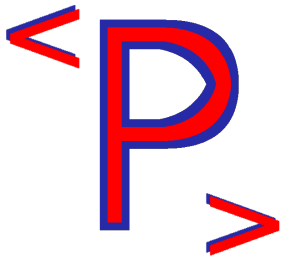

Independant

Année de Céation du Projet
2021

Base
Git & GitHub

Auteur 1 : POREE Baptiste

Table des matières


Les Base

1) Les Base de Git et Github

1.1: Naviguer dans un shell

pwd savoir dans quel dossier je me trouve
ls liste le dossier courant
cd on vas dans le dossier
mkdir créer un dossier
touch créer fichier

1.2: Initialisation de Git

— git config - -global
user.nameabel "toto titi"
user.email mon@mail.com
- **-list** Affiche les information dans le ficher global
color.diff auto
color.satuts auto
color.branch auto
core.editor editeurText on initialise l'editeur de text par defaut de git (notpad++,Visual Studio Code, ...)
merge.tool vimdiff

1.3: Premier version d'un dépôt

git init Pour initialiser le dépôt (.git dans le dossier courant)

1.4: le BAB

git add ♯chemin ajouter les modification des ficher du ♯chemin ("."."*" pour tout) (le Stage)

git commit -m "mesage" valider et sauvegarde les ficher qui on était indiquer dans le add ("le(s) fiche" si vous voulait être spécifique), le message est la pour se rappeler pour nous et pour les autres se qu'il a était fait. (le Repository)

git push envoie les commit verre le serveur GitHub (serveur)

git push recquet envoie les commit verre le serveur GitHub (serveur) open source sur la branche principal

git pull mes a jour le code en local par rapport au code de GitHub (serveur)

git status permet de voir le status de notre répertoire

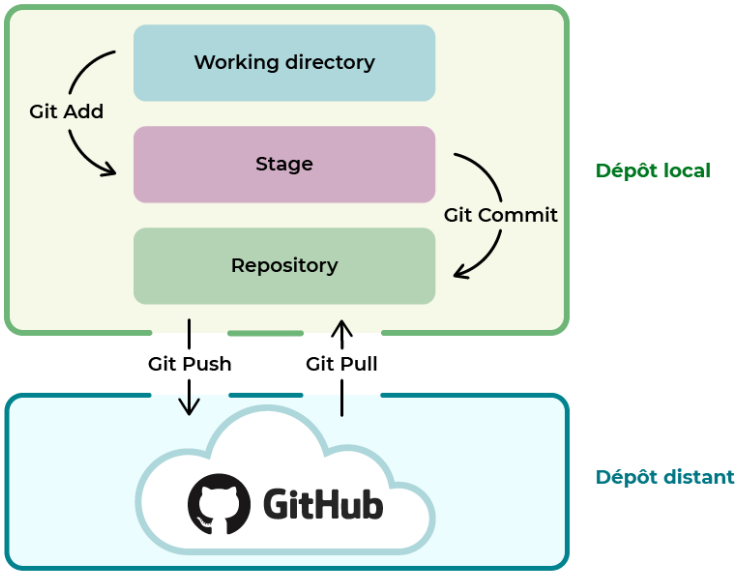

FIGURE 1.1 – Shéma logique de git avec github

1.5: lien entre git et le serveur

git remote add origin ♯url utiliser au début (git init)
vous souhaitez maintenant le lier avec votre dépôt local sur Git. DEP représente le nom court qui sera utilisé pour appeler le dépôt.

git clone ♯url utiliser plus quand on n'as déjà un dépôt crée
vous allez devoir dupliquer son contenu en local.

1.6: lien entre les branche d'un dépôt

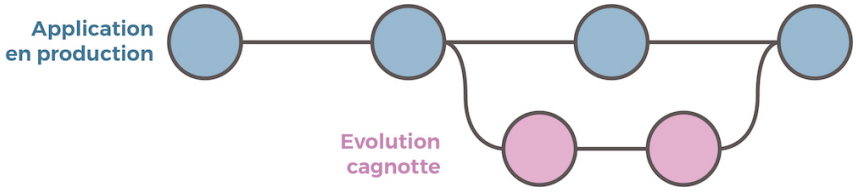

Figure 1.2

git branch voir la liste de toute les branche sur notre dépôt (* indique sur la quel on n'est et sur cel que l'on travail dessus)

git branch ♯bracnche crée une nouvel branche

git branch -d ♯bracnche supprimer une branche nommer

git branch -M ♯bracnche se connecte a la branche principal (master ou main depuis le 1er octobre 2020 pour le github)

git checkout ♯bracnche ce connecte a la branch

git push -u origin ♯bracnche on dépose le code sur la branche.

git merge ♯bracnche fusion des branche encoure (celui où nous somme et celui indiquer ♯bracnche)

3 phases de gestion de version
— Untracked : non suivi (dépôt local ne prend pas en compte les modification des ficher)
— Staging : préparait a la partager
— commit : finaliser les modification

1.7: Gestion des erreurs en local

1.7.1: supprimer une branch

git branch -d ♯bracnche supprimer une branche nommer

1.7.2: modifier la branche par erreur alors que c'était une autre viser

git stach sauvegarder dans une "remise" les modification avec le add déjà fait.

git stach list pour voir la liste des "remise"

git stash apply stash@x Prend la modification et la mes dans la branche situer. (stash@x : intituer dans le stach liste)

1.7.3: modifier la branche après avoir fait un commit

git log liste de tout les commit

git log -n2 liste des deux dernier commit

git reset - -hard HEAD^ le HEAD^ indique que c'est bien le dernier commit qui vas être supprimer

git reset - -hard ♯Author le Author (8 permier caractères sont nécessaire) nous mes a la version demander supprime les autre

git show ♯Author vas sur le commit de la version

git checkout ♯Author vas a la version du code demander

git checkout main vas a la version la plus rasante

git reset applique le commit supprimer

1.7.4: modifier un message commit

git commit - -amend -m "message" supprimer l'ancien message par le nouveau. Si il y pas de push qui a était fait et sur le dernier commit en local.

1.7.5: oublier un ficher dans le commit

git add ♯ficher ajouter un ficher

git commit - -amend - -no-edit ajouter un ficher a l'ancien commit. Si il y pas de push qui a était fait et sur le dernier commit en local.

1.8: Gestion des erreurs déjà push

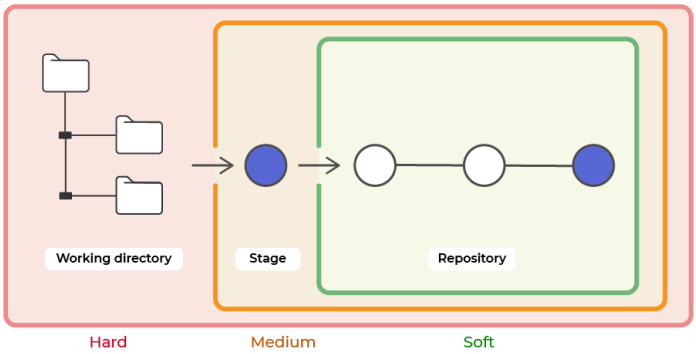

FIGURE 1.3

1.8.1: corrigez nos erreur en local et distance

git revert pour **annuler des changements d'un commit**, apportés à une branche publique

git reset pour **annuler des changements non commit**, mais sur une branche privée

git reset va revenir à l'état précédent sans créer un nouveau commit, alors que git revert va créer un nouveau commit. Schéma explicatif de la différence entre git revert et git reset

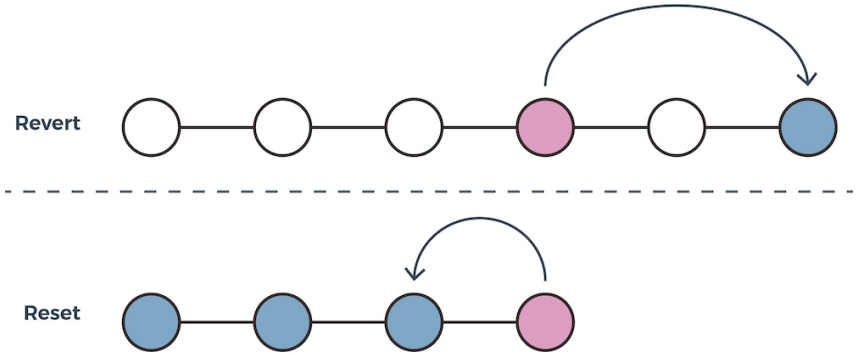

FIGURE 1.4

— git reset est une commande puissante. Elle peut être appliquée de 3 façons différentes (–soft ; –mixed ; –hard).
— La commande git merge produit un conflit si une même ligne a été modifiée plusieurs fois. Dans ce cas, il faut indiquer à Git quelle ligne conserver.
— git reset permet de revenir à l'état précédent sans créer un nouveau commit.
— git revert permet de revenir à l'état précédent en créant un nouveau commit.

1.9: corrigez un commit raté

git log voir l'arborescence et tout les commit fait par qui avec quel ficher

git reflog voie les commit et action effecteur par nous même

git blame ♯ficher qui a fait quel commit et où dans le code dans un ficher spécifique

git cherry-pick ♯Author permet de voir un commit particule et voir se qu'il a changer dans le code

— git log affiche l'historique des commits réalisés sur la branche courante.
— git reflog est identique à git log. Cette commande affiche également toutes les actions réalisées en local.
— git checkout un_identifiant_SHA-1 permet de revenir à une action donnée.
— git blame permet de savoir qui a réalisé telle modification dans un fichier, à quelle date, ligne par ligne.
— git cherry-pick un_identifiant_SHA-1 un_autre_identifiant_SHA-1 permet de sélectionner un commit et de l'appliquer sur la branche actuelle.

1.10: Découvrez l'arbre Git et sa Structure

commit qui va pointer vers un arbre spécifique et le marquer, afin de représenter son état à un instant donné (général)

tree où l'arbre Git qui est une forme de répertoire. Il va référencer une liste de trees et de blobs (sous-répertoires et fichiers)

blob qui représente en général un fichier (Binary Large Object) (un ficher spécifique)

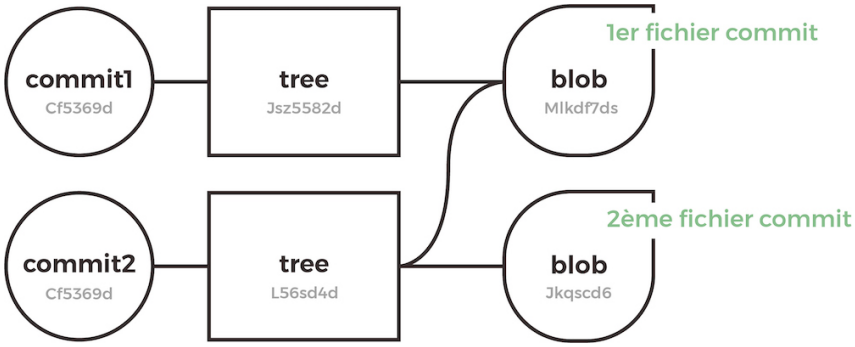

Figure 1.5

1.10.1: mettre au propre ca branche

git fetch prend les commit récent et garde de coter dans un ficher local (sans impacter notre dépôt local)

git merge prend le ficher local et intégré dans le dépôt local vas fusionner les branche

git pull prémet de faire git fetch et git merge en même temps

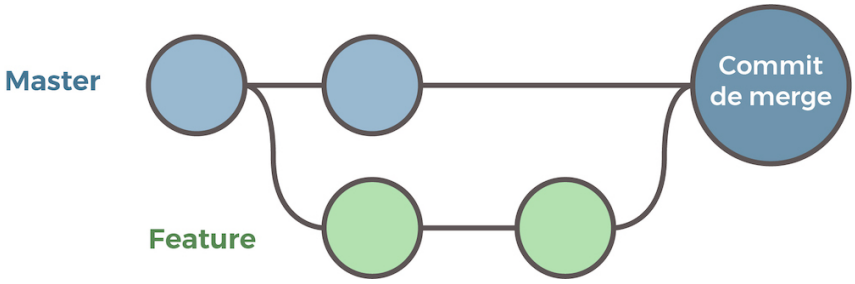

Figure 1.6

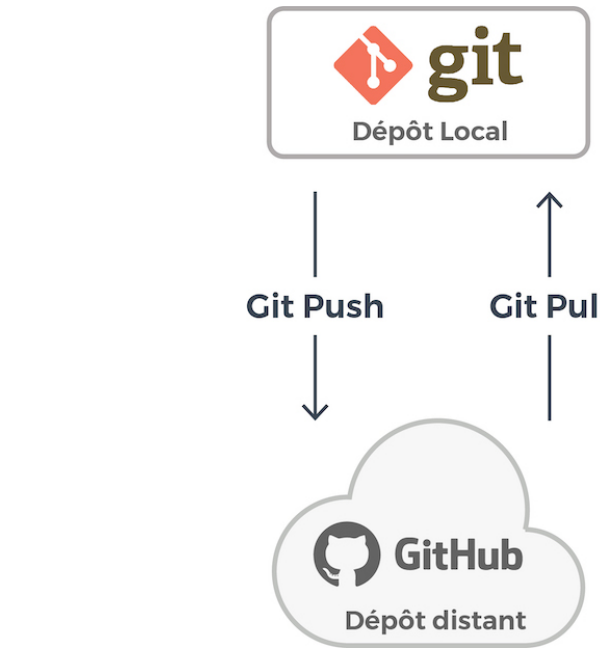

FIGURE 1.7

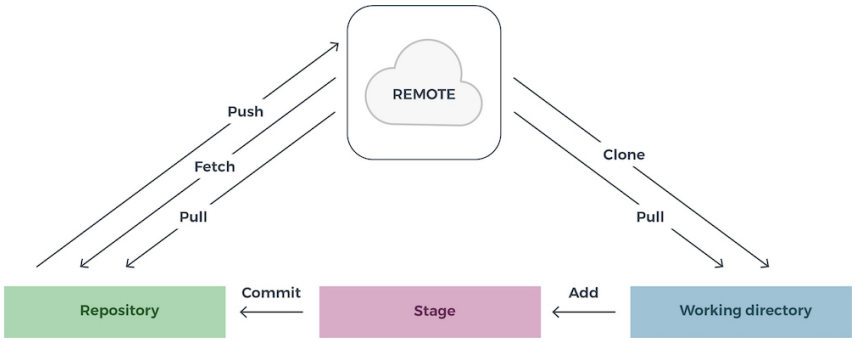

FIGURE 1.8

git rabase -i ♯Author mettre le commit d'une branche x sur une branche y voir même sur la branch main

git interactive rebase permet de déplacer des commit dans l'ordre historique et peut les fusionner

Commandes avec rebase :
— p, pick = utilisez le commit
— r, reword = utilisez le commit, mais éditez le message de commit
— e, edit = utilisez le commit, mais arrêtez-vous pour apporter des changements (git commit - -amend|git rebase - -continue)
— s, squash = utilisez le commit, mais intégrez-le au commit précédent
— f, fixup = commande similaire à "squash", mais qui permet d'annuler le message de log de ce commit
— x, exec = exécutez la commande (le reste de la ligne) à l'aide de Shell
— d, drop = supprimez le commit

1.10.2: nettoyage de branche

— élément périmer (branche, ficher) qui est plus bon ou plus du tout utiliser ou non suivie
— Rebesage interactif (nettoyer et réorganiser les commit du dépôt)
— Reformulait les message des commit pour qu'il sont plus explicite

a la fin tout le monde doit comprendre comment on n'as travail sur le projet et qui pas être le seul a voir le chemin fait.

Squash

permet de regrouper plusieurs commit entre

git rebase -i ♯Author afficher le ficher et mettre un "squash" a la ligne.

Où est le bug

git bisect start ♯Author(bug) ♯Author(clean = non buger) fait une comparaison entre les deux pour trouver où est le bug qui a apparut pour la premier fois

git bisect good commit qui ne contient pas le bug (donc il sarrette de cherche)

git bisect bad commit qui contient encre le bug (donc il vas cherche plus loin)

1.10.3: lien entre bibliothèque (github)

submodule lien avec l'arbre de l'autre github sans le copier

git subtree pull clone l'arbre de l'autre github sur le notre et on peu le modifier si on le souhaite et se mais a jour avec juste un pull

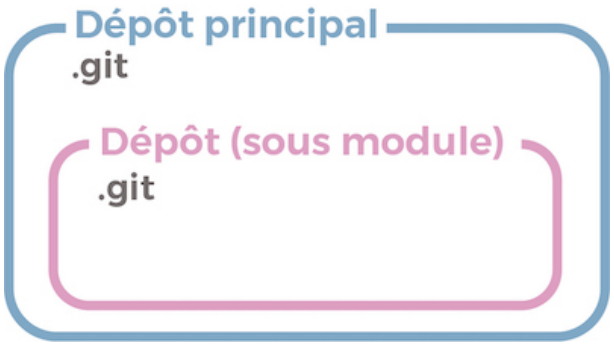

Figure 1.9

1.11: WorkFlow

1.11.1: gitFlow

git flow init : pour crée une arboraisance d'un projet de travail sous la forme d'un gitFlow

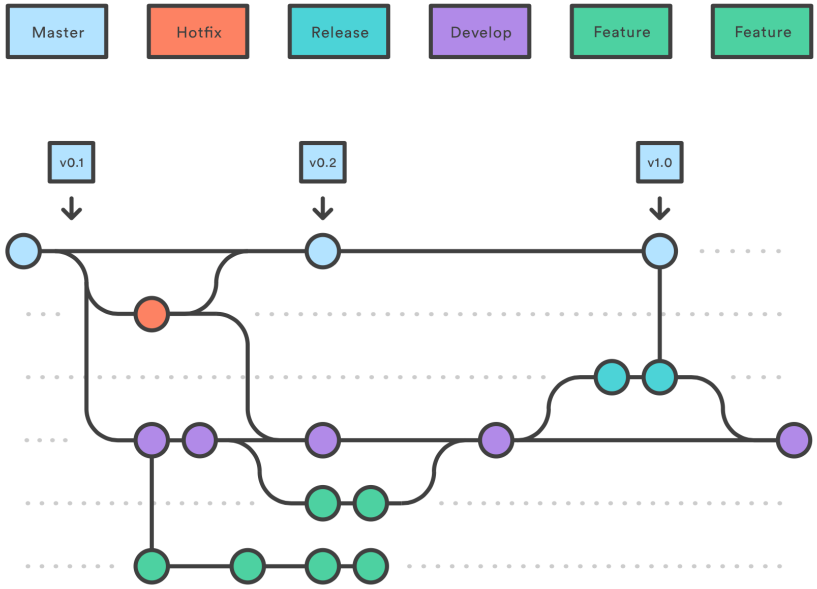

FIGURE 1.10

Branche anglais	Branche francais
master / main	principale
feature	fonctionnaliter
develop	developpement
release	corection de bug
hotfix	corection de bug sur master/main

FIGURE 1.11

Branche	Nombre	Branche d'origine	Durée de vie	Fonction
master	Unique		Permanente	Code stable, testé et validé potentiellement éligible pour une MEP (Mise En Production)
feature	Plusieurs	develop	Développement d'une feature	Code en cours de développement qui implémente une fonctionnalité à embarquer dans la prochaine version de l'application
develop	Unique	master	Permanente	Code de la prochaine version de l'application. Une fois que le développement d'une fonctionnalité (*feature*)est fini, le code est fusionné sur cette branche
release	Unique	develop	Recette	Branche sur laquelle on corrigera les bugs détectés pendant la phase de recette
hotfix	Aucune/Plusieurs	master	Correction d'un bug	Branche où on fait les corrections des bugs sur le code présent sur la branche `master` (production)

FIGURE 1.12

1.12: outils et leur fonctionnalités

1.12.1: WhiteSource Bolt

Dans le domaine de la sécurité, nous pouvons citer WhiteSource Bolt !

WhiteSource Bolt for GitHub est une application gratuite, qui analyse en permanence tous vos dépôts, détecte les vulnérabilités des composants open source et apporte des correctifs. Il prend en charge les référentiels privés et publics !

1.12.2: ZenHub

Dans le domaine de la gestion de projets, nous pouvons citer ZenHub

ZenHub est le seul outil de gestion de projet qui s'intègre de manière native dans l'interface utilisateur de GitHub. ZenHub est un outil de gestion de projet agile fonctionnant par sprint et générant des rapports assez poussés.

1.12.3: Travis CI

Dans le domaine de l'intégration continue, nous pouvons bien sûr citer Travis CI, qui est la référence en la matière.

Travis CI permet à votre équipe de tester et déployer vos applications en toute confiance. Très polyvalent, il s'adapte aux petits comme aux grands projets !

1.12.4: WinMerge et Meld

Ce petit outil est indispensable en développement ! Il vous sera très utile pour comparer deux versions de votre fichier. Les deux ont exactement le même but, comparer simplement deux fichiers en indiquant les zones où votre code est différent !

En plus de vous indiquer les différences entre vos deux fichiers, vous allez pouvoir les fusionner de façon intelligente. Pour chaque ligne différente, l'outil de comparaison vous demandera quelle version vous souhaitez conserver. Il est donc indispensable en cas de conflit dans Git.

Par exemple, vous avez travaillé sur une fonctionnalité et votre collègue aussi. Au moment du push, vous avez un conflit sous Git, car les lignes que vous avez modifiées ont aussi été modifiées par votre collègue. Grâce aux outils de comparaison, vous n'écrasez pas le code de l'autre bêtement !

1.13: gitLab / CI-CD

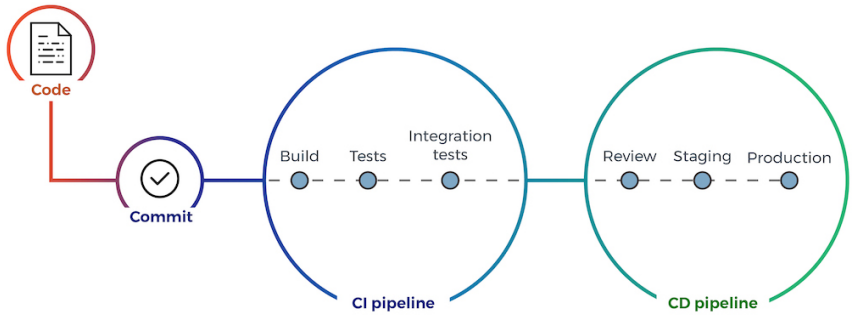

FIGURE 1.13

Le CI pipeline comprend : build, tests, integration tests.

Le CD pipeline comprend : review, staging, production (cf. schéma ci-dessus).

1.14: Gérez les demandes de pull

Les badges permettent de garder le projet à jour et indiquent une certaine qualité. Les badges peuvent être utilisés pour tout un tas de choses (sécurité, tests, version, maintenabilité...).