Lic. Info. 2ème année — Université d'Orléans
SLA3IF06 - Outils du développeur — Année 2023-2024

TP n°6 - Gestion de versions - Git

1. Création d'un dépôt local

Dans cette section nous allons créer un dépôt ex nihilo et apprendre à utiliser les commandes Git de base. Créez un répertoire vide `enerGit`. Rendez-vous dans ce répertoire et faites-en un dépôt Git en lançant

```
git init
```

Remarquez avec `ls -a` qu'un répertoire `.git` est apparu.

Copiez tous les fichiers du dossier `data/haddock` dans ce répertoire. Lancez `git status` et observez le rapport fourni par Git. Les fichiers ne sont pour l'instant pas suivis (*untracked*) par Git. Pour les ajouter, lancez

```
git add *
```

Lancez de nouveau `git status` pour observer le changement du rapport. Les fichiers sont maintenant suivis (*tracked*) par Git et ont été ajoutés à la liste des modifications à prendre en compte (*staged*) lors du prochain enregistrement, mais aucun enregistrement (commit) n'a encore été fait. Pour faire votre premier enregistrement :

```
git commit -m "Premier commit"
```

Remarque : Git peut refuser votre commit et vous demander au préalable de configurer votre nom d'utilisateur et/ou adresse mail... Ajoutons au passage la prise en charge de la coloration.

```
git config --global user.name "Jeannot Lapin"
git config --global user.email "jeannot.lapin@basse-cour.fr"
git config --global --add color.ui true
```

Il faut voir Git comme un mini-système de fichiers dont chaque commit contient le cliché pris à un instant donné des fichiers du système.

Modifiez le contenu d'un des fichiers et lancez de nouveau `git status`. Le fichier étant suivi (*tracked*) par Git, il est reconnu comme modifié dans le rapport, mais ses modifications ne seront pas prises en compte lors du prochain enregistrement ! Pour les prendre en compte, il faut de nouveau ajouter le fichier avec `git add` (afin de faire passer le fichier dans l'état *staged*) puis valider avec `git commit`.

Modifiez de nouveau un fichier suivi, ajoutez-le avec `git add,` puis modifiez-le de nouveau. Lancez `git status`, comment interpréter ce rapport ? En l'état, quelle version du fichier sera enregistrée lors du prochain `git commit` ?

Essayez maintenant de retirer un fichier (par exemple `noms`) du suivi de Git avec chacune des trois commandes suivantes (mais **n'en utilisez qu'une seule à la fois**) :

```
git rm noms
rm noms
git rm --cached noms
```

Identifiez les différences (pensez à `ls` et `git status`). Pour restaurer le fichier, vous pouvez dans tous les cas faire :

```
git checkout HEAD -- noms
```

`HEAD` désigne toujours la révision courante du dépôt. Ajoutez des fichiers et/ou effectuez des modifications et enregistrez-les avec `git commit`. Répétez cela 3-4 fois. Le dernier commit peut être identifié par `HEAD`, le précédent par `HEAD^`, celui d'avant par `HEAD^^` et on peut remonter au nième parent avec `HEAD~n`. Mais d'une manière générale, les commits sont identifiés par leur hash `SHA1`.

Lancez `git log` pour afficher la liste des commits. Visualisez les différences entre les différents commits à l'aide de :

```
git diff <commit1> <commit2>
```

où `<commit1>` et `<commit2>` désignent soit des identifiants complets (hash `SHA1` de 40 chiffres hexadécimaux) de commits, soit des préfixes de ces identifiants (généralement les 5 à 7 premiers caractères sont suffisants pour identifier de façon unique un commit, même sur de très gros projets), soit des raccourcis des formes évoquées précédemment (`HEAD, HEAD^`, etc), soit des tags de commit (on y reviendra), etc...

2. Restauration de fichiers

Un dépôt Git travaille simultanément avec plusieurs versions d'un même fichier suivi :

- les versions dans l'historique (la dernière datant du dernier commit) ;
- la version *staged* (datant du dernier `add`), en attente de commit ;
- la version de travail courante (celle dont l'utilisateur dispose dans son répertoire de travail).

Dans le répertoire `enerGit`, modifiez le fichier `noms` puis testez la commande suivante. Le fichier `noms` est de quelle version ?

```
git checkout HEAD -- noms
```

Modifiez de nouveau le fichier `noms` puis testez la commande suivante. Le fichier `noms` est de quelle version ?

```
git checkout noms
```

Idem pour la commande suivante. Identifiez le rôle de chacune de ces commandes.

```
git reset noms
```

3. Échanges avec dépôt distant

Faire d'abord le paramétrage suivant si vous travaillez dans les salles de l'université (A NE PAS FAIRE quand on fait le TP chez soi) :

```
export http_proxy="wwwcache.univ-orleans.fr:3128"
export https_proxy=$http_proxy
```

Nous allons apprendre les mécanismes de Git permettant d'importer/exporter des modifications en dehors du dépôt (ou clone) de travail. Lorsque plusieurs utilisateurs souhaitent travailler sur un projet commun via Git, ils disposent d'outils permettant de propager des commits locaux vers l'extérieur (opération `push`) ou d'importer des commits distants vers leur dépôt local (opération `pull`).

Vous disposez d'un dépôt auquel vous pouvez accéder avec un navigateur web à l'adresse suivante :

```
https ://pdicost.univ-orleans.fr/git/projects/L2OD/
```

Vous vous logguez avec les paramètres suivants :

```
- login : o+numéro étudiant (par ex. o3442132)
- mot de passe : votre numéro NNE (que vous trouverez dans l'ENT)
```

Vous devez maintenant voir un dossier "ODxxx" (xxx étant un numéro) : cliquez sur ce dossier. Voilà, vous êtes dans votre dépôt...

Sur la page d'accueil vous trouverez la commande git à taper pour faire un clone du dépôt sur votre machine dans la rubrique "Je veux juste cloner ce dépôt". Sur votre machine, créer un répertoire `clone1`, descendre dedans et lancer le clonage :

```
mkdir clone1
cd clone1
git clone https://... # à remplacer par la commande trouvée sur votre dépôt distant
```

Vous constaterez qu'il n'y a rien dans `clone1` à l'intérieur, à part le répertoire caché `.git`.

Créez un fichier `Hello.txt` dans votre clone, avec le contenu "Bonjour" (ou autre chose si vous préférez. c'est juste un exemple).

Faites un `add` de ce fichier puis un `commit`. Enfin faites un `push`, c'est-à-dire une synchro de votre clone vers le dépôt distant (en indiquant explicitement la branche `master` car le dépôt distant, correspondant au `remote origin`, est pour l'instant vide) :

```
git push origin master
```

Le dépôt distant n'est plus vide. Vous pouvez consulter son contenu grâce à l'interface web.

Vous pouvez ensuite créer une seconde copie de votre dépôt sur votre machine (en faisant un nouveau `git clone` dans un autre répertoire ; attention à ne pas placer cette nouvelle copie à l'intérieur du répertoire de la première copie ! ça ne fonctionnerait pas...)

```
cd ..
mkdir clone2
cd clone2
git clone https://... # a remplacer par la commande trouvée sur votre
dépôt distant
```

Puis faire des modifications dans cette seconde copie (eg modifier le fichier `Hello.txt` et faire son `add` dans l'index), commiter, pusher, vérifier que le `push` a bien été fait sur le dépôt.

Afin de récupérer les modifications distantes (synchro du distant vers votre clone), on utilise la commande `git pull`.

Repasser dans la première copie, faire un `pull`, vérifier que vous avez bien le dernier commit.

```
cd ../clone1
git pull
ls -l
```

Remarque : pour le clone on aurait pu procéder en une seule étape : demander au `git clone` de créer le répertoire contenant le clone. La commande est alors :

```
git clone https://... mon_repertoire
```

En pensant bien sûr à remplacer "https ://..." et "mon_repertoire" par les valeurs souhaitées.

4. Échanges avec dépôt local

Nous allons maintenant fabriquer notre propre dépôt GIT "partagé" et approfondir le paramétrage de l'accès aux dépôts.

Une bonne pratique consiste à partager un *dépôt nu (bare)* dans lequel on ne peut pas directement travailler (il n'y a pas de copie de travail des fichiers suivis, seulement la machinerie interne de Git et l'historique du projet), mais sur lequel tous les utilisateurs peuvent faire des `pull` et `push`.

Rendez-vous dans le répertoire du TP6. Commençons par créer deux clones locaux d'un dépôt distant, l'un *bare*, l'autre pas :

```
git clone --bare https://gist.github.com/6883859.git depot-bare
git clone https://gist.github.com/6883859.git depot-travail
```

Vous remarquerez que le dépôt `depot-bare` ne contient pas de copie de travail des fichiers, il contient directement (à peu de choses près) le contenu du dossier `.git` du `depot-travail`.

Les dépôts distants enregistrés dans un dépôt sont appelés *remotes*. Consultons les remotes du dépôt de travail :

```
cd depot-travail
git remote -v
```

Pour l'instant, vous devez voir un seul remote nommé `origin` qui correspond à l'URL du dépôt cloné. Vous ne disposez pas des droits pour effectuer des `push` sur ce dossier. À la place, vous allez déclarer le dépôt nu `depot-bare` comme nouveau remote sous le nom `partage`.

```
git remote add partage file:///home/mon_login/depot-bare
git remote -v
```

Git ne fait aucune vérification lors de l'ajout d'un remote, ainsi si vous vous êtes trompé d'adresse, il ne vous le dira pas immédiatement. Afin de s'assurer qu'il n'y a pas d'erreur, inspectons les remotes déclarés :

```
git remote show origin
git remote show partage
```

Si la seconde commande signale une erreur concernant le remote `partage`, vous pouvez le supprimer avec `git remote rm partage` et le recréer correctement.

Afin de récupérer les modifications distantes, on utilise la commande `git pull`. Cependant, le nouveau `remote partage` n'est pas déclaré comme remote par défaut, c'est donc le `remote origin` qui va être consulté lors d'un `git pull` (faire `git pull -v` pour le constater). Pour faire un pull sur partage, on doit spécifier explicitement le nom du remote et la branche (ici il n'y a qu'une seule branche `master)` :

```
git pull partage master
```

On peut aussi déclarer le remote partage comme remote par défaut pour la branche courante (master), ainsi `pull` et `push` seront moins lourds à utiliser :

```
git config branch.master.remote partage
git pull -v
```

À présent, créez un nouveau clone de travail, nommé par exemple `depot-travail-bis`, du dépôt `depot-bare` puis modifiez l'un des fichiers de `depot-travail` et faites un commit des modifications. Propagez ce commit au remote partage :

```
git push partage
```

Mentionner partage n'est pas indispensable si vous l'avez déclaré comme remote par défaut pour la branche courante `master`.

Rendez-vous dans depot-travail-bis et récupérez les modifications avec `git pull`. Refaites de même depuis ce dépôt modifiez un fichier, `commit, push,` puis `pull` dans `depot-travail`.

Vous savez maintenant propager et récupérer des modifications avec un dépôt extérieur. Voyons comment sont gérés les conflits de modifications. Modifiez un fichier de `depot-travail`, faites un `commit`, puis un `push` de vos modifications. Rendez-vous dans `depot-travail-bis` et, sans faire de `pull`, faites une modification conflictuelle du même fichier. Faites alors un `commit`, puis un `push`. Que se passe-t-il ? Gérez le conflit jusqu'à sa résolution complète sur

`depot-travail-bis`, propagez les modifications avec `push` et récupérez-les sur `depot-travail` avec `pull`.

Représentez la fin de la chaîne des `commits` lors des différentes phases de gestion du conflit (après chaque commande git sur `depot-travail-bis`) en faisant attention à bien identifier le(s) parent(s) de chaque commit.

Remarque : Lorsqu'on s'apprête à faire de nombreuses modifications qui seront source de conflits, il est préférable :

- soit de de pas commiter ces modifications (pas de `add` sur les fichiers concernés) avant qu'elles ne soient finalisées, mais dans ce cas elles ne sont pas archivées ;
- soit de créer une nouvelle branche pour faire les modifications conflictuelles, de faire des commits sur cette branche, et de faire un `merge` des branches à la fin.