous indiqué votre idenfiant ? Si oui, visionnez la vidéo https://youtu.be/iub0_uVWGmg.

Commandes de base

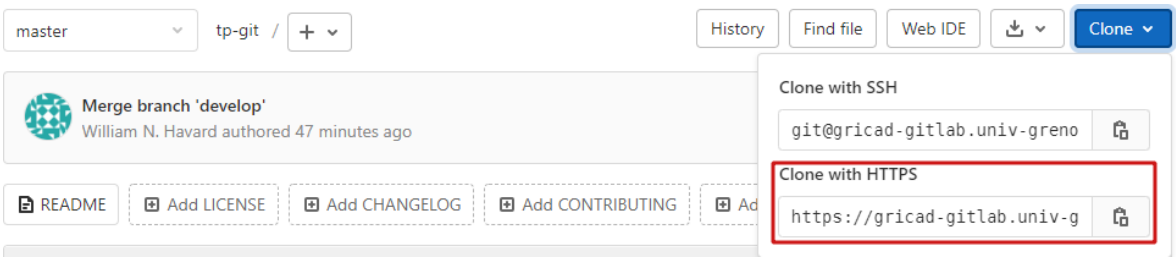
Étape 1 (*U1* & *U2*)**.** Configurez `git` sur votre machine à l'aide des commandes suivantes :

```
$ git config --global --add user.name 'Prénom NOM'
$ git config --global --add user.email 'prenom.nom@etu.univ-grenoble-alpes.fr'
```

Étape 2 (*U1* & *U2*)**.** Clonez le dépôt GitLab de votre groupe sur chacune de vos machines locales à l'aide de la commande suivante :

```
$ git clone https://lien-https
```

Vous trouverez le lien à utiliser pour cloner le dépot GitLab en cliquant sur le bouton « Clone » : utilisez le lien https (`https://...`).



Quel a été l'effet de la commande ?

Étape 3 (*U1*)**.**
- Déplacez-vous dans le dossier créé par la commande `$ git clone`.
- Ajoutez à la ligne 2 du fichier `README.md` la phrase « `U1 fait une modification` ».
- Enregistrez le fichier.
- Qu'observez-vous en saisissant la commande `$ git status` ?

Explication

Dans `git`, les fichiers qui sont versionnés peuvent être dans deux états lorqu'ils ont été modifiés : *staged* ou *unstaged*. Seules les modifications des fichiers dans l'état *staged* seront enregistrées dans le prochain `commit`.

On peut sélectionner finement quelles modifications apparaitront dans la prochaine version. Par exemple, lorsqu'on ajoute une nouvelle fonctionnalité, il est rare de ne modifier qu'un seul fichier. Plutôt que de créer une version pour chaque fichier, on crée une version pour tous les fichiers concernés par la nouvelle fonctionnalité. Dans ce cas, on passe tous les fichiers concernés en mode *staged*. Ainsi, en créant le prochain `commit`, on aura une version avant l'ajout de la fonctionnalité et une version après l'ajout de la fonctionnalité.

La commande `$ git diff` permet de voir les modifications entre la dernière version enregistrée et la version actuelle. Les ajouts sont signalés par un « `+` » et les suppression avec un « `-` ».

L'étape suivante indique comment changer le mode d'un fichier vers le mode *staged*.

Étape 4 (*U1*)**.** Ajoutez le fichier `README.md` à la liste des fichiers qui seront ajoutés à la prochaine version grâce à la commande suivante :

```
$ git add README.md
```

Que renvoie la commande `$ git status` maintenant ?

Explication

Vous avez changé le mode du fichier `README.md`. Les modifications que vous venez de faire seront enregistrées dans le prochain `commit`.

La commande `$ git diff --staged` vous permet de voir les modifications qui seront enregistrées.

Maintenant que les modifications sont marquées, il ne reste plus qu'à créer une version.

Étape 5 (*U1*)**.** Enregistrez une nouvelle version à l'aide de la commande suivante :

```
$ git commit --message="U1: Ajout d'une ligne"
```

Explication

Vous venez de créer une nouvelle version. Vos modifications sont maintenant enregistrées de manière perenne, et il est possible de revenir à tout moment à cet état de vos fichiers. Remarquez que la commande `$ git status` ne montre plus le fichier `README.md`.

L'option `--message` peut être omise. En ce cas, `git` utilisera l'éditeur de texte configuré (ou à défaut `vi`).

Une version (`commit`) est identifié par une empreinte numérique (*hash*) de type

SHA-1. En général les 6 premiers caractères sont suffisants pour identifier une version. En cas de collision (rare), l'empreinte entière est utilisée.

Toutes les version enregistrées jusqu'à présent ne sont connues que sur la machine de *U1*.

Étape 6 (*U1*)**.** Envoyez vos versions enregistrées sur le dépot GitLab avec la commande suivante :

```
$ git push
```

Que s'est-il passé ? Observez le dépôt GitLab (si nécessaire, rafraichissez la page).

Explication

Vous venez de publier vos versions (historique) sur le serveur GitLab. Tout le monde peut maintenant voir les modifications que vous avez fait. Les autres utilisateurs·trices peuvent amener de nouvelles modifications en repartant de votre version publiée.

Étape 7 (*U2* et autres)**.** Reprenez les Étapes 3 à 6 en utilisant votre nom d'utilisateur. Que se passe-t-il lorsque vous essayez d'envoyer vos modifications au serveur GitLab ? Pourquoi ?

Explication

Vous essayez d'envoyer vos modifications sans connaître les nouvelles versions qui existent sur le serveur GitLab. `git` vous empêche de synchroniser vos modifications sans prendre connaissance de celles qui ont été ajoutées depuis votre dernière synchronisation. Il vous faut donc rappatrier les nouvelles modifications, vous synchroniser avec, puis envoyer le tout au serveur.

Étape 8 (*U2* et autres)**.** Rappatriez l'historique des versions depuis le serveur GitLab avec la commande `$ git pull`. Que se passe-t-il ? Ouvrez le fichier `README.md` et examinez son contenu. Comment est signalé le conflit ?

Explication

`git` ne sait pas comment synchroniser les modifications apportées par *U1* et celle apportées par *U2* : elles concernent la même portion de fichier. `git` vous montre les modifications qu'il ne sait pas gérer de la manière suivante :

```
<<<<<<< HEAD
modifications locales
=======
modifications obtenues du serveur GitLab
>>>>>>> (SHA-1 de la version du serveur)
```

C'est à vous de choisir ce qu'il faut conserver. Vous pouvez tout garder, ne conserver que vos modifications, un mélange des deux, … Une fois le conflit résolu, il faut enlever les lignes marqueurs de conflit (`<<<<<<<`, `=======` et `>>>>>>>`).

Étape 9 (*U2*)**.** Résolvez le conflit en gardant les deux lignes, puis en supprimant les lignes marqueurs (`<<<<<<<`, `=======` et `>>>>>>>`). Indiquez à `git` que le conflit est résolu à l'aide de la séquence de commandes suivante :

```
$ git add README.md
$ git commit
$ git push
```

Étape 10 (*U1*)**.** Rappatriez les modifications depuis le serveur GitLab. Que constatez-vous ?

Étape 11. Reprenez les Étapes 3 à 10 en inversant les rôles de *U1* et *U2*.

Étape 12 (*U1* & *U2*)**.**

U1 Ajoutez à la ligne 3 « `U1 ajoute une nouvelle ligne` ». Créez une nouvelle version avec le message « *U1* : nouvel ajout » puis synchronisez vos modifications avec le serveur GitLab.

U2 Rappatriez les modifications depuis le serveur GitLab. Le `pull` s'est-il déroulé correctement, pourquoi ?

Explication

Ici, *U2* n'a pas ajouté de version locale inconnue du serveur. *U2* peut donc rappatrier les nouvelles versions sans conflit.

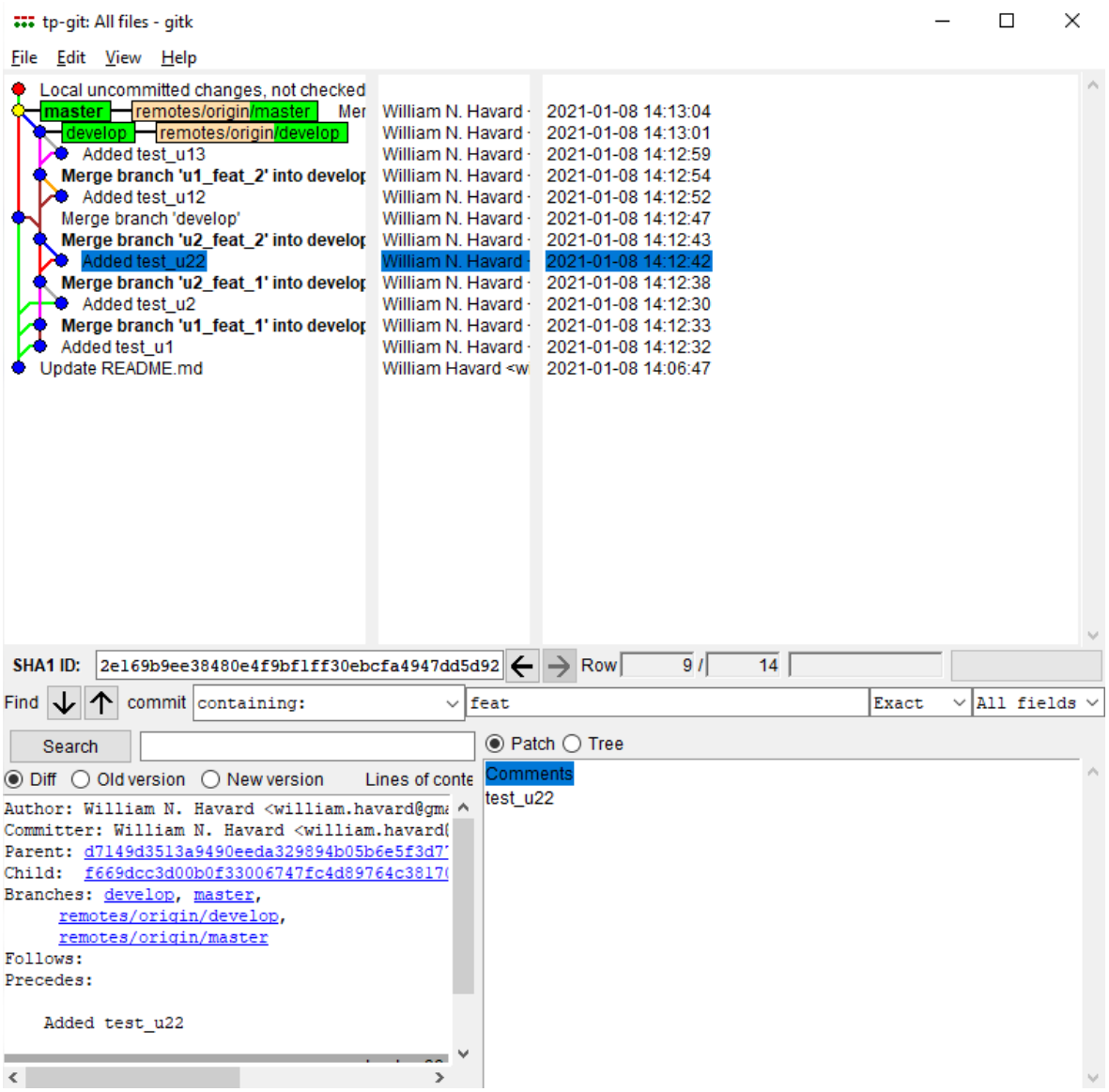
Bonnes pratiques

- Faire des `commit`s régulièrement pour vos modifications locales. Attention, on ne crée pas une nouvelle version à chaque ligne de code : il faut qu'une version ait un sens du point de vue de l'application. Cela peut être une nouvelle fonction, classe. Cela peut aussi être une correction de bug (en ce cas, il se peut que la modification n'affecte qu'une unique ligne).
- Utiliser des messages de `commit` explicite. Un message clair qui identifie ce que fait une version permet de trouver les problèmes plus rapidement. Par exemple : « Ajout de la fonctionnalité X », « Correction du bug Y », « Renommage de la variable Z ».
- Faire des `pull`s régulièrement. Cela permet d'intégrer rapidement les modifications. Il est plus simple de résoudre plusieurs petits conflits qu'un gigantesque conflit.
- Faire des `push`s avec parcimonie. Dans l'idéal, on envoie des versions fonctionnelles et « sans bogue ».

`[facultatif]` Gestion (simple) de l'historique

Les différentes versions créées avec la commande `$ git commit` conservent un lien vers la (ou les) version(s) précédente(s) : on parle alors d'historique.

Il existe plusieurs outils pour visualiser l'historique d'un dépôt `git`. Dans un terminal, vous pouvez utiliser la commande `$ git log`. Sur la page web de votre dépôt GitLab, l'historique est visisible dans le menu `Repository > Graph`. Vous pouvez aussi visualiser l'historique depuis votre machine avec l'outil `gitk`. Pour ce faire, saisissez dans un terminal la commande `$ gitk --all` depuis le répertoire où se situe votre dépôt `git`.



Étape 13 (*U2*)**.** *U2* se rend compte que la dernière version ajoutée par *U1* n'est pas pertinente. On décide d'annuler cette version. Utilisez `gitk` ou la commande `$ git log` pour trouver l'identifiant (SHA-1) du dernier `commit`. Annulez la dernière version avec la commande `$ git revert {sha-1}`.[1] Comment est modifié l'historique ? Pourquoi ?

1. On peut aussi utiliser `HEAD` pour identifier la dernière version, en ce cas la commande devient `$ git revert HEAD`.

`[facultatif]` Utilisation des branches

L'objet de cette section est de prendre en main la notion de branche. Cette notion est centrale pour utiliser `git` de manière efficace.

Une branche représente une portion d'historique indépendante. Par défaut, `git` crée une unique branche nommée « *master* » : c'est cette branche que vous avez utilisé jusqu'à présent. En travaillant à plusieurs sur un projet, on est ammené à développer simultanément plusieurs fonctionnalités indépendantes. En utilisant plusieurs portions d'historique différentes (branches) il est possible d'échanger des modifications sans avoir d'impact sur tout l'historique. Cela a plusieurs avantages : les modifications liées à une fonctionnalité particulière sont regroupées, l'historique est plus lisible, etc.

Étape 14 (*U1*)**.** Créez une nouvelle branche `feature/u1` puis envoyez là au serveur GitLab avec la suite de commandes suivante :

```
$ git checkout -b feature/u1
$ git push --set-upstream origin feature/u1
```

Étape 15 (*U2*)**.** Rappatriez la nouvelle branche depuis le serveur GitLab avec les commandes suivantes :

```
$ git fetch
$ git checkout --track origin/feature/u1
```

Étape 16 (*U1* & *U2*)**.** Créez un nouveau fichier, modifiez le et ajoutez plusieurs versions sur la nouvelle branche. À l'aide de `gitk` observez où vos `commit`s sont enregistrés. Qu'observez-vous ?

Étape 17 (*U1*)**.** À l'aide de la commande `$ git checkout master`, déplacez vous sur la branche *master*. Fusionnez les deux branches à l'aide de la commande :

```
$ git merge --no-ff feature/u1
```

Observez à l'aide de `gitk` comment l'historique est modifié.

Étape 18. Reprenez les Étapes 14 à 17 en inversant les rôles de *U1* et *U2*. Vous créerez une nouvelle branche `feature/u2`.

Références / Aller plus loin

Références

manuel `git` `man 1 git`, `man 1 git commit`, `man 1 git add`, ...

documentation `git` https://git-scm.com/doc

pense bête graphique https://ndpsoftware.com/git-cheatsheet.html

Aller plus loin

navigation avancée https://learngitbranching.js.org/

***workflow* `git`**

- https://docs.gitlab.com/ee/topics/gitlab_flow.html
- https://nvie.com/posts/a-successful-git-branching-model/