1. afficher l’url de remote origin :

**git remote show origin**

1. git add . : ajouter tous les fichiers modifiés à l’index sauf les fichiers supprimés

git add “\*.html” git add –all ajout les fichiers supprimés

1. Annuler les changements dans le work directory (supprimer modification non indexe)

**git checkout fichier**

1. Annuler les modifications indexes du fichier(le contraire de git add file)

**git reset HEAD fichier** ou bien **git reset -- fichier**

**git reset** : va revenir à l’état unstaged (non indexé) et sauvegarde les modifications (dangereux on ne peut pas la défaire)

**git reset --hard** : va revenir à l’état unstaged (non indexé) et a les modifications

**git reset commit\_id :** supprimer toutes les commites postérieurs à « commit\_id » mais garde l’espace de travail intacte.

**git reset HEAD^** : retourne d’un cran (un seule commit)

**git reset HEAD^^** : retourne de deux crans (2 commits)

**git reset HEAD^** **(soft, mixed, hard) = (modifications stagé on fait Just un commit, modifications non stagé on fait add et un commit ,supprime les modifications)**

Cette commande n’est pas sécurisé elle retourne en arrière et supprime l’historique

1. Afficher les détails des modifications non indexes avec la versions dans le dernier commit

git diff

1. git log don l’historique des comites

* git log –n 2 : don les 2 dernières commites
* git log –oneline : on une seule ligne s’il yà bcp de commits
* git log –p path : voir les commits spécifique à un fichier ou un répertoire

1. git commit –a –m pour abrégé (-a pour stagé)
2. git checkout « id du commit » : navigué dans l’historique des commit (voir le fichier comment il est avant ce commit)

**git checkout maste**r : retourné à la branche mater

**git checkout idcommit filename :** annuler les modification sur le fichier filename(défaire ce qu’on fait)

1. git revert commit\_id : défit un commit (supprimer les changements de ce commit) revert et ajouté à l’historique
2. git stash
3. git branch : pour tester une expérimentation (changer le code source sans être sur que je vais le sauvardé)
4. git checkout –b test : créer une nouvelle branche et se met sur la nouvelle branche teste
5. git checkout nom de la branche
6. conflits
7. logging avec blame
8. git show sha : voir exactement ce qui est passé dans le commit
9. ignorer des fichiers qu’on veut pas envoyé au repot distant mais resterons localement en ajoute un fichier «.gitignore » puis ajouté sur chaque ligne les fichiers et les dossiers à ignoré
10. git stash suspendre un travail puis le reprendre après.
11. Git stash pop recharger le stash et supprime ce qu’il y’à dans le stash
12. Config

Soit local propre au dépôt au global pour tous les dépôts. Les deux variable à configurer et user.email et user.name ..etc

Pour voir la configuration on fait git config –list

1. Git status : l’état du dossier
2. Zone stagging zone de transit

////////////////////////////////////les commande de staged area

Montrer que stagged utilisé pour selectionner le changemet a effectuer et celui à ne pas effectué

1.echo «hello word » >> hello.txt

2. faire “git ls-files -s” pour voir comme quoi il y’à rien dans le staged area

4. faire gs –s pour voir « ?? heelo.txt »

5. git add hello.txt

6. faire “git ls-files -s” pour voir comme sua hello.txt est dans le staged area

7. faire gs –s pour voir «  A heelo.txt » A : pour added

8. echo >> « change 1 » >> hello.txt

9. git –s : affiche “AM hello.txt” modified in work tree

10 faire gs –v et (voir a/h.txt(stagged) b/hel.txt(work directory)) et le hash

9. pour savoir qu’est ce qu’il y’à dans le stage area

**git ls-files –s**

11. git no-pager diff ( compare le stagged et le work directory)

12. git no-pager diff --satged (compare le stagged et le depo git)(nul rien dans le depot)

13. echo >> « change- 2 » >> hello.txt

14 git add –p (ajouter la ligne change- 2 mais pas la ligne change- 1)

15. faire ? puis e et supprimer « change- 1 »

16. git no-pager diff --satged (compare le stagged et le depo git)(nul rien dans le depot)

17 git restore hello.txt pour supprimer les changelment dans work directory

18 cre un repertoire vide puis faire git status ( src napparit pas tracker les fichiersl)

19 Git mv hello.txt src/hello.txt et voir (il ajoute src a untracked)

19 Git mv src/hello.txt hello.txt et voir (src n’apparait pas dans gs)

/////////////////////////tag

Git tag v0.1.0 pour sortir une release

Pour envoyé le tag g

Git push origin v0.1.0

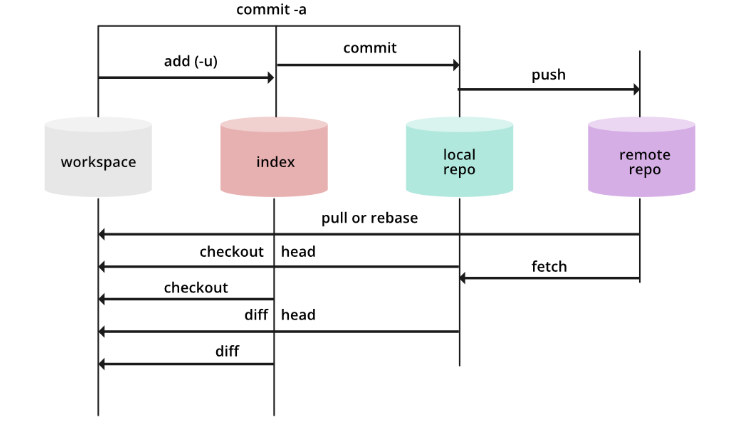
Git push origin –d v0.1.0

git

git merge

Git reset --hard

Git reset --soft



//////////////

Avec l’outil de visualisation montrer la branche expérimentale.

Leur montrer le détache branch puis l’attaché à la nouvelle branch

////////////////////// les conflits

1. crée un repo master avec fichier txt
2. crée une nouvelle branche à utiliser comme conflit de merge(git checkout –b dev)
3. modifier le « fichier txt » dans dev branch et exécuté le commit
4. retourne vers master modifier le fichier text et commité
5. fait un merge (git merge dev)

//////////////////////////repot distant dans un dossier

Git init --bare

///////////////////////////generate ssh file

Server git authentifie avec la clé public SSH. Chaque utilisateur doit générer une si n’a pas déjà une.

1. on doit tout d’abord vérifie si on a pas déjà généré une clé public

* vérifie le dossier .ssh qui se trouve dans le home (avec une paire de fichiers id\_rsa ou id\_dsa et un autre fichier .pub)

1. on peut les créer avec le programme ssh-keygen fournit par la package SSH ou bien avec git dans Windows
2. pose deux questions l’emplacement et la deuxième question et la passephrase
3. si on utilise un pwd on fait –o pour stocké la clé de manière plus sécurisé (brute force)
4. on peut utiliser ssh-agent evité d’entrer chaque fois un mot de passe

/////pull resuest

Mettre à jour le fork

1. Git add upstrem « url de repo d’origin ou on à forker »
2. Git checkout master(Git checkout main)
3. git rebase upstream/master(git rebase upstream/main)

////////////////////////////

**Mainline développement** :

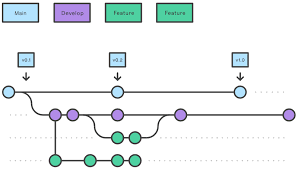
Les changements sont fait directement dans master, zéro branchement, l’équipe font attention de leur processus de qualité.la revue de code est une maniéré importante d’évité au développer d’introduire des problèmes de sécurité et de qualité. La vérification de code à l’apparition est meilleure que la vérification après l’intégration

* Il faut être confiant des suites de teste, une fois que les tests sont en vert on puise intégrer le pull request dans le release

Il faut corriger les bugs qui vont apparaitre dans le release et aller en avant et de ne pas revenir en arrière.et ajouter un test pour que le bug ne se reproduit pas au future, réparé le problème et faire le release avec la répartition

//////////////

**Branches fonctions :** chaque nouvelle fonction doit résider dans son propre branche.

****

**Workflow gitflow** : https://www.atlassian.com/fr/git/tutorials/comparing-workflows/gitflow-workflow

Définie un modèle de branchement stricte autour des releases du projet. Utilisé dans les grands projets et idéales pour les projets qui ont des releases en cycles.il attribut à chaque branche un rôle est définit comment et quand vont interagir. En plus des branches de fonction il créer des branches des releases

Utilise la branche « develop » comme branche parent. Une fois la fonction est achevé doit être mergé vers « develop ».ces branches doivent jamais interagir avec master.

Ce workflow au lieu d’utilisé une seule branche master il utilise 2 branches pour enregistré l’historique du projet.

Master contient du code stable c’est l’historique officiel des releases contient ce qui en production

Branches fonction peut être instable

Une fois que develop aura acquis assez de fonctionnalités en vue d'une livraison (ou qu'une date de livraison prédéfinie approche), faites un fork d'une branche de version (release) à partir de develop. La création de cette branche marque le début du cycle de livraison suivant

**Hoytfix branches** : Les branches de maintenance. Pour faire de patch à partir de la branche master à la fin elle est mergé vers master et develop. Il est recommandé de consacrer une ligne de développement aux corrections de bugs

**Etapes :**

1. Créer la branche « develop » en locale et la pusher.
2. La mettre comme branches par défaut (pull request vers develop)
3. Créer une branche fonction modifié et pusher vers origin
4. Demander pull request dans github
5. Demandé des changements dans la fonction
6. Squash and merge pour concaténé toutes les commit faite dans branche fonction en une seule pour faciliter le retour en arrière et supprimé la fonction dans develop
7. Supprimé la branche fonction après merge
8. En locale changer vers branch develop et faire git pull
9. Une fois satisfait de develop créer une branche release/1.0 (télécharger en zip ou bien pusher dans un autre repo distant hiruku)
10. Dans le cas des erreurs en fait des commits (corrige)
11. Merger release/1.0 vers master avec pull request (merge pull request pas squash)
12. Merger release/1.0 vers develop en cas de changement dans release avec pull request (pull request)
13. Supprimer la branche release/1.0
14. Créer une release v1.0 avec le tag v1.0
15. Suicher et Mettre a jour master dans la dépôt locale avec remote
16. Créer hotfix branche depuis master et fixer l’erreur et push vers remote
17. Vers un pull request vers master (Squash and merge)
18. 19 puisque master a changer on va créer un nouveau release v1.0.1
19. Créer une pull request de hotfix vers develop

Le flux global de Gitflow est le suivant :

1. Une branche develop est créée à partir de main.
2. Une branche release est créée à partir de la branche develop
3. Des branches feature sont créées à partir de la branche develop
4. Lorsqu'une fonctionnalité est terminée, elle est mergée dans la branche develop.
5. Lorsque la branche release est terminée, elle est mergée dans la develop et dans main.
6. Si un problème est détecté dans la branche main, une branche hotfix est créée à partir de main.
7. Une fois la branche hotfix terminée, elle est mergée dans develop et dans main.