TP Git

Git est un système de contrôle de version distribué. Il permet de suivre les modifications et de gérer différentes versions du code, et de collaborer efficacement.

Il existe d'autres outils de gestion de version, tels que Subversion (SVN) et Mercurial. Mais Git se distingue aujourd'hui par des fonctionnalités clés :

- il est distribué
- il inclue la création de branches pour travailler de manière isolée
- il est d'une grande performance via les commits qui ne stockent pas leurs données comme une série de modifications mais plutôt comme une série d'instantanés (snapshot).

GitHub et GitLab sont des plateformes de développement collaboratif basées sur Git. Elles offrent des outils similaires, mais GitLab se distingue par la possibilité de l'auto-héberger, offrant ainsi une plus grande souveraineté et un contrôle sur l'infrastructure.

Git est généralement intégré aux IDE via des plugins ou autre. Pour ce TP mais aussi pour plus de compréhension, vous manipulerez `git` via la CLI.

Instanciez une VM Docker sur eCloud. Git y est installé et cette VM servira pour les TP suivants.

Voyez cet énoncé comme une feuille de route, sur laquelle vous pourrez vous appuyer par la suite si besoin.

Votre premier projet

Créez un répertoire **GIT/projet1** puis créez un fichier vide dedans : *fichier1*

Connectez vous sur https://im2ag-gitlab.univ-grenoble-alpes.fr

Créez un nouveau projet et décochez « Initialize repository with a README ».

Dans un 1er temps, configurez votre nom, prénom et adresse mail :

```
git config --global user.name "username"
git config --global user.mail "email"
```

Puis comme ici vous êtes dans le cas où vous avez déjà un répertoire existant, suivez les instructions adéquates.

```
cd existing_folder
git init --initial-branch=main
git remote add origin https://im2ag-gitlab.univ-grenoble-alpes.fr/.....
git add .
git commit -m "Initial commit"
git push -u origin main
```

Lors du push, vous devriez avoir un message d'erreur

Pour résoudre cette erreur, il vous faut un **Access Token** à utiliser à la place du mot de passe. Un Access Token est un jeton d'accès utilisé pour authentifier les utilisateurs, souvent en remplacement d'un mot de passe.

Pour créer cet Access Token, allez sur le projet depuis im2ag-gitlab > Settings > Access Tokens.

Puis mettre à jour l'url du dépôt distant :

git remote set-url origin https://<username>:<token>@im2ag-gitlab.univ-grenoble-alpes.fr/.....

Poussez (push) de nouveau !

PS : GitLab privilégie plutôt la sécurité en établissant la connexion avec votre clé ssh, mais pour des raisons de simplicité et d'efficacité, nous faisons avec l'access token pour ce TP.

Créez un nouveau fichier README.md, écrivez ce qu'il vous enchante dedans.

Ajoutez (git add), commitez (git commit) puis poussez (git push) sur le dépôt.

Comprendre son rôle.

Consultation des logs `git log`

Vous pouvez ajouter des options à la commande `git log` :

```
git log --oneline --graph --name-status --abbrev-commit
git log -p
```

Consultation de l'état

Ajoutez un *fichier2*, écrivez quelque chose dedans, puis faites la commande :

```
git status
```

Que donne cette commande ?

PS : n'hésitez pas à vous servir très fréquemment de ces deux commandes `git log` et `git status`. Je dirai même plus, mettez des alias pour ces deux commandes dans votre bashrc.

Différence entre commits `git diff`

Vous avez des commits qui ont été créés, faites un diff entre deux d'entre eux (avec leur ID) :

```
git diff <commit1> <commit2> : commit2 étant le plus récent
```

Revenir en arrière (sur un commit)

Avant de commencer avec les `git revert`, je vous conseille de modifier l'éditeur de texte par défaut pour git, en mettant votre éditeur préféré :

```
git config --global core.editor "vim"
```

Créez un fichier3 puis commitez son ajout.

Maintenant pour revenir en arrière (et donc supprimer ce fichier3), il faut faire un revert sur ce dernier commit.

```
git revert <last commit>
```

La logique derrière `git revert` peut sembler contre-intuitive mais elle découle d'un principe fondamental de Git : **la conservation de l'historique**. Un `git revert` créé un nouveau commit qui annule les modifications du commit renseigné.

Vous pouvez aussi revenir en arrière sur un commit avec la commande `git reset`. Cette commande déplace le pointeur HEAD vers un commit spécifique (elle n'annule pas les modifications du commit renseigné). Mais ATTENTION : cette commande, en fonction des options supprimera les commits (et donc les historiques). PRIVILEGIEZ `git revert`.

Annuler des modifications dans un fichier

Faites des modifications dans un fichier puis annulez ces modifications :

```
git restore <nom du fichier>
```

ATTENTION : si vous avez modifié ce fichier puis vous l'avez indexé (`git add`). Il vous faudra ajouter une option à `git restore` pour le supprimer de l'indexation. L'option est **--staged**.

```
L'indexation dans Git est un processus qui consiste à préparer des modifications pour un commit
```

Une autre commande intéressante :

```
git clean -f : supprimer un fichier non indexé (-d pour un dossier)
```

Git stash/pop

Admettons que vous êtes entrain de travailler sur tel fichier et que finalement vous souhaitez travailler en priorité sur un autre fichier. Vous souhaitez donc mettre temporairement de côté votre modification pour vous concentrer sur l'autre fichier, le pusher puis revenir sur votre fichier. C'est possible avec `git stash`.

Créez un fichier puis indexez le. Au final vous souhaitez ne pas travailler sur ce fichier (pour le moment) mais sur un autre :

```
git stash : mettre de côté
git stash list : lister les stashes
```

Qu'affiche la commande `git stash list` ?

Faites une modification d'un autre fichier puis pushez le.

Maintenant revenez sur votre travail sur le premier fichier :

```
git stash pop
```

Que constatez-vous ?

Analyse ligne à ligne d'un fichier (git blame)

La commande `git blame`permet de lister les dernières modifications de chaque ligne d'un fichier.

```
git blame <fichier>
git blame -L 05,10 <fichier> : de la ligne 5 à 10
```

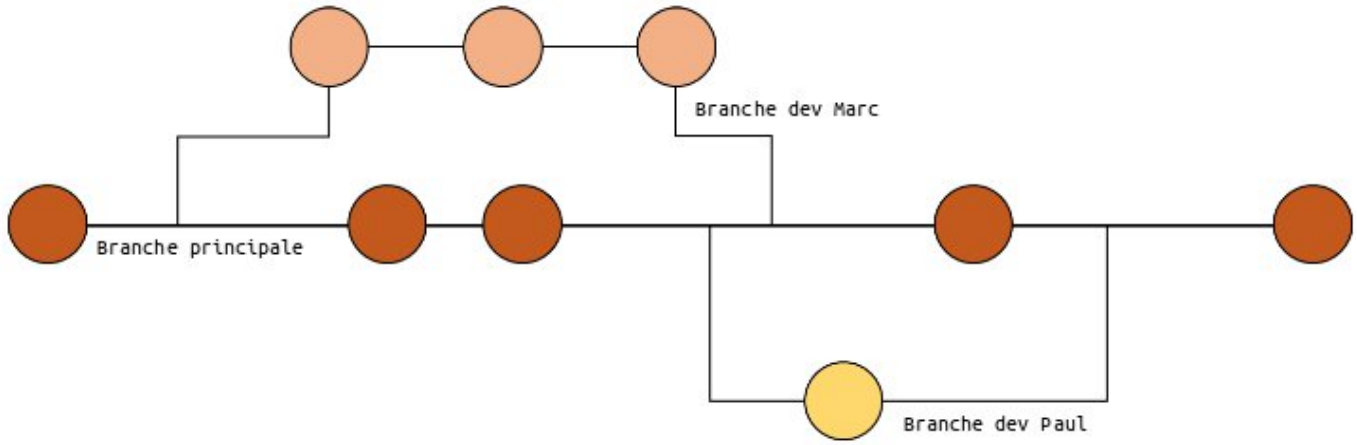
Dans le **fichier1** écrivez le nombre de ligne que vous souhaitez, puis faites un `git blame` dessus. Que constatez-vous ?

Indexez et commitez cette modification. Maintenant que constatez-vous ?

Avec cette commande vous pouvez donc voir **qui** a fait **quoi**, à **quelle heure**, sur **quel commit**.

Les branches

Sur Git, à l'initiation, une branche principale (main) est créée. Il est possible de créer d'autres branches de manière à travailler de façon isolée et à travailler sans impacter la branche principale. Les branches facilitent la gestion des versions en permettant de travailler simultanément sur plusieurs fonctionnalités ou correctifs sans interférer avec la version principale. Elles permettent également une meilleure collaboration entre développeurs, en isolant les contributions et favorisent une organisation claire des environnements de déploiement (développement, test, production)."




```
git branch <nom_branche> : création d'une branche
git checkout <nom_branche> : se positionner sur telle branche
git merge <branche_source> : import d'une branche source vers la branche actuelle
Git checkout -b nom_branche> : créer une branche et se positionner directement dessus
```

Créez une nouvelle branche <username>, déplacez vous dessus, faites une modification (ajout de fichier par exemple), indexation, commit, push. Que se passe-t-il ?

```
git push --set-upstream origin <nom_de_la_branche>
```

Allez sur im2ag-gitlab constatez la nouvelle branche puis faites un merge :

```
git merge <branche_source> : depuis la branche main
```

Pour l'organisation des branches, chaque équipe/service peut faire comme il le souhaite. Mais une méthode semble tout de même être privilégiée : une branche PROD (main), dessous une branche DEV (ou PRE-PROD), et sur cette dernière on met des branches par utilisateurs, ou aussi parfois par fonctionnalités.

Créer une nouvelle branche « dev », puis mettez les 3 branches à jour pour qu'elles soient bien à la même version. Via les logs, vous pouvez voir si c'est le cas sur le HEAD.

Faites un `git log`, que constatez-vous ?

Travail en équipe

Mettez-vous en binôme et créez depuis https://im2ag-gitlab.univ-grenoble-alpes.fr un nouveau projet sur l'interface de l'étudiant1. L'étudiant1 clone ce nouveau projet en local, puis créé un fichier1 et le poussez (vérifiez que tout est ok côté im2ag-gitlab).

Il va falloir maintenant que l'étudiant1 ajoute l'étudiant2 à son projet. Avant toutes choses, quelques définitions :

- **Guest** : permissions très limitées, souvent en lecture seule
- **Reporter** : responsable de la création et de la gestion des issues (tickets)
- **Developer** : peut cloner le dépôt, créer des branches, pousser des commits, ouvrir des merge requests, et souvent, fusionner des branches après revue
- **Maintainer** : approuve et fusionne des merge requests, gère les branches, publie des versions, etc.
- **Owner** : contrôle total, peut tout faire

Pour notre exercice, l'étudiant1 va ajouter l'étudiant2 en tant que développeur puis créer un Access Token pour développeur (copiez-le de côté). L'étudiant1 va aussi créer depuis im2ag-gitlab une branche `dev` pour l'étudiant2 **qu'il va ensuite protéger**. Pour protéger cette branche, allez dans « Settings > Repository > Protect a branch » puis autorisez les `merge` et `push` aux développeurs et mainteneurs. Que constatez-vous au niveau de la branche `main` ?

Clonez maintenant le projet en local pour l'étudiant2.

En tant qu'étudiant2, essayez de poussez un nouveau fichier depuis la branche `dev` et faites pareil depuis la branche `main`. Que constatez-vous ?

La solution à cela.... **Le merge request !**

Il est possible d'effectuer les `merge request` depuis un terminal en local avec l'outil `**glab**`, mais ici vous allez le faire depuis l'interface web de Gitlab. A la fin de cette manip vous devez avoir en local sur les deux machines, les deux branches à jour avec le `merge request` validé.

Mais d'ailleurs... comment voir les changements distants ? Et comment mettre à jour en local le projet/dépôt distant ?

Gestion de conflit

Les deux étudiants vont modifier le même fichier sur la branche `dev` (de manière différente pour plus de lisibilité). L'étudiant1 va pousser cette modification, ok... Puis l'étudiant2 va aussi pousser sa modification. Que se passe-t-il ?

Essayez de vous débrouiller par vous même pour résoudre le conflit avec l'aide du retour console de Git. Créez plusieurs conflits de manière à tester les deux premières options du `git pull` (la troisième donnera généralement une erreur d'abandon, donc pas la peine).

Maintenant que vous avez vu les branches, les merge request et les conflits...

Un avis personnel ??

Et sur VScode...

Git est complètement intégré à la plus part des IDE. Généralement vous utiliserez VScode, ouvrez le et installez l'extension "GitLab Workflow". Ajoutez un projet à VScode et testez comment manipuler Git avec.

Faites vous la main !!

PS : Je vous invite aussi à tester les `git stage`, qui n'ont pas été abordés jusque là, mais qui permettent de sélectionner précisément quelles parties des modifications d'un fichier vous souhaitez ajouter au staging avant de committer. Cela offre une grande flexibilité pour gérer les changements par petits incréments et éviter de committer des modifications non désirées

Conclusion

En tant que développeur vous utiliserez quotidiennement cet outil pour la majorité d'entre vous. Il est donc très important d'en comprendre les principes, les possibilités et les fonctionnalités.

D'ailleurs je vous invite à poursuivre votre montée en compétences sur Git de manière ludique sur le site suivant : https://learngitbranching.js.org/

Comme vous venez de le voir, vous n'utiliserez sûrement pas cet outil en CLI, mais plutôt directement depuis vos IDE via des extensions.

Je vous encourage à regarder sur le net et à vous renseigner auprès d'autres développeurs, sur les outils et extensions (à intégrer à votre IDE, ou à part) les plus utiles pour faciliter votre travail avec Git. Profitez-en pour découvrir de nouvelles façons d'optimiser votre flux de travail et améliorer votre efficacité au quotidien !