Rapport individuel : Git / GitHub

Encadré par : Imane Gannaoui

Réaliser par : Khadija Amardoul

Youcode

TABLE DES MATIÈRES

Introduction :

Un système de contrôle de version, ou VCS, suit l'historique des modifications lorsque des personnes et des équipes collaborent sur des projets. Au fur et à mesure de l'évolution du projet, les équipes peuvent effectuer des tests, corriger les bogues et contribuer au nouveau code, en toute confiance. Les développeurs peuvent consulter l'historique du projet pour savoir : Quelles modifications ont été apportées, qui a fait les changements et quand les changements ont-ils été effectués.
Git est un exemple de système de contrôle de version distribué (DVCS) couramment utilisé pour le développement de logiciels open source et commerciaux.

Git :

Dans l'intérêt de connaitre git, moi et les autres membres d'équipe avons résolu 3 scénarios différents. Ces derniers se compose des différentes étapes. Dans chaque étape on découvre les commandes de git et leurs arguments utiles.

Or, avant de parler des commandes de git, nous devrions d'abord savoir ce qui est git et quel est l'intérêt de l'utiliser. Bien évidemment, Git est un logiciel de gestion de versions décentralisé. Son but est de suivre l'évolution d'un code source, pour retenir les modifications effectuées sur chaque fichier et être ainsi capable de revenir en arrière en cas de problème. Ainsi il permet à plusieurs personnes de travailler sur le même projet sans risquer de se marcher sur les pieds. Si les deux modifient un même fichier en même temps, leurs modifications doivent pouvoir être fusionnées sans perte d'information. Allons maintenant découvrir le monde de git

Premier Scenario :

Premièrement je vais parler de tout ce que j'ai appris dans les 4 premier étapes de scenario 1. D'ailleurs, pour d'utiliser git il faut d'abord le configurer en ajoutant le nom et le mail d'utilisateur, cela permet de garder la traçabilité, chose qui est très importante surtout dans un projet collectif. Pour le faire on utilise la commande suivante « git config –global user. name/email ' nom ou email' ».Puis on va créer un dossier et un fichier dans ce dossier et aussi on va l'initialisé ce fichier grâce aux commandes suivantes : 'mkdir « nom de dossier »' pour la création de dossier, 'touch file' pour la création de fichier ' et 'git init' pour finalement initialiser ce dernier.
Cependant, si on ajoute un autre fichier ou on fait une modification quoi que ce soit, il ne faut jamais oublier de sauvegarder. C'est pourquoi on utilise les commits, qui permettent de figer l'état d'un dossier à un moment précis dans le temps afin de pouvoir effectuer différentes

opérations dessus. En effet on pourrait vouloir comparer les fichiers à deux moments différents dans le temps, ou bien même vouloir revenir en arrière. C'est à cela que servent les commits. Ainsi on utilise les commandes suivantes : « git add . » pour indexer le contenu de fichier et « git commit –m ' message de commit' » et en peut aussi utiliser la commande suivante qui permet de faire le adding / commiting au même temps : « git commit –am ' messgae de commit
' » .

En outre, quand on initialise le projet qu'on a créé sur GitBach, un dossier caché « .git » est automatiquement créé. Ce dossier git contient toutes les informations nécessaires au projet dans le contrôle de version. Les informations sur les validations, et l'adresse du référentiel. Il contient également un journal qui stocke l'historique de validation afin qu'on vous puisse revenir à l'historique. D'autre part si on veut que certains fichiers soient exclus de notre projet il suffit d'ajouter. gitignore qui a pour rôle d'ignorer les fichiers que on veut pas visionner dans Git.

Revenant au commit , la commande ' git log' nous permet d'afficher tous les commit qu'on a fait, mais si on veut affichez le dernier commit, avec l'option d'affichage de la hiérarchie de la branche, avec les commits et leur branche aussi. On utilise la commande suivante : ' git log -all --graph –decorate'.

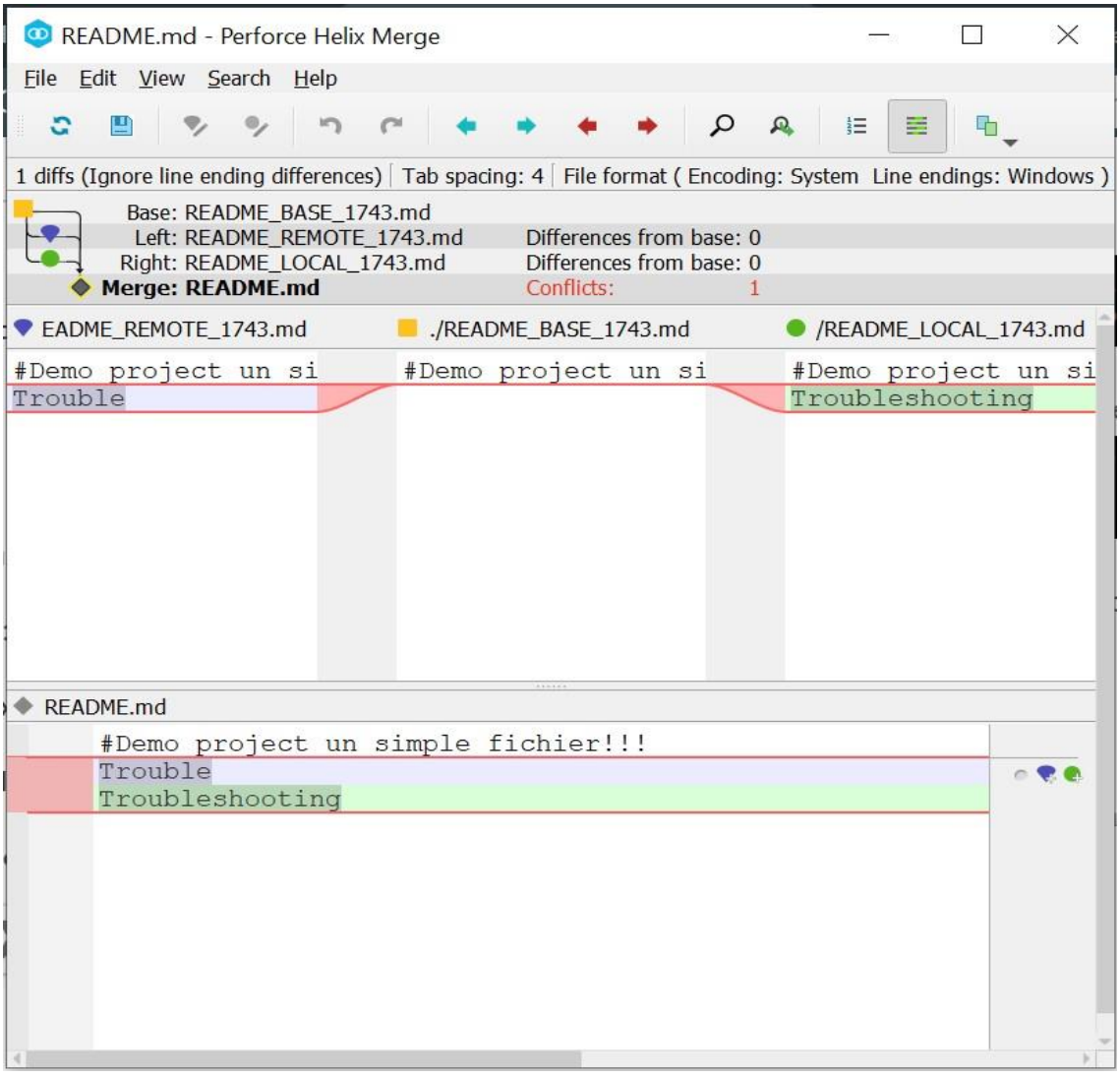
Cette commande 'git log --all --graph –decorate' est trop long n'est-ce pas.si on ne veut pas avoir à taper l'intégralité du texte de chaque commande, on peut facilement définir un alias pour chaque commande en utilisant git config : « git config --global alias. 'nom d'alias' 'nom de commande'»

Après avoir connaitre la configuration de git et les différentes commandes de base, nous allons maintenant faire un petit résumé de ce qu'on a vu dans les étapes 5,6 et 7 du scenario 1 qui parlent des branches, de fusionnement, et aussi les conflits engendrés après le fusionnement.
D'ailleurs les branches dans git est une ligne d'évolution divergent de la ligne d'évolution courante(master), celles-ci se poursuivant indépendamment l'une de l'autre. Elle permette par conséquence de se lancer dans des évolutions ambitieuses en ayant toujours la capacité de revenir à une version stable et aussi de pouvoir tester différentes implémentations d'une même fonctionnalité de manière indépendante. Pour crée la branche on utilise la commande suivante : git branch « nom de la branche », on utilise aussi « git checkout 'nom de la branche ' » pour switcher à cette dernière.

En effet, la création de branche est un outil indispensable et très efficace. Mais, le problème s'engendre lorsque on veut que les modifications que on a fait la branche se fusionnent avec le master. Dans certains cas, la fusion peut être effectuée automatiquement, car les informations de l'historique sont suffisantes pour reconstruire les modifications et ces dernières ne sont pas en conflit, on applique la commande « git merge 'nom de la branche' ».

Dans d'autres cas, une personne doit décider exactement de ce que les fichiers résultants doivent contenir.de nombreux outils logiciels existent mais on a choisi dans le scenario 1 de travailler avec le programme p4merge. Ainsi On a configuré git en ajoutant p4merge comme un mergetool et on a édité la configuration de prompt au niveau de fichier .gitconfig :

Tapant la commande « git mergetool », voilà le résultat :



Deuxième et troisième scenario :

Dans ce deuxième scenario j'ai découvrit deux nouveaux concepts : tagging et stashing.

Tagging

Les tags sont un aspect simple de Git, ils vous permettent de marquer des points spécifiques de l'historique d'un référentiel comme importants. Généralement, les utilisateurs utilisent cette fonctionnalité pour marquer les points de publication (v1.0, v2.0, etc.).

Il existe deux types de tags dans Git : **annonated** et **Lightweight**. Ils vous permettront tous les deux de faire référence à un commit spécifique dans un référentiel, mais ils diffèrent par la quantité de métadonnées qu'ils peuvent stocker. : **annonated** tags stockent des métadonnées supplémentaires telles que le nom de l'auteur, les notes de publication, le message de balise et la date sous forme d'objets complets dans la base de données Git.de l'autre côté **Lightweight** tags elles ne stockent que le hachage du commit auquel elles font référence. Ils sont créés en l'absence des options -a, -s ou -m et ne contiennent aucune information supplémentaire.

Pour la création d'un tag, on utilise la commande suivante : « git tag -a V1.0 -m ' nom de tag' »

Stashing :

git stash stocke temporairement (ou cache) les modifications que on a apportées à notre copie de travail afin que nous puissions travailler sur autre chose, puis revenir et les réappliquer plus tard. Stashing est pratique si nous devront changer rapidement de contexte et travailler sur autre chose, mais on est au milieu d'un changement de code et nous ne sommes pas encore prêt à s'engager. Pour réintégrer dans le répertoire de travail et l'index les données contenues dans le dernier stash il faut utiliser la commande suivante : « git stash pop ».

GITHUB :

Commençons la nouvelle aventure de GitHub. En premier lieu, GitHub est un site Web et un service en nuage qui aident les développeurs à stocker et à gérer leur code, ainsi qu'à suivre et

à contrôler les modifications apportées à leur code. Dans le scenario 2 et 3 nous avons exploré les différents aspects de GitHub et nous avons aussi compris comment le lier avec la version locale de Git.

Clonning / Pushing :

On peut créer un référentiel sur GitHub et le cloner pour créer une copie locale sur notre ordinateur et effectuer une synchronisation entre les deux emplacements, en utilisant la commande « git clone url ». D'autre part si on a fait des modifications sur git et on veut les visualisé sur GitHub il suffit de pousser tout le projet existant sur la version locale vers le serveur GitHub avec la commande suivante : « Git push –u origin master ». D'un autre côté, sur GitHub en peut créer un fichier et l'initialiser et aussi faire des modifications

NB : le –u « set-upstream » est utilisé pour ajouter une référence de suivi au serveur en amont sur lequel nous poussons. Aussi il faut mentionner que si on veut pousser un tag on doit ajouter '—tags' la commande s'écrit comme suit : « git push –u origin master –tags'

Fetch & Pull :

Lorsque nous apportons des modifications dans github, il est nécessaire de tirer ces modifications vers gitbach. Pour ce faire, nous pouvons utiliser soit git fetch ou git pull.

En fait, git fetch télécharge uniquement les nouvelles données d'un référentiel distant, mais n'intègre aucune de ces nouvelles données dans vos fichiers de travail. Fetch est idéal pour obtenir une nouvelle vue sur tout ce qui s'est passé dans un référentiel distant.En raison de sa nature "inoffensive", vous pouvez être assuré: fetch ne manipulera, ne détruira ni ne bousillera rien. Cela signifie que vous ne pouvez jamais aller chercher assez souvent.

Git pull, en revanche, est utilisé avec un objectif différent : mettre à jour votre branche HEAD actuelle avec les dernières modifications apportées par le serveur distant. Cela signifie que pull télécharge non seulement de nouvelles données ; il les intègre également directement dans vos fichiers de copie de travail actuels.

Pull request :

Un Pull Request est une demande de pull (ajout) d'un peu de code au répertoire, à un projet sur GitHub.
Il arrive que vous vous demandiez si certaines modifications ou ajouts de code que vous réalisez peuvent être utilisés et intégrés au projet. Ces changements peuvent intéresser des anomalies ou des nouvelles fonctionnalités. En effet, Une fois un dépôt distant cloné en local, il est facile de mettre régulièrement à jour sa version, à l'aide de la commande git pull.Par contre pour envoyer ses versions développées localement sur le dépôt distant, cela nécessite une pull request.

Rebase :

Rebase est un autre moyen d'intégrer les changements d'une branche à une autre. Bien évidemment le Merge et le Rebase ont le même objectif. Ils sont conçus pour intégrer les modifications de plusieurs branches en une seule. Bien que l'objectif final soit le même, ces deux méthodes fonctionneront de manière différente. D'ailleurs, Rebase compresse tous le changement en un seul « patch ». Ensuite, il intègre le patch sur la branche cible. Ainsi pour faire le rebase on utilise la commande suivante : « git rebase master »

Cette commande déplace donc l'intégralité de la branche « featur » au-dessus de la branche principale. Pour ce faire, il réécrit l'historique du projet en créant de nouveaux commits pour chaque commit dans la branche « featur » originale.

Gestion de projet :

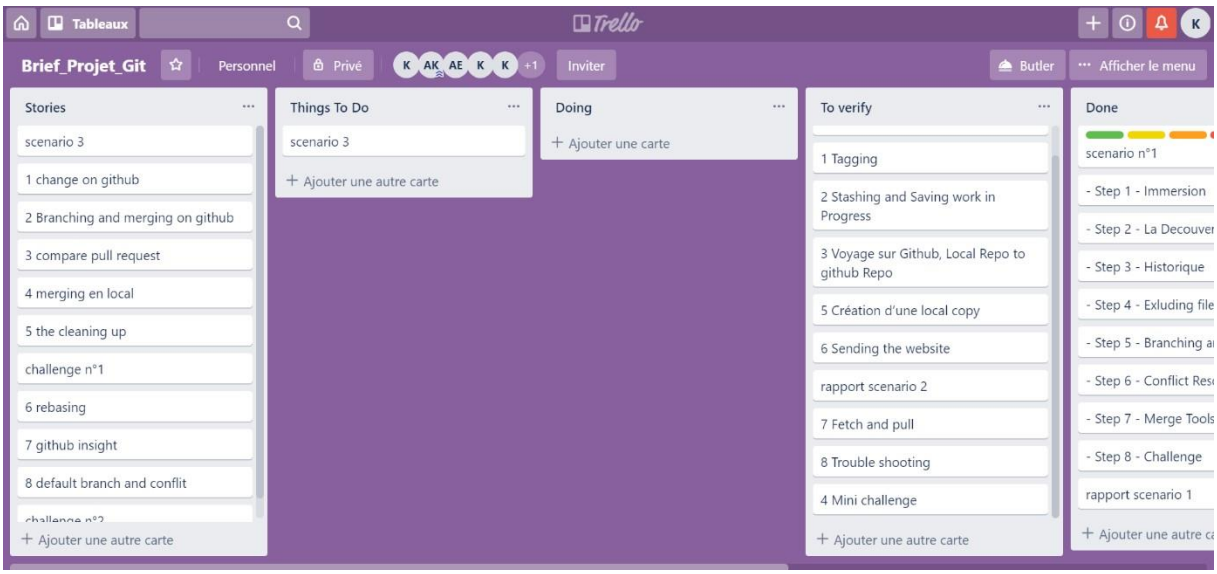
Le management d'équipe demande la motivation, la dynamisation, et une communication saine entre les collaborateurs afin d'avoir une prise de décision intelligente et un travail en équipe efficient. De ce fait, nous avons travaillé avec une méthode de gestion de projet agile : la méthode scrum. Cette méthode est considérée comme un cadre ou « framework » de gestion de projet constitué d'une définition des rôles, de réunions et d'artefacts. En effet, chaque rôle est attribué au chacun des membres du projet :

- Le product Owner : "le propriétaire" du produit, celui qui va définir les caractéristiques finales souhaitées pour le produit.

- Le SCRUM master : il s'agit du "maître SCRUM". C'est lui qui aura pour but de superviser les membres en les aidant, en les encourageant, en facilitant le partage, la communication des informations essentielles. Dans notre équipe c'était moi le scrum master.

- L'équipe de développement : elle recouvre les autres membres d'équipe qui en charge de la réalisation du produit.

D'un autre coté on a utilisé trello comme outil de gestion de projet. Cet outil collaboratif nous a permet d'organiser les tâches, gérer le temps pour et aussi de voir la progression de notre projet. Voici une capture d'écran de nos tableaux sur trello



Conclusion

En conclusion, Git fournit un moyen de garder une trace des versions antérieures des logiciels et des papiers, facilitant la collaboration entre différents auteurs et assurant la sauvegarde de votre logiciel. Cela s’est avéré très utile pour la communauté des logiciels libres et les milieux universitaires. Git a aussi plusieurs autres commandes qui peuvent être utiles pour effectuer des actions moins standard telles que remove données sensibles, nettoyer l'historique pour économiser de l'espace, etc.