Projet informatique 2019-2020

TP - Git

Ce mini-TP / Tutoriel a pour objectif de vous faire tester Git pour que vous soyez en mesure d'utiliser cet outil à l'avenir.

1 Git

Vous devez faire ce TP de manière synchronisée avec le reste de votre groupe. Certaines questions vous demandent d'attendre que les autres membres de votre groupe aient fini pour avance. Si vous savez déjà utiliser `git`, aidez les autres membres du groupe à manipuler `git`. Si vous savez tous manipuler `git`, vous pouvez commencer le projet.

Nous utiliserons gitlab pour nos dépôts (`https://gitlab.com`). Comme expliqué en cours, il ne serait pas correct de vous forcer à créer un compte à votre nom sur gitlab. Si vous ne souhaitez pas créer un compte avec votre adresse ENSIIE, on vous propose de vous créer une adresse email factice avec laquelle créer ce compte. Vous aurez ensuite tout le loisir de supprimer votre compte gitlab et votre email à l'avenir. Bien entendu, si cela ne vous dérange pas, n'hésitez pas à créer un compte avec votre adresse ENSIIE.

Une question fera parfois référence à un des membres du groupe. Ils seront nommés `user1`, `user2`, `user3` et `user4`. A vous de choisir dans votre groupe qui est quel `user`. Certains noms de fichier contiennent `userX`, vous pouvez remplacer le X par votre numéro.

1.1 Créer et cloner un dépôt

1. Créez un compte sur gitlab (`https://gitlab.com` –> "Register") si vous n'en avez pas. Connectez vous ("Sign in").
2. `user1` doit créer un dépôt sur son compte. Ce dépôt sera vide à l'exception d'un README. ("New Project" –> Donner un nom au projet –> Niveau de visibilité privé –> Cocher "Initialize repository with a README).
3. `user1` doit ensuite ajouter les autres membres comme `developper` du groupe (À gauche, "Settings" –> "Members" –> Onglet Invite members –> Indiquer le nom d'utilisateur de `user2`, `user3` ou `user4` dans le premier champs –> Choisir le "role permission" `Developper` –> "Invite").
4. `user2`, `user3` et `user4` doivent accepter l'invitation.
5. Clonez ce dépôt sur votre machine (y compris `user1`). Sur gitlab, copiez l'adresse du dépôt (À gauche, "Project overview" –> À droite, "Clone" –> Copiez le lien associé à "Clone with HTTPS" [1])

```
> user:~$ git clone https://gilab... (lien copie)
```

   Un nouveau dossier est apparu dans votre dossier courant. Rentrez dedans avec la commande `cd`. Vérifiez qu'un fichier README existe bien dedans.

1.2 Status, Commit

6. `user1` doit créer quatre fichiers nommés `user1-A.txt`, `user1-B.txt`, `user1-C.txt` et `user1-D.txt`. De même pour `user2`, `user3` et `user4` (en changeant le numéro dans le nom de fichier).

```
> user:~$ touch userX-A.txt
> user:~$ touch userX-B.txt
> user:~$ touch userX-C.txt
> user:~$ touch userX-D.txt
```

1. Vous pouvez aussi ajouter une clef SSH à votre compte gitlab pour cloner le dépôt via SSH, mais pas sûr que ça fonctionne depuis les PC de l'école.

7. Regardez le statut de vos fichiers avec

```
> user:~$ git status
```

   Remarquez que tous vos fichiers sont en rouge, marqués comme "Fichier non suivi". C'est le mode `untracked` indiqué dans le cours. Pour l'instant, ces fichiers n'appartiennent pas au dépôt.

8. Ajoutez les fichiers `userX-A.txt`, `userX-B.txt` et `userX-C.txt` au dépôt avec

```
> user:~$ git add userX-A.txt userX-B.txt userX-C.txt
> user:~$ git status
```

   Remarquez que ces fichiers passent en mode "Modifications qui seront validées" (le mode `staged` du cours) et que rien n'a changé pour `userX-D.txt`.

9. Commitez ces fichiers avec

```
> user:~$ git commit
```

   Un logiciel de traitement de texte s'ouvre, remplissez la première ligne avec un message explicatif de commit (par exemple "Ajout de nouveaux fichiers pour le TP.")
   Refaites un statut pour remarquer que les trois fichiers ont disparu de la liste. (Le mode `unmodified` du cours)

10. Ouvrez et modifiez les fichiers `user-A.txt` et `userX-B.txt` en rajoutant une ligne dedans.
    Refaites un statut pour remarquer les deux fichiers sont passés en mode "Modifications qui ne seront pas validées" (Le mode `modified` du cours)

11. Ajoutez `userX-A.txt` au prochain commit avec

```
> user:~$ git add userX-A.txt
>
```

    Refaites un statut pour remarquer que le fichier `userX-A.txt` est en mode `staged`. Chacun de vos fichiers est maintenant dans un mode différent.
    Remarquez aussi sur gitlab que rien n'a changé. Toutes vos modifications sont locales à votre machine.

    Quelques explications
    Un commit permet de créer une nouvelle version de votre dépôt. Entre deux versions, un fichier (ou plusieurs) est ajouté, enlevé ou modifié.

    — `userX-A.txt` est en mode `staged`. C'est un fichier qui a été ajouté ou modifié depuis le dernier commit (ici il a été modifié). En cas de commit, cette modification sera présente dans la nouvelle version.
    — `userX-B.txt` est en mode `modified`. Ce fichier était présent dans la dernière version commitée, mais a été modifié depuis. Cependant, cette modification ne sera pas prise en compte dans le prochain commit.
    — `userX-C.txt` est en mode `unmodified`. Ce fichier était présent dans la dernière version commitée et n'a pas été modifié depuis.
    — `userX-D.txt` est en mode `untracked`. Ce fichier n'a jamais été présent dans aucune version commitée.

    Essayez de répondre aux questions suivantes avant de vérifier vos réponses en testant les commandes.

12. Dans quels états seront les 4 fichiers si vous faites un `git commit` ? Effectuez ce commit.
13. Ajoutez `userX-C.txt` au prochain commit avec `git add` sans le modifier. Dans quel état sera-t-il ?
14. Mettez `userX-C.txt` en état `unmodified`. Modifiez `userX-C.txt` puis annulez la modification. Dans quel état est ce fichier ?
15. Mettez `userX-A.txt` dans l'état `staged`. Modifiez `userX-A.txt`. Dans quel état est ce fichier ?
16. Modifiez `userX-D.txt`. Dans quel état est ce fichier ?
17. Modifiez `userX-B.txt`. Dans quel état est ce fichier ?
18. Si vous commitez `userX-B.txt`, combien de versions différentes de B auront été enregistrées dans le dépôt au cours des différents commits ?
    Vous pouvez continuer à manipuler les commandes `git add` et `git commit` tant que vous le souhaitez avant de passer à la partie suivante.

1.3 diff, log

19. Si ce n'est pas déjà le cas, mettez le fichier `userX-A.txt` dans l'état `modified`. Affichez tout ce qui a été modifié dans ce fichier depuis le dernier commit avec

```
> user:~$ git diff userX-A.txt
```

    Si vous voulez toutes les modifications de tous les fichiers `git diff` seul fonctionnera.

20. Commitez toutes vos modifications. Qu'affiche `git diff` ?
21. Modifiez le fichier `userX-A.txt` et mettez le dans l'état `staged` sans commiter. Qu'affiche `git diff` ?
22. Qu'affiche `git diff` sur un fichier `unmodified` ou `untracked` ?
23. Testez la commande suivante pour afficher tous vos commits.

```
> user:~$ git log
```

    Remarquez que les commits des autres membres du groupe n'apparaissent pas et que rien n'a changé sur le dépôt `gitlab`. Pour l'instant, tous vos commits sont locaux.

24. Utilisez la commande pour afficher une version compacte de `git log`.

```
> user:~$ git log --graph --pretty=format:"\%Cred\%h\%Creset \%Cgreen(\%cd) \%C(bold blue)<\%an>\%
    Creset \%s-\%C(yellow)\%d\%Creset"
```

1.4 pull et push

Attendez que tout le monde arrive à ce niveau pour avancer.

25. `user1` doit taper la commande suivante pour envoyer tous les commits en ligne sur `gitlab`.

```
> user:~$ git push
```

    Vous pouvez tous vérifier sur le dépôt gitlab que tous les commits de `user1` y sont présents.

26. `user2`, `user3` et `user4` doivent taper la commande `git push`.
    Remarquez un message d'erreur. Vous ne pouvez pas envoyer des commits sur un dépôt si quelqu'un a envoyé d'autres commits sur le dépôt avant vous. Pour être à jour, il faut récupérer tous ces commits sur votre machine.
    Pour remédier à cela, utilisez la commande

```
> user:~$ git pull
> user:~$ git push
```

    Remarquez que, une fois tous les utilisateurs à jour, tous les fichiers commités par tous les utilisateurs sont apparus sur votre machine et sur le dépôt `gitlab`.

27. Effectuez un log. Que remarquez vous ?

1.5 Conflits

Attendez que tout le monde arrive à ce niveau pour avancer.

28. Si vous avez des fichiez en état `modified` passez les en état `unmodified` avec `git checkout -- nomdufichier`.
29. `user1` doit modifier le fichier `user1-A.txt`, commiter et pousser (`git push`) sous la forme suivante :

```
Ceci est la premiere ligne.
-- 10 lignes vides --
Ceci est la seconde ligne.
```

    Les 3 autres peuvent récupérer les modifications avec `git pull`.

30. `user1` doit modifier la première ligne du fichier `user1-A.txt`, commiter et pousser sous la forme suivante :

```
Ceci n'est pas la premiere ligne.
-- 10 lignes vides --
Ceci est la seconde ligne.
```

    Les autres membres ne doivent pas récupérer cette modification.

31. `user2` doit modifier la deuxième ligne du fichier `user1-A.txt`, commiter sous la forme suivante :

```
Ceci est la premiere ligne.
-- 10 lignes vides --
Ceci n'est pas la seconde ligne.
```

Est-il possible de pousser ce commit sur le dépôt ? Que se passe-t-il si vous faites un `git pull` ?

32. `user3` doit modifier la première ligne du fichier `user1-A.txt`, commiter sous la forme suivante :

```
Ceci est tres certainement la premiere ligne.
-- 10 lignes vides --
Ceci est la seconde ligne.
```

Est-il possible de pousser ce commit sur le dépôt ? Que se passe-t-il si vous faites un `git pull` ?

Quelques explications

Dans la partie précédente, les utilisateurs ont poussé des commit relatifs à des fichiers différents. Aucun conflit n'apparaît.

`user1` et `user2` ont modifié 2 lignes différentes séparées par des lignes vides, `git` a pu effectuer une fusion automatique des modifications de `user1` et `user2`. (Cela peut dépendre de la version de git ou des options activées). git demande juste une confirmation à `user2`.

`user1` et `user3` ont modifié la même ligne, `git` n'a pas pu effectuer une fusion automatique des modifications et demande à `user3` de régler ce conflit. `user3` doit donc ouvrir ce fichier et décider comment résoudre ce conflit, éventuellement avec l'aide de `user1`.

1.6 Branches

Attendez que tout le monde arrive à ce niveau pour avancer.

Une branche est, en quelque sorte, une zone de travail. Si deux utilisateurs travaillent sur deux branches distinctes, il ne peut pas y avoir de conflit. On peut ensuite fusionner les branches pour réunir leurs travaux.

Jusqu'à présent, vous avez tous travaillé sur la branche `master`, provoquant ainsi des conflits sur les fichiers modifiés.

33. Créez une branche avec

```
> user:~$ git branch BuserX
```

34. Utilisez la commande suivante pour voir la liste des branches. Vous devriez voir `master` et `BuserX`, git vous indique que vous êtes actuellement sur la branche `master`.

```
> user:~$ git branch BuserX
> * master
>   test
```

35. Effectuez la commande suivante pour passer sur la branche `BuserX`.

```
> user:~$ git checkout BuserX
> Basculement sur la branche 'BuserX'
```

36. Effectuez la commande suivante pour rajouter cette branche sur le dépôt gitlab. Vous pouvez vérifier en ligne que la branche est apparue.

```
> user:~$ git push -u origin BuserX
> * [new branch]      BuserX -> BuserX
```

37. `user1`, `user2`, `user3` et `user4` doivent faire des commits sur le même fichier et les pousser, par exemple `userA-1.txt`, pour remarquer que personne n'a de conflit. Effectuez `git log` pour remarquer que les commits des autres membres n'apparaissent pas dans votre log. Vous êtes complètement séparé du travail des autres membres.
38. Effectuez `git branch -a` pour voir les branches ajoutées par les autres membres. Remarquez que ces branches ne sont pas sur votre machine.

```
> user:~$ git branch -a
> * master
>   BuserX
>   remotes/origin/Buser1
>   remotes/origin/Buser2
>   remotes/origin/Buser3
>   remotes/origin/Buser4
```

39. Effectuez la commande suivante pour récupérer la branche `BuserY` depuis le dépôt et tous ses commits (remplacez `Y` par le numéro d'un autre membre du groupe). Effectuez ensuite `git checkout BuserX` ou `git checkout userY` pour passer d'une branche à l'autre.

```
> user:~$ git checkout --track origin/BuserY
> La branche 'BuserY' est parametree pour suivre la branche distante 'BuserY' depuis 'origin'.
```

40. `user4` doit fusionner son travail, effectué sur la branche `Buser4` avec celui de `user1`. Passez sur la branche `Buser1` et effectuez la commande suivante pour fusionner les deux branches. Réglez les conflit, committez et poussez les résultats. Remarquez maintenant avec `git log` que les deux branches pointent au même endroit.

```
> user:~$ git merge Buser4
```