TD5 - Gestion de version

Ce TD, se déroule en plusieurs parties. Tout d'abord vous allez revoir le fonctionnement de git avec le chargé de TD qui va vous faire une démonstration en vous demandant vos prédictions, que vous noterez sur **papier**. Ensuite, vous allez appliquer cela sur machine, individuellement, puis en équipe.

Exercice 5.1. Démonstration Git

Dans cette première partie, vous allez individuellement avec le professeur noter l'évolution d'un suivi de version git réalisé entre le serveur et différentes machines.

Suivez avec le professeur l'évolution du dépôt en prenant des notes au vidéoprojecteur.

On considère la situation schématisée sur la première diapo du **rappel de cours** : **Luke** et **Leia** sont 2 utilisateurs qui exécutent la suite de commandes suivantes (dans des terminaux) dans le but de travailler sur un plan et de le versionner, sauvegarder, partager entre eux.

NB : pour ne pas rentrer dans le détail de l'édition de fichier, on invente une commande `write(Address file, int line, String msg)` *qui écrit dans le fichier* `file`*, à la ligne* `line` *le contenu* `msg`*. S'il y a déjà quelque chose d'écrit à la ligne* `line`*, le contenu est écrasé et remplacé par* `msg`*.*

SANS UTILISER D'OUTIL GIT MAIS EN PRENANT DES NOTES :

Question 5.1.1. Expliquez ce qu'il se passe en donnant le résultat de chacune des commandes suivantes, sur la machine de Leia, sur la machine de Luke, ainsi que sur le serveur.

1. `Leia@linux : git clone https://gitlab.univ-nantes.fr/Leia/episode4.git`

2. `Leia@linux : write(plan.txt, 1, Etoile Noire)`
3. `Leia@linux : git status`
4. `Leia@linux : git add .`
5. `Leia@linux : git status`
6. `Leia@linux : git commit -m "creation du plan"`
7. `Leia@linux : git status`

8. `Leia@linux : write(plan.txt, 2, Plan technique)`
9. `Leia@linux : git commit -m "Plan"`
10. `Leia@linux : git add .`
11. `Leia@linux : git commit`

12. `Leia@linux : write(plan.txt, 3, Diamètre : 120 kilomètres)`
13. `Leia@linux : git commit -a -m "diametre"`

14. `Leia@linux : git rm plan.txt -f`
15. `Leia@linux : ls`
16. `Leia@linux : git status`
17. `Leia@linux : git checkout -f`

18. `Leia@linux : git fetch`
19. `Leia@linux : git status`
20. `Leia@linux : git push`

21. `Luke@macosx : git clone https://gitlab.univ-nantes.fr/Leia/episode4.git`
22. `Luke@macosx : git log -1 --abbrev-commit`

23. `Luke@macosx : write(plan.txt, 4, Equipage : 265675 hommes)`
24. `Luke@macosx : git commit -a -m "equipage"`
25. `Luke@macosx : git fetch`
26. `Luke@macosx : git status`
27. `Luke@macosx : git push`

28. `Leia@linux : git fetch`
29. `Leia@linux : git status`
30. `Leia@linux : git diff master origin/master`
31. `Leia@linux : git merge`

32. `Leia@linux : write(plan.txt, 4, Equipage : 265675 hommes, 57276 artilleurs)`
33. `Leia@linux : git commit -a -m "MAJ Equipage"`
34. `Leia@linux : write(plan.txt, 4, Equipage : 265675 hommes, 57276 artilleurs, 400000 droides)`
35. `Leia@linux : git commit -a -m "MAJ Equipage"`

36. `Leia@linux : git fetch`
37. `Leia@linux : git status`
38. `Leia@linux : git push`

39. `Luke@macosx : write(plan.txt, 5, dont 167216 pilotes)`
40. `Luke@macosx : git commit -a -m "MAJ equipage"`
41. `Luke@macosx : git push`

42. `Luke@macosx : git fetch`
43. `Luke@macosx : git status`

44. `Luke@macosx : git diff master origin/master`
45. `Luke@macosx : git merge`

46. `Luke@macosx : résoudre le conflit dans un éditeur`
47. `Luke@macosx : git commit -a -m "merge conflict"`
48. `Luke@macosx : git push`

49. `Leia@linux : git pull`

Exercice 5.2. Deuxième partie – Application de git sur PC en ligne de commande

Mettez-vous en trinôme : simplement entre voisins (si le nombre de trinôme n'est pas juste dans un groupe de TD, il peut y avoir un et un seul binôme ou quadrinôme)

Question 5.2.1. Rendez-vous sur

https://univ-nantes.io/mottu-jm/butinfo-gpo2-projet-2023

Normalement on accède à gitlab par https://gitlab.univ-nantes.fr/ mais il y a des règles d'accès qui nécessitent de jongler entre un accès direct et un accès nomade. C'est pour cela qu'on utilise cette alternative (https://univ-nantes.io).

Si ça ne fonctionne pas, il faut probablement utiliser https://nomade.univ-nantes.fr/

Puis dans le champ à côté de Browse, mettre l'adresse https://gitlab.univ-nantes.fr/mottu-jm/butinfo-gpo2-projet-2023 et cliquer sur Browse. S'y connecter par LDAP avec vos login/password habituels.

Et si ça bloque dans les étapes suivantes, il faut probablement quitter nomade et vous reconnecter en direct à https://gitlab.univ-nantes.fr/

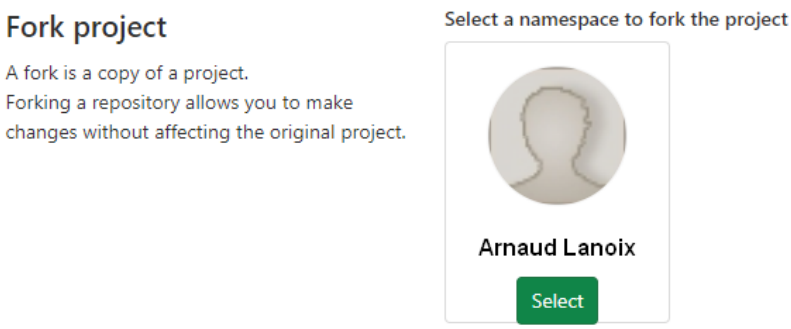
Vous n'avez pas le droit de modifier ce projet : vous n'en êtes pas un membre (avec suffisamment de droits).

Par contre, vous avez le droit de faire un « fork », au sens littéral c'est une bifurcation : cela crée un nouveau projet. Vous aurez le droit de travailler dans ce nouveau projet. De plus, il reste connecté au projet original, permettant des échanges entre les deux. Classiquement, cela permet de travailler de son côté, à l'extérieur de l'équipe du projet initial pour ensuite proposer ses contributions.

Imaginez par exemple que vous trouviez une faute d'orthographe. Je n'ai pas envie que vous la modifiez directement (parce que vous n'êtes pas dans l'équipe « prof » et je ne veux pas que vous trafiquiez le sujet), mais je serais content que vous me soumettiez une correction (spoiler : il y en a).

Un étudiant par trinôme crée un fork du projet :

A l'écran suivant choisissez votre namespace :



Cela va créer un fork du projet qui aura une autre adresse :
`https://gitlab.univ-nantes.fr/lanoix-a/butinfo-gpo2-projet-2023`

ou `https://univ-nantes.io/lanoix-a/butinfo-gpo2-projet-2023`
selon les besoins d'accès par nomade ou pas

Il ne s'agit donc pas du même projet, celui-ci n'est pas dans les projets de mottu-jm, mais dans les projets de votre identifiant (ici de M. Lanoix par exemple).

Tout d'abord, réglez la visibilité, vous allez y faire un travail personnel et vous ne voulez pas que quelqu'un le copie :

`Settings>General>Visibility>Project visibility: Private puis Save changes)`

Ensuite, ça facilitera le travail de tout le monde que vous renommiez votre projet en ajoutant « -nom1-nom2-nom3 » (en remplaçant avec vos noms respectifs) :

`Settings>General>Project name : butinfo-gpo2-projet-2023-nom1-nom2-nom3`

Finalement, invitez les membres de votre trinôme (pour qu'ils bossent) et votre chargé de TD (pour qu'il note votre travail quand il sera rendu).

`Members > Invite Member`, cherchez leurs noms et donnez-leur les droits de « Maintainer ».

Tout le monde peut voir le projet dans sa liste de projet `https://univ-nantes.io`

Question 5.2.2. Ouvrez un terminal, déplacez-vous dans un répertoire de travail créé pour l'occasion (en local sur votre machine ou votre dossier d'étudiant). Sur vos machines personnelles, il faut installer git au préalable : https://git-scm.com/

Sur la page d'accueil du projet (avant qu'il y ait déjà eu du travail effectué), quelques commandes devraient être données pour vous guider. Il faut d'abord vous identifier :

```
git config --global user.name "<identifiant>"
git config --global user.email "<mail>"
```

Puis, effectuez le clone dans ce répertoire (il y a plusieurs manières de s'authentifier, comme par exemple la suivante) :

git clone https://<identifiant>@gitlab.univ-nantes.fr/<identifiant créateur>/butinfo-gpo2-projet-2023

(en remplaçant bien les chaînes entre <> (en enlevant aussi les <>))

Ensuite, il faut se déplacer dans le dossier créé par le clone (c'est le dossier qui contient le dossier caché `.git`).

Cette fois vous allez travailler avec le fichier `SpecificationPlatoonVehicules.md` du dossier `cahier_des_charges`. Vous pouvez voir son code et faire un preview directement en ligne.

Effectuez un travail similaire à l'exercice 1 :

Question 5.2.3. Essayez l'enchainement classique des commandes sans conflits. Chacun votre tour ajoutez votre nom dans le tableau.

Faites la suite de commandes classiques pour chaque étudiant, chacun son tour :

1. On récupère le travail (*fetch*, *status*, *merge*)
2. On travaille et on enregistre en local le fichier `cahier_des_charges.md`
3. On ajoute à l'index (*add*), on versionne (*commit*)
4. On vérifie qu'il n'y a rien de nouveau (*fetch*, *status* et s'il y a du nouveau *merge*)
5. On pousse (*push*)

Faites cela, en observant l'évolution du dépôt en ligne (il faut actualiser la page) :

`Repository > Files-Commits-Graph`

Et en faisant des commandes `git status` pour l'évolution en local.

Question 5.2.4.

Perturbons cela en faisant des conflits :

1. Remplissez tous l'entête du fichier sur votre machine sans vous concerter.
2. Le trinôme1 versionne et partage son travail d'abord (ça ne devrait pas poser de soucis, comme à la Question 4).
3. Puis les trinôme2 et trinôme3 versionnent et essaient de partager leur travail :
    a. Ajout (add), versionnage (commit) sans soucis
    b. Normalement le fetch (nécessaire avant tout push) doit récupérer le travail du trinôme1, à vérifier avec la commande status.
    c. Puisqu'il y a du nouveau, il faut faire un merge (vous ne pourrez pas pousser votre travail sinon). Cela crée un conflit.
    d. Concertez-vous pour résoudre ce conflit : choisir quelle est la bonne variante de l'entête, en supposant qu'aucune des 3 variantes n'était parfaite
    e. Il faut mettre à jour le dépôt maintenant et repartager entre vous une variante commune (c'est ce qu'il y a de plus délicat)
        i. Supposons que le binôme3 s'occupe d'intégrer cette variante
            1. Il ouvre un éditeur et produit le fichier avec la variante choisie
            2. Il ajoute (add) et versionne (commit) le fichier et le pousse (push)
        ii. Le binôme2 n'est pas encore sorti d'affaire
            1. Soit il fait comme le binôme3
                a. Il fetch à nouveau pour récupérer la variante finale, résous le conflit (ce qui reviens à renoncer à ses changements), add, commit, push.
            2. Soit il annule sa version et fait un checkout (cf doc et exo1)
        iii. Le binôme1 aussi doit récupérer cette nouvelle version. S'il n'a rien fait depuis, alors il peut se contenter d'un pull (= fetch+merge).

Exercice 5.3. Travail collaboratif.

Changeons le niveau de droits des uns et des autres pour travailler sur le contrôle des contributions grâce aux branches.

On ne donne pas forcément les mêmes droits d'écriture sur le référentiel publique à tous les collaborateurs. Typiquement, ils ne sont pas « maintainer » mais seulement « developer », ils ont le droit de récupérer la branche principale (origin/master) mais pas d'y faire des pushs. Néanmoins ils ont le droit de créer d'autres branches sur le référentiel publique pour travailler. Finalement quand ils pensent que leur travail mérite d'être intégré, ils demanderont aux maintainers de récupérer leur travail : c'est un « *merge request* » (dénomination de gitlab, « *pull request* » pour github).

Descendez le niveau des droits du binôme2 à « developer ».

Question 5.3.1. Travaillez cette série d'action :

Le trinôme2 écrit une phase d'intro dans la section 1 :

« L'entreprise ElectroIUT souhaite informatiser un distributeur de boissons. »

Il essaie de partager cela sur le dépôt partagé : *add*, *commit*, *push*.

Que se passe-t-il ?

Question 5.3.2. Travail sur une branche

1. `Trinôme2 : git branch collaboration`
2. `Trinôme2 : git checkout collaboration`
3. `Trinôme2 : ls`
4. `Trinôme2 : Vérifier le contenu de cahierdescharges.md`
5. `Trinôme2 : git push origin collaboration`

Il crée une branche (*branch*), il bascule sa copie de travail sur cette branche pour y travailler (*checkout*), il travaille (versionne), il pousse sur le dépôt partagé (origin) la branche (collaboration).

Question 5.3.3. Une fois cette contribution faite sur sa branche, il va falloir l'intégrer à la branche principale. Cela se passe en ligne sur https://univ-nantes.io.

Trinôme2 demande la création d'un « `merge request` » qu'on documente pour convaincre les maintainers du référentiel public.

Les autres trinômes peuvent aller sur la page « `merge requests` » pour trouver la demande. Cela peut ouvrir des discussions, on peut aller comparer les branches, analyser le graphe, etc.

Si les maintainers acceptent, l'un d'eux fait « *merge* », ce que tout le monde peut vérifier en ligne et en local :

1. `git pull`
2. `git checkout master (pour le Trinôme2)`

Un *pull* récupère l'ensemble du dépôt avec toutes ses branches, un *checkout* bascule la version de travail dans la branche voulue pour ouvrir et vérifier son contenu.

Exercice 5.4. Outil graphique et intégré

Tout cela peut être fait avec des outils plus graphiques. Soit directement dans un IDE (un environnement de développement logiciel), par exemple utiliser eclipse et le plugin eGit, ou IntelliJ, soit avec un outil dédié à git (sourcetree par exemple sous windows ou mac, gitkraken sous linux) pour effectuer un travail similaire.

Question 5.4.1. Continuez votre projet de cette manière.