1, Retourner à une version ultérieure de fichier ctrl +z; ctrl +z; ctrl +z; ctrl +z; ctrl +z;ctrl +z;ctrl +z

2, donner un nom différents au fichier

3, partager les modifications avec nos collègues.

4. travailler avec plusieurs pcs sur le même projet (travail, maison)

3. c’est une application a cessé de fonctionner comment savoir la cause et réparer si on n’a pas le csv. Avec le CSV ont peur retourné à un état stable

4, si l’application à un bug, chaque développer créer un autre branche (au lieu de cloner et le mettre quelque part) règle le bug puis chef de projet tester la solution si elle non il va être adopté et merger la branche principale

Comme vous savez un produit logiciel et développé par un équipe et que chacun peut être situé dans des endroits différents et chacun contribue à un type spécifique de fonctionnalité en modifions le code source (ajout /supprimer).

Un VCS permet au équipes de développeur à communiqué et gérer efficacement les modifications apportées au code source ainsi les informations telles qui a fait et qu’elles modifications ont été apportées

Chaque branche est créer pour l’utilisateur les modifications serons pas mergé à la branche principale jusqu’à que le code sois analysé et validé, ce qui implique en gagne en productivité.

-Accélère la livraison de produit

Exemple is **Git, Helix core, Microsoft TFS ////////////fin slide 1**

**Locale :** Simple, assure le suivit des fichiers localement,

**SCV centralisé :** CSV local on ne peut pas l’utilisé si on veut collaborer avec d’autre développeurs

Pour cela création de la version centralisé seul serveur contient les versions des fichiers. Chaque commit affecte le serveur centrale et visible ou autre développeur,

1. Chacun de système à l’information sur ce que font les autres font dans le projet,

2. L’administrateur a le contrôle des autres développeurs

Le point faible quand le serveur tombe en panne ou bien le HDD est corrompue on perd le travail, pour cela la version distribué a été développé,

**2,SCV Distribue** (comme Git, Mercurial, Bazaar ou Darcs) :

Il y’à une copie de repo (projet) pour chaque développer (contient l’historiques des branches) (plusieurs versions du projet indépendants chacun avec son propre historique).il peut de déconnecter et travailler localement.

Nos commit n’affecte pas le serveur distant (invisible ou autre développeur). Puis push le changement vers le repot distant.si on pert le disque de dur on le repo distant il y’. a toujours un copie de repo avec un autre développeur .inconvénient gestion et utilisation compliqué**//fin2**

**Hist:** Lancer par linus torvalds pour hébergé linux à cause de litige avec BitKeeper

Def:Rapide, open source supporté par tous les système d’exploitation

**Fonctionnement :**

**1. Repo locale :** contient les métadonnées et la base de données des objets du projet

**2. La zone de transit/d’index** : fichier binairecontenant des informations à propos de ce qui sera pris en compte lors de la prochaine commit (espace ou en peut éditer notre changement avant de l’intégrer dans le nouveau commit (commit par sujet) et les stocké dans le dépôt git). On fait des commit par sujet pour bien comprendre les modifications au futur

**3.Work tree , workspace, reperoire de travail :** est une copie personnel de tous les fichiers du projet(versioné et non versionné). On peut modifier sans affecter le travail des autres

**Conflits** : il intègre le travail qui a été fait simultanément par différents développeur. Dans des cas rare des conflits d’édition survient par deux personnes dans une même ligne de même fichier, une assistance humain est demandé par ce SCV pour réglé le conflit///fin3

Git utilise SSH pour communiqué avec dépôt distant (public key, private key)

- Les messages sans crypter par la clé publique et décrypté par la clé privée (même si on dispose de la clé publique on ne peut rien faire)

Dans windows les clés se trouve dans le home directory dans un répertoire caché

//fin5

**Contenu de .git**

1. HEAD : un pointeur vers le dernier commit

**Avec git clone :** remote va pointer à l’url d’où il été cloné

Afficher une variable git config --get user.name

Git status affiche (non track, indexé, non indexé) utilisé gs –s (pour montrer les fichiers qui sont dans stagged arrea & et les fichiers dans work directory)

Git checkout : on ne peut pas le faire avec un fichier supprimé, pour supprimer l’indexation d’une suppression on fait git restore –stagged file//fin 7

**Git clean** : opère sur les fichiers et répertoire untracké (non versionné)

**Les fichiers non versionnés (untracked file) :** sont des fichiers crées dans le WD mais qui ne sont pas ajouté à l’index avec la commande « git add »

Git est configuré pour que la commande clean doit être exécuté avec l’option force (mécanisme de sécurité) elle n’est pas réversible. Elle fait une suppression physique

**Git add :** met à jour l’index par le nouveau contenue dans le WD elle peut s’exécuté plusieurs fois avant le commit. Cette commande n’ajoute pas les fichiers ignorés.

Importante pour git commit (étape pour sélectionner ce qui va être commité) marque les fichiers à inclure dans le commit suivant.la commande n’affecte pas le repo locale.

Cette commande et exécuté à chaque fois qu’on modifie un fichier

Les changements ne sont pas effectué dans le Dépôt locale jusqu’à l’exécution de **« git commit** ».

* Git add \* (ignore celui qui commence avec ‘ **.**‘)
* git add –u(n’ajoute pas les fichiers non suivit)

Pour défaire cette action on utilise « **git reset** »

**Git rm** : qu’on veut que git ne plus suive un fichier (exemple ajouté par erreur).

Utilisée pour supprimer les fichiers dans l’index et dans le répertoire de travail (il faut commité pour supprimer).

Pas d’option pour supprimé les fichiers dans le répertoire de travail uniquement. On ne peut pas supprimer les fichiers untracker avec git rm. C’est comme l’inverse de « git add ».

Pour utiliser git rm sur un fichier il doit être identique à la version qui est dans HEAD, pour éviter de supprimé des fichiers en progression (mécanisme de sécurité), mais on peut le forcé avec l’option -f

**Option « –cached »** : le supprime dans l’index (dépôt locale) mais pas dans le système de fichier

Elle réversible avec la commande « git reset HEAD » et « git restore filename »

**Git status :** affiche l’état de répertoire de travail et de l’index (staging area).elle nous permet de savoir ce qui a été indexé et celui qu’il n’est pas encore. Elle affiche la différence entre index et le commit HEAD courante (les changements qui vont être inclut dans **git commit**) et entre WD et l’index(les changements qui vont inclut dans **git add**) (non track, indexé, non indexé)

**Option –s** : short format

**git commit –m ’fonction’ :** une fois on a fini le travail ont fait un nouveau commit qui va contenir le contenu actuel de l’index et le message qui va décrire le commit (capture un instantané des changements actuellement stagés du projet). Ce commit va être l’enfant directe de HEAD puis la branche va être mis à jour pour pointé vers ce commit.

Les commits servent à suivre nos progrès et changement pendant que nous travaillons. Est un pont vers lequel on peut se retourné si on trouve un bug.

On peut revenir à la version précédant du fichier uniquement quand il est dans l’index avec (git restore --staged <file>) mais pas dans le WD.

Est un point dans notre projet ou on peut retourner si il yà un bug.

**Option –amend** : pour modifié le commit précédant au lieu de créer un autre.

**Option -a** : on ne veut pas passer par la zone intermédiaire car c’est un petit changement (gagné du temps) s’applique aux fichiers déjà indexéca

Le message de commit doit être clair pour que tout le monde sache ce qui à été changé dans ce commit

**-Le message de commit peut contenir les infos suivantes :**

1. Ce qui est nouveau 2.Les raisons de changements

**Git Diff** (**Analyse de code**) : dés fois on a besoin de savoir ce qui va être comité para port au dernier commit. Dés fois on a besoin de savoir la différence entre deux branches, deux commits, deux fichiers.et dés fois on a besoin d’identifier le source d’erreur dans le code.

Différences entre 2 source (branches, commit, file) afin de mieux suivre ou vérifier les modifications.

**Git diff –cached** : compare l’index par rapport au commit pointé par HEAD (stagged area and local repot)

**git diff HEAD :** compare le HEAD par rapport au répertoire de traval

**EXEMPLE1 :**

1. 1.echo " Hello World!" > example.txt
2. Git add .
3. Git commit –m ’m’
4. echo "Hello from Educative" > example.txt
5. git diff HEAD example.txt

Git Stash : prend les changements non commité (indexé ou bien non indexé) les sauvegarde (planque) pour le reprendre après (prend pas en charge les fichiers qui non jamais été indexé).

Appelé stash sans argument si comme si on fait git stash –push

**Option –u** :pour prendre en charge les fichiers non tracké

**Git stash pop :** reprendre le code caché et le supprime dans stash

**Git stash applay :** reprendre le code et le garde dans stash (utile lorsque on veut appliquer le même code stashé dans plusieurs branches)

**Git ignore (.gitignore) :** git voit chacun de nous fichiers dans notre répertoire de travail en trois état suivante

1. Fichier tracké (suivit) est un fichier qui a été déjà indexé ou bien commité
2. Fichier non tracké est un fichier qui n’a pas été déjà indexé ou comité
3. Fichier ignoré est fichier que nous avons demandé à git d’ignoré

Parmi ces fichiers qu’on veut ignorer il y’à les fichiers log, fichiers temporaire (temp, .tmp), la cache de dépendance (/vendor), code compilé, répertoire de sortie de build), fichiers généré à l’exécution (runtime).

Les fichiers et répertoires à ignorer sont listé dans le fichier « .gitignore » placé dans la racine de notre répertoire de travail

Pour dir à git de ne pas ignorer le fichier/dossier on utilise point d’exclamation

**Git log :** le but de CVS est d’enregistré des modifications apporté au code.il nous permet de retourné dans l’historique du projet pour savoir qui a fait quoi. Déterminé les bugs et annulé les modifications qui ont causé le problème.la commande log nous permet de navigué à travers l’historique du projet et le lire. Git log nous affiche les commits.la sortie de la commande peut être personnalisé selon un formatage (forme oneline graph) et selon le contenu qu’elles sont les commits qui vont être affichés (n dernière commits les commits relatifs à un fichier)

**Git blame :** affiche les métadonnées du l’auteur du commit, utilisé pour explorer l’historique d’un code spécifique et répondre aux questions quoi et comment et pourquoi un code a été ajouté à l’historique

Opère sur un seule fichier affiche (l’auteur, date, numéro ligne, contenue ligne)

Examine le fichier ligne par ligne pour savoir pour chaque ligne quand est les dernières modifications et qui est l’auteur de ces modification (sortie de la commande personnalisé selon les options).investigation quand et comment le bug a été introduit et contacté l’auteur pour des explications sur un morceau de code.

Le rapport ne vous dit rien sur les lignes qui ont été supprimées ou remplacées ///fin10

**Git tag :**

Un autre mécanisme pour créer des snapshot de dépôt git. Est utilisé pour créer des numéros de versions avec sens correspondant aux cycles de production de logiciels **v<major>.< Minor>.< patch>** , utilisé pour référencé les commits pendant le workflow de production.

* **Incrémente major** quand on introduit des modifications qui seront incompatible avec la version précédant
* **Incremente minor** quand cette version est compatible avec version precedante
* **Incremente patch** qu’on fix un bug ou un patch

--tag d'annotation

**--lightweight tag**

tag utilisé comme les branches pour pointé un commit spécifique (tag), on n’utilise pas les options on don juste le nom

--**tag d'annotation :**

Peut être utilisé pour annotation stocké comme ensemble d'objets dans la base de donnée de Git, (nom tag, email, date, message, signature GPG)

pour la créer on utilise -a

-m spécifie un message

Elle peut avoir un message l'auteur, et date diffèrent du commit sur lequel elle point (décrire un release sans créer un commit de release)

**tagé un commit specifique** (On peut tagé un commit même on la dépasse)

* ***git tag <tag\_name> <commit\_sha>***

**git push** Transfer pas les tags au serveur distant, pour transférer git push "tagname"

**git tag –n : affiche les tags avec les messages**

--supprimé un tag

**git tag -d tagname**

**Git revert (mal comprise) :** s’applique uniquement sur des commits ne supprime pas le commit dans l’historique de projet (essentiel pour l’intégrité de l’historique des révisions et une bonne collaboration) mail il le supprime de répertoire de travail. On détecte une erreur introduit par un commit au lieu de corrigé manuellement on peut faire git reverse pour corriger authomatiquement.il prend le commit en question et inverse les changements (file création =>file suppression, text added=>text removed) puis créer un nouveau commit de reverse.

**Git reset**

**///Quand on veut fixer un commit public on utilise revert on n’utilise pas reset**

On veut déplacer le pointeur la branche (réinitialisé le dépôt) vers une commit postérieure, annulé les commit qui suis cette commit mas le répertoire de travail reste le mêletout d'abord on localise la commit ou on a envie de retourné (git log).

Il est tolérable d'apporté ce type de modification à notre dépôt locale. Mais à éviter de faire ces changement et réécrire l'historique de dépôt distant surtout si d'autre développeurs travail dessus.

Même si les commits n'apparaissent plus dans le journal, elles ne sont pas supprimé du Git (devient orphelin).avec hard même je reviens au dernier commit mais je perds les modifications. Utilise pour unstage des changements rien avoir avec le commit prochain. Annulé des changements fait dans une expérimentation locale

Git reset –soft **HEAD^ :** supprime le dernier commit et retourne a état staged

Git reset **HEAD^:** mixed par défaut supprime le dernier commit et retourne à l’état unsatagé

Git reset –hard **HEAD^: supprime le dernier commit et supprime les modification\\fin11**

**Git checkout HASH commit**

Va charger l’arbre de commit dans le répertoire de travail.

Ecrire dans le fichier index (stagged area)

Mettre à jour le fichier HEAD à cette commit (detached head). ////Fin9

**Pull = fetch +merge :** apportera les changements**(fetch**) et merge les changement de la branches distante.

**fetch:** vous pouvez être assuré : fetch ne manipulera, ne détruira ou ne bousillera jamais quoi que ce soit.

Fetch est inoffensive on doit faire merge pour répurqueté les changements dans le répertoire de travail

Pour répercuté les changements dans l’espace de travail on exécute la commande :

***git merge origin/branchname*** Ou bien ***git rebase origin branchname***

Pour voir les changements on tape «***git chekcout origin/branchname***»

git diff develop origin/develop

git log develop..origin/develop (voir les commit supplémentaire)

**Pull** : modifié le HEAD ont plus de téléchargement il intègre les changements dans l’espace de travail (peut générer des conflits) (pull doit être utilisé avec une copie local propre son changement un commité)

**Rebase** :

Applique chaque commit de manière individuel pour avoir un historique propre (créer pas de commit on ne sait pas quand est ce que le merge a été effectué)

O, peut pas faire de suivit (à ne jamais utilisé dans un dépôt public). —skip pour retour en arrière

**merge :**

Par default fast farword quand la branche de base n’a pas été modifié pas de création d’un nouveau commit.

1. applique toutes les commit de la branche mergé dans un seule commit ajouté à la fin.

2. Ne change pas l’historique du projet il ajoute juste un commit

**Rebase :**

1. Trouve toutes les ommits des deux branches et les applique une à une

2. Change l’historique du projet il ne crée pas un nouveau commit pour mergé///14

**Définition :** est une recommandation sur la façon d’utiliser Git Pour en profiter pleinement et être productive pour cela on doit le maitrisé.

Git nous donne une liberté a son utilisation il n’impose pas une méthode de travail (dépend de nous et de notre connaissance de la technologie) et on doit le gère d’une manière ou d’une autre c’est là où il apparut la notion de workflow (flux de travail) qui décrit ce que nous devons faire et dans quel ordre pour que le travail du l’équipe avec git se passe bien de manière cohérente et facile à.

Quel stratégie adopté pour organisé le workflow de collaboration pour livre rapidement. Permet de détermine les rôles et les responsabilités

Le nommage des branches dépend de l’équipe

-**Branches de longue durée :** utile pour les grands et complexe projets la branche master et la branche develop ont une existence long jusqu’à la fin de projet le code plus au moins stable les commit sont moins fréquentes (une fois qu’on a bien testé).

**-**on fait jamais de commit à la branche master (branche de production) l’ajout se fait par l’intégration (merge)

**Devlop :** on créer une branche pérenne « develop » à coté de master « place sur ou introduire du code qui peut casser notre projet» .on a besoin d’une stratégie de teste pour s’assurer de ne pas introduire de bug à master l’hors des merge (inconvénient on ne peut pas développer plusieurs fonctionnalité en parallèle- il y’à qu’un seule ou deux développer peut travailler activement sur le projet, supprimer ou restore une fonctionnalité est un challenge)

**Branches de courte durée :** utile pour les petits et grands projets généralement créer pour une fonctionnalité particulière. Dans un projet on créer fréquemment ce genre de branche on travaille dessus et on les merge puis on les supprime. **//fin15**

**1. Workflow Centralise :** si comme si on travaille avec un CSV centraliser. Stratégie 0 Branche, tous les développeurs fond des changements sur la branche principale (branche par défaut) puis la mergé une fois terminé. Les cas de figure où il est utilisé

1. Le code n’a pas besoin d’être contrôlé (on travail seule, une petite équipe chacun spécialisé dans un Domain (back-end, front -end), qu’on veut livrer rapidement parce que La revue de code prend du temps).répondue dans la collaboration entre chercheur (demain scientifique)

-efficace pour une petite équipe (qui communique bien) parce qu’il ne modifie pas la même partie du code simultanément.

Avent d’envoyé les changements il faut faire un fetch et un rebase pour éviter les conflits (historique linéaire).

**Etapes suivez par le développeur :**

1. cloner

2. faire des commits dans le branche principale locale

3. fetch la branche principale distante

4. merger la branche principale distante (rebase ces changements au top qui peut générer des conflits)

5. pusher la branche principale local

**inconvénient :**

-utilise pas le potentiel donné par git

-difficile de trouvé le moment stable pour faire un release de changement

- beaucoup de conflits, développé d’autre fonctionnalité et réglé les problèmes

-difficile de supprimé et de restauré des fonctionnaliste

-Perte de temps dans le réglage de conflit au lieu le développement

////

**Workflow de branche par fonctionnalité :**

Extension de précédant, branche master est l’historique officiel du projet contient code stable et déployable.il y’à toujours un dépôt centralisé .mais au lieu de pusher dans la branche master, Le développeur créer une nouvelle branche (branche fonctionnelle) à chaque fois qu’ils commencent de créer une nouvelle fonctionnalité. La branche fonctionnelle doit avoir un nom descriptif.

-Une fois la fonctionne est terminée le développeur suit les étapes ci-dessous :

**I) methode1**

1. Cloner le dépôt distant

2. Créer une branche de fonction depuis la branche principale

3. Faire les commits dans feature branche.

4. Pull la branche principale distante

5. Mergé la branche de fonction on vers la branche principale

6. Pusher la branche principale

7. supprimé la branche fonctionnel

II) **Méthode 2(PR)**

Avant que la branche soit mergé à la branche master doit être vérifié et contrôlé qu’il ne contient pas d’erreur. Le développeur réer un pull request.puis les autres membres revoit le code et laisse des commentaires une fois c’est bon il mergent.

Les étapes comme c’est dessus.

1. Cloner le dépôt distant

2. Créer une branche de fonction depuis la branche principale

3. Faire les commits dans feature branche.et pusher le travail dans la branche de même nom dans le serveur

4. Si on a besoin d’aide ou de feedback ou qu’on pense que la branche est prête pour mergé on ouvre un pull request.

5. Après que le code soit revue et signé par les reviewer on peut la mergé vers master.

6. Une fois mergé à master, la fonction est prête pour déploiement.

1. **Avantage :**

-Partagé une fonction sans touché au code officiel.

-Promouvoir la collaboration avec des pull request et merge de revue

-permet de travailler sans perturbé le code de base et la branche principale ne contient pas d’erreur.

-idéal quand on a besoin de maintenir une seule version dans la production.

Inconvenant :

1. Le code de production peut être instable fréquemment.

2. Non adéquate quand on a besoin de plans de release.

3. Résout rien en terme déploiement, environnements, et release.

4. n’est pas recommandé quand à besoin de multiple versions on production.///fin16

**Gitflow :** modèle de branchement de projet git, chaque branche à un rôle spécifique. Comment et quand il faut que les branches vont interagir. Ce qui fait une grande aide dans l’implémentation de processus d’intégration continue et le processus de livraison continue.

**Main** : créer au début du projet et maintenue tout au long du processus de développent. Tagé en plusieurs commits dans le but de montrer les différentes versions du code. Le code et stable est un avantage en cas de problème dans les autre branches. Souvent utilisé comme branche d’intégration pour la branche develop.

**La branche develop :** est souvent utilisé comme branche d’intégration pour les branches de fonction

**Branche de fonction :** et bifurqué depuis **develop (le parent).**quant on termine de implémentation de la fonction on l’intègre à **develop**. Les branches fonction n’interacte jamais avec le master//fin17

**Avantages**

Nous permet de vérifie chaque ligne de code avant qu’elle soit intégré dans les versions//fin18

1. **Branche release :** une fois la branche develop contient suffisamment de fonctionnalités une branche release est bifurque de develop. une fois créer pour un cycle de release aucune fonction ne peut être ajouté à partir de ce point juste la tache de fixation de bug et d’autre taches relatives au release peuvent ajouter à cette branche une fois le release est fait on doit mergé ce release à la branche master avec un tag d’une nouvelle versions. aussi il faut mergé le release vers la branche develop car il peut avoir des changements (progressé) qui sont pas dans develop. ces branches permet de devisé l’équipe une partie travail sur le release et l’autre pour développer des nouvelle fonctions pour la prochaine release
2. **Hotfix :** bifurqué depuis master utilisé pour corriger rapidement la release en production. À la fin de correctif il doit être mergé vers « ***main*** » et « **Develop**» le merge à « Develop » est critique pour s’assurer que le correctif persiste dans le prochain release. Avoir une line de développement pour la correction des bugs nous évitent l’interruption de reste du workflow et attendre la prochaine release//fin19

Deux branches distinctes modifie le me fichier, Les conflits sont couteux et prend de temps. Les conflits affecte la personne qui à fait le merge, On peut toujours retourner à l’état avent le merge avec la commande « git merge –abort »//fin22

**Pull request** : Interface conviviale pour discuter les changements proposé avant de les intégrés.

Manière d’informé les autres membres qu’il à finit le développement de la fonctionnalité

PR : On lui demande de faire un pull de la branche depuis notre repo

Quand on fait un fork il apparait dans la liste des dépôts dans github et on peut le cloné dans la machine locale et modifie puis pusher dans le fork dans github

Pull request offre un forum pour discuter des changements proposés avant d’être intégré dans la branche partagée

Les 5 fonctionnalités principales qui garantissent que les pull request vont fonctionné de maniéré efficace

1. Ajouter un reviewer

2. Ajouter des Commentaires sur toutes les choses (dans un fichier à part ou bien dans le code)

Reviewer peut faire des suggestions ou bien félicité le membre de l’équipe de son pièce de code brillant (commentaire sur un pull request, sur un commit, sur une ligne, sur un fichier)

3. Revue itérative://fin23

Un pull request créer un boucle de feedback (on écrit un code, l’équipe revoie le code, on incorpore des changements) puis l’équipe approuve, puis merge

Le viewer vois juste le dernier changement dans le pull request//fin23

Quand on fait un fork il apparait dans la liste des dépôts dans github et on peut le cloné dans la machine locale et modifie puis pusher dans le fork dans github

Deux solutions

Solution 2

1. avec github le bouton (sync fork)

2. git fetch

3. git merge origin/master//fin26