*GIT Commande*

Git commit

*Un commit dans un dépôt Git (repository) enregistre une image (snapshot) de tous les fichiers du répertoire. Comme un Copier-Coller géant, mais en bien mieux ! Git fait en sorte que les commits soient aussi légers que possible donc il ne recopie pas tout le répertoire à chaque commit. En fait, Git n'enregistre que l'ensemble des changements ("delta") depuis la version précédente du dépôt. C'est pour cette raison que la plupart des commits ont un commit parent -- ainsi que nous le verrons plus tard. Pour cloner un dépôt, il faut décompresser ("résoudre") tous ces deltas. C'est la raison pour laquelle la commande écrit :*

*git branch [nom]*

*Branches Git Les branches sous Git sont incroyablement légères. Elles sont simplement des références sur un commit spécifique -- rien de plus. C'est pourquoi beaucoup d'enthousiastes répètent en cœur : des branches le plus tôt possible, et des branches souvent Parce qu'il n'y a pas de surcoût (stockage/mémoire) associé aux branches, il est facile de diviser son travail en de nombreuses branches plutôt que d'avoir quelques grosses branches. Nous verrons comment les branches et les commits interagissent quand nous les utiliserons ensemble. Pour l'instant, souvenez-vous qu'une branche est un moyen d'exprimer "Je veux inclure le contenu de ce commit et de tous les commits parents."*

*git checkout [nom]*

Indiquons à Git que nous voulons nous positionner sur la branche avec

git checkout [nom]

Cela nous positionne sur la nouvelle branche avant de faire un commit avec nos modifications

*Branches et Merges*

*git merge*

*Super ! Nous savons désormais comment faire des commits et des branches. Maintenant nous devons apprendre comment combiner ensemble les contenus de deux branches différentes. Ceci nous permettra de créer une nouvelle branche, développer une nouvelle fonctionnalité sur cette dernière, puis intégrer cette fonctionnalité en combinant le contenu de cette branche de développement à la branche d'origine (main par exemple). La première méthode que nous allons voir pour combiner le contenu de deux branches est git merge. Faire un 'merge' avec Git crée un commit spécial qui a deux parents. Un commit avec deux parents indique en susbtance "Je veux inclure le contenu de ce parent et le contenu de cet autre parent, et l'ensemble de leurs parents."*

Git Rebase

La seconde façon de combiner les contenus de deux branches est rebase. Rebase prend un ensemble de commits, les "recopie", et les ajoute en bout de chaîne à un autre endroit. Bien que cela puisse sembler compliqué, l'avantage de rebase est de permettre d'obtenir une simple séquence linéaire de commits. Les logs/l'historique du dépôt seront bien plus propres si seul rebase est autorisé (plutôt que merge).

HEAD

Premièrement nous avons parlé de "HEAD". HEAD est le nom symbolique pour le commit sur lequel nous nous situons actuellement -- plus simplement c'est le commit sur lequel nous travaillons. HEAD pointe toujours sur le commit le plus récent dans l'arbre des commits. La plupart des commandes Git qui modifient l'arbre des commits vont commencer par modifier HEAD. Normalement HEAD pointe sur le nom d'une branche (comme bugFix). Quand vous effectuez un commit, le statut de bugFix est modifié et ce changement est visible par le biais de HEAD.

*git checkout [nom] avec ^*

Comme je l'ai dit, spécifier un commit par son identifiant n'est pas très pratique, c'est pourquoi Git a des références relatives. Elles sont géniales ! Avec les références relatives vous pouvez commencer par vous placer à un endroit mémorisable (comme la branche bugFix ou HEAD) et travailler depuis cet endroit. Les commits relatifs sont puissants, et on va en présenter deux simples ici :

Revenir d'un commit en arrière avec ^

Revenir de plusieurs en arrière avec ~<num>

L'opérateur"~"

Imaginons que vous souhaitiez remonter beaucoup de niveaux dans l'arbre des commits. Cela peut être ennuyeux d'utiliser ^ plusieurs fois, c'est pourquoi Git a aussi l'opérateur tilde (~). L'opérateur tilde prend optionnellement à sa suite un nombre qui spécifie le nombre de parents que vous souhaitez remonter.

Forcer les branches

Vous êtes maintenant un expert des références relatives, alors servons-nous en. L'une des principales raisons pour lesquelles j'utilise les références relatives est qu'elles permettent de réorganiser les branches. Vous pouvez directement réassigner les branches à un commit avec l'option -f. Ainsi la commande suivante :

git branch -f main HEAD~3 bouge (de force) la branche main à trois parents derrière HEAD.

Git checkout HEAD~4

Git Reset

git reset annule des changements en déplaçant la référence en arrière dans le temps sur un commit plus ancien. En ce sens, on peut considérer cela comme une façon de "réécrire l'histoire"; git reset fait remonter une branche en arrière comme si le(s) commit(s) n'avai(en)t jamais eu lieu

Git Revert

Bien que le reset marche parfaitement pour les branches locales sur notre propre machine, cette façon de "réécrire l'histoire" ne marche pas avec les banches distantes (remote) que d'autres personnes utilisent. Pour pouvoir annuler des changements et partager ces annulations avec d'autres, nous devons utiliser git revert.