**Git**

1. **Definition**
2. **Definition**

Git est un système de contrôle de version distribué qui permet de suivre les modifications apportées à des fichiers et des dossiers au fil du temps.

Le contrôle de version consiste à enregistrer l'historique des modifications apportées à un ensemble de fichiers au fur et à mesure que des développeurs travaillent sur un projet. Git permet aux développeurs de collaborer efficacement en suivant les modifications, en fusionnant des branches de code et en gérant les conflits lorsqu'ils surviennent.

Les caractéristiques clés de Git comprennent :

1. **Distribution décentralisée :** Chaque contributeur possède une copie complète de l'historique du projet, ce qui permet de travailler en mode déconnecté et de fusionner les modifications ultérieurement.
2. **Branches :** Git permet de créer des branches indépendantes du code source, ce qui facilite le développement parallèle et la création de nouvelles fonctionnalités sans perturber le code principal.
3. **Commits :** Les modifications apportées au code sont enregistrées sous forme de "commits", accompagnés de messages expliquant les modifications apportées.
4. **Fusion et fusion résolue :** Git permet de fusionner différentes branches de code pour intégrer des modifications dans le code principal. En cas de conflits entre les modifications, Git offre des outils pour résoudre ces conflits manuellement.
5. **Historique complet :** L'historique des commits, y compris les messages, les auteurs et les modifications, est conservé, ce qui facilite la compréhension de l'évolution du projet au fil du temps.
6. **Suivi des modifications :** Git suit les modifications apportées aux fichiers, ce qui permet de voir comment le code a évolué et de revenir à des versions précédentes si nécessaire.
7. **Plateformes d'hébergement :** Des plateformes telles que GitHub, GitLab et Bitbucket offrent des services d'hébergement de dépôts Git, ce qui facilite la collaboration et le partage du code entre les développeurs.

En résumé, Git est un outil essentiel pour la gestion efficace du code source dans les projets de développement de logiciels, car il permet aux équipes de travailler ensemble de manière fluide tout en préservant l'intégrité de l'historique et en facilitant le suivi des modifications.

1. **La différence entre git et GitHub**

Git est le système de contrôle de version, tandis que GitHub est une plateforme en ligne basée sur Git, qui facilite l'hébergement, la collaboration et la gestion de projets de développement de logiciels.

1. **La difference entre le repertoire du travail et l'index et le Référentiel Git**

Bien sûr, je vais expliquer en détail les trois concepts clés de Git : le répertoire de travail, l'index (zone de préparation) et le référentiel Git.

1. **Répertoire de travail** : Le répertoire de travail, également appelé "working directory" en anglais, est l'endroit où vous travaillez directement avec vos fichiers. C'est là que vous effectuez vos modifications, ajoutez de nouveaux fichiers ou supprimez des fichiers existants. Le répertoire de travail contient la version actuelle des fichiers tels qu'ils apparaissent sur votre disque dur. Lorsque vous effectuez des modifications dans le répertoire de travail, Git ne suit pas automatiquement ces modifications pour les inclure dans les commits. Vous devez explicitement ajouter les modifications à l'index pour qu'elles soient prises en compte dans le prochain commit.
2. **Index (Zone de préparation)** : L'index, également appelé "staging area", est une étape intermédiaire entre le répertoire de travail et le référentiel Git. L'index permet de contrôler quelles modifications seront incluses dans le prochain commit. Lorsque vous effectuez des modifications dans le répertoire de travail, elles ne sont pas directement ajoutées aux commits. Vous devez utiliser la commande **git add** pour ajouter les modifications au niveau de l'index. En ajoutant les modifications à l'index, vous préparez ces modifications pour le prochain commit. Cela vous permet de sélectionner spécifiquement les modifications que vous souhaitez inclure dans un commit.
3. **Référentiel Git** : Le référentiel Git, souvent appelé "repository" ou "repo", est la base de données où Git stocke l'historique complet de votre projet, y compris tous les commits, les branches, les tags et les autres références. Chaque commit dans le référentiel Git enregistre un instantané de l'état des fichiers à un moment donné, en se basant sur le contenu de l'index à ce moment-là. Le référentiel Git conserve l'intégralité de l'historique de votre projet, ce qui permet de revenir en arrière dans le temps, de créer des branches pour des développements parallèles et de collaborer efficacement avec d'autres développeurs.

En résumé, le répertoire de travail est l'endroit où vous modifiez vos fichiers, l'index est une étape intermédiaire où vous préparez les modifications pour les commits, et le référentiel Git est la base de données qui enregistre l'historique complet de votre projet. L'utilisation de ces trois concepts vous permet de gérer vos modifications de manière contrôlée, de créer des commits cohérents et de conserver un historique fiable de votre projet.

1. **Git Commandes**
2. **Commande Basic**
3. **git init :** Initialise un nouveau dépôt Git dans le répertoire actuel.
4. **git status :** Affiche l'état actuel du dépôt, montrant les fichiers modifiés, ajoutés ou non suivis.
5. **git add <fichier(s)> :** Ajoute les modifications apportées aux fichiers spécifiés à l'index (staging area), en les préparant pour le prochain commit.
6. **git push :** Envoie vos commits locaux vers le dépôt distant.
7. **git pull :** Récupère les dernières modifications depuis le dépôt distant et les fusionne dans la branche actuelle.
8. **git clone <URL> :** Clone un dépôt Git distant sur votre ordinateur local.
9. **Importe et crée un projet**

**git init** et **git clone** sont deux commandes Git utilisées à des étapes différentes pour créer des dépôts Git. Voici comment elles fonctionnent :

1. **git init :** La commande **git init** est utilisée pour créer un nouveau dépôt Git à partir de zéro dans un répertoire existant. Cela signifie que vous commencez avec un répertoire qui contient déjà des fichiers, puis vous exécutez **git init** pour commencer à suivre les modifications de ces fichiers à l'aide de Git.

Voici comment utiliser **git init** :

* 1. Naviguez dans le répertoire que vous souhaitez transformer en dépôt Git.
  2. Exécutez la commande **git init**.
  3. Cela crée un nouveau sous-répertoire nommé **.git** dans le répertoire actuel, qui contient toutes les informations nécessaires pour le suivi des modifications avec Git.
  4. Vous pouvez maintenant commencer à ajouter des fichiers avec **git add** et créer des commits avec **git commit**.

Cette commande est utilisée lorsque vous créez un nouveau projet ou que vous voulez commencer à suivre les modifications de code existant.

1. **git clone :** La commande **git clone** est utilisée pour copier (cloner) un dépôt Git distant complet sur votre ordinateur local. Cela vous permet d'obtenir une copie complète du dépôt, y compris tout l'historique des commits, les branches et les fichiers.

Voici comment utiliser **git clone** :

* 1. Trouvez l'URL du dépôt Git distant que vous souhaitez cloner (par exemple, sur GitHub).
  2. Exécutez la commande **git clone <URL>** dans le répertoire où vous souhaitez que le dépôt soit cloné.
  3. Git copie tout le contenu du dépôt distant dans un nouveau répertoire local.
  4. Vous avez maintenant une copie du dépôt distant sur votre ordinateur, et vous pouvez commencer à travailler avec les fichiers et les branches localement.

Cette commande est utilisée lorsque vous souhaitez collaborer sur un projet existant ou obtenir une copie d'un projet distant pour contribuer ou travailler dessus.

En résumé, **git init** est utilisé pour créer un nouveau dépôt Git local à partir de zéro, tandis que **git clone** est utilisé pour copier un dépôt Git distant complet sur votre ordinateur local.

1. **Partager et mettre à jour des projets en utilisant Git**
   1. **Git fetch et git pull**

**Une image contenant texte, capture d’écran, Police, Rectangle

Description générée automatiquement**

**git fetch** et **git pull** sont deux commandes de Git utilisées pour récupérer des modifications à partir d'un dépôt distant, mais elles fonctionnent de manière légèrement différente. Voici la différence entre les deux :

1. **git fetch :** La commande **git fetch** est utilisée pour récupérer les modifications à partir d'un dépôt distant vers votre dépôt local. Cependant, **git fetch** ne fusionne pas automatiquement les modifications dans votre branche locale. Au lieu de cela, il met à jour les références locales (comme les branches distantes) pour refléter l'état actuel du dépôt distant.

Utilisation de **git fetch** :

**git fetch origin**

Cette commande télécharge les modifications du dépôt distant (par exemple, **origin**) vers vos références locales, mais ne modifie pas votre branche actuelle. Vous devrez utiliser d'autres commandes, telles que **git merge** ou **git rebase**, pour appliquer les modifications récupérées dans votre branche locale.

**git pull :** La commande **git pull** est un raccourci qui combine **git fetch** et une opération de fusion automatique (**git merge**). Elle récupère les modifications à partir du dépôt distant et les fusionne automatiquement dans votre branche locale.

Utilisation de **git pull** :

**git pull origin <nom\_de\_votre\_branche>**

Cette commande récupère les modifications du dépôt distant (par exemple, **origin**) et les fusionne dans votre branche locale spécifiée. Elle est utile lorsque vous souhaitez mettre à jour votre branche locale avec les dernières modifications du dépôt distant et que vous voulez fusionner ces modifications immédiatement.

En résumé, **git fetch** récupère les modifications du dépôt distant vers vos références locales sans les fusionner, tandis que **git pull** fait cela et fusionne automatiquement les modifications dans votre branche locale.

* 1. **Git push**

La commande **git push** est utilisée pour envoyer vos commits locaux vers le dépôt distant. Elle met à jour le dépôt distant avec vos modifications. Cependant, vous ne pouvez pousser que sur des branches pour lesquelles vous avez les droits d'accès en écriture. Le push met à jour le dépôt distant avec vos derniers commits.

* 1. **Git remote**

1. **Branch et le Merge**
2. **Branch**

La commande **git branch** est utilisée pour interagir avec les branches dans un dépôt Git. Une branche dans Git est simplement une ligne de développement qui permet de travailler sur des modifications distinctes en parallèle sans perturber le code principal. Voici quelques utilisations courantes de la commande **git branch**

1. **Afficher les branches existantes :** Pour voir toutes les branches existantes dans votre dépôt, vous pouvez simplement exécuter la commande **git branch** sans aucun argument. Par exemple : **git branch**

Cette commande affichera une liste des branches locales présentes dans votre dépôt, et un astérisque (\*) indiquera la branche sur laquelle vous êtes actuellement.

1. **Créer une nouvelle branche :** Pour créer une nouvelle branche, vous pouvez utiliser la commande suivante : **git branch nom-de-branche**

Cela crée une nouvelle branche à partir de la branche actuelle.

1. **Changer de branche :** Pour basculer vers une branche spécifique, utilisez la commande **git checkout** suivie du nom de la branche : **git checkout nom-de-branche**

À partir de Git version 2.23, vous pouvez également utiliser la commande **git switch** pour basculer entre les branches : **git switch nom-de-branche**

1. **Supprimer une branche :** Si vous avez fini de travailler sur une branche et que vous souhaitez la supprimer, vous pouvez utiliser la commande **-d** (ou **-D** pour forcer la suppression) avec **git branch** : **git branch -d nom-de-branche**

Assurez-vous de ne pas supprimer une branche importante sans avoir fusionné ou sauvegardé vos modifications si nécessaire.

1. **Renommer une branche :** Pour renommer une branche, vous devez la renommer localement et la supprimer à distance, puis la repousser avec le nouveau nom. Voici les étapes :

**git branch -m nouveau-nom** # Renomme la branche localement

**git push origin :ancien-nom** # Supprime la branche à distance

**git push origin nouveau-nom** # Pousse la branche renommée

Ces sont les utilisations les plus courantes de la commande **git branch**. Elle permet de gérer efficacement le flux de travail en équipe en travaillant sur différentes fonctionnalités ou corrections de bugs de manière isolée sur des branches distinctes.

1. **Git Merge**

La commande **git merge** est utilisée pour fusionner des modifications provenant d'une branche vers une autre. L'objectif principal est d'intégrer les changements d'une branche de développement dans une branche principale (généralement la branche **master** ou **main**) ou d'intégrer les modifications d'une branche de fonctionnalité dans la branche principale une fois que la fonctionnalité est terminée.

Voici comment utiliser la commande **git merge** :

1. **Assurez-vous d'être sur la branche de destination :** Avant de fusionner, assurez-vous d'être sur la branche dans laquelle vous souhaitez intégrer les modifications. Par exemple, si vous voulez fusionner les modifications de la branche **feature** dans la branche principale, utilisez **git checkout** pour vous rendre sur la branche principale : **git checkout nom-de-branche-principale**
2. **Effectuez la fusion :** Une fois sur la branche de destination, exécutez la commande **git merge** en spécifiant la branche que vous souhaitez fusionner (la branche source) : **git merge nom-de-branche-source**

Par exemple, pour fusionner la branche **feature** dans la branche principale : **git merge feature**

Git essaiera automatiquement de fusionner les modifications de la branche source dans la branche de destination. Si les modifications dans les deux branches ne créent pas de conflits, la fusion sera effectuée automatiquement. Si des conflits surviennent (lorsque Git ne peut pas déterminer comment fusionner automatiquement les modifications), vous devrez résoudre ces conflits manuellement en modifiant les fichiers en conflit et en utilisant la commande **git add** pour les marquer comme résolus.

1. **Validez la fusion :** Après avoir résolu les conflits et effectué les modifications nécessaires, utilisez la commande **git commit** pour créer un nouveau commit qui valide la fusion : **git commit -m "Merge branch 'nom-de-branche-source' into nom-de-branche-principale"**

Assurez-vous d'ajouter un message de commit significatif pour décrire la fusion que vous avez effectuée.

La commande **git merge** est essentielle pour intégrer les modifications de différentes branches dans un dépôt Git, ce qui permet de maintenir un historique de développement cohérent et organisé.

1. **Détache Le Head && référence Relative (^) et (~) && Annule les changements**
   1. **Détacher le Head**
      1. **Pourquoi Détacher le Head**

Détacher le HEAD en Git peut être utile dans certaines situations spécifiques, bien que cela ne soit pas recommandé pour un usage courant. Voici quelques raisons pour lesquelles vous pourriez vouloir détacher le HEAD :

1. **Inspection des commits :** En détachant le HEAD sur un commit spécifique, vous pouvez examiner de près ce commit et voir son contenu, les modifications qu'il contient et même effectuer des tests sur cet état précis du projet.
2. **Débogage :** Si vous devez effectuer un débogage approfondi sur un commit spécifique, vous pouvez détacher le HEAD pour tester différents scénarios sans risquer de modifier les branches principales.
3. **Cherry-picking :** Lorsque vous souhaitez appliquer un commit spécifique d'une branche à une autre, vous pouvez détacher le HEAD sur le commit souhaité, puis créer une nouvelle branche à partir de là pour intégrer ce commit.
4. **Révision de code :** En détachant le HEAD sur un commit, vous pouvez effectuer une revue de code plus précise en examinant les modifications apportées dans ce commit sans distraction.

Cependant, il est important de noter que travailler dans un état détaché peut être risqué et peut causer des problèmes si vous n'êtes pas prudent. Les commits que vous créez en état détaché ne seront pas attachés à une branche, ce qui peut les rendre difficilement accessibles ou perdus si vous ne les enregistrez pas ailleurs.

Si vous avez besoin de travailler sur un commit spécifique ou d'effectuer des opérations de vérification et de test, il est généralement recommandé de créer une nouvelle branche à partir de ce commit plutôt que de détacher le HEAD. Cela vous permettra de travailler de manière isolée tout en maintenant un lien clair avec le reste du développement.

* + 1. **Comment Détacher le HEAD**

Détacher le HEAD en Git signifie se placer dans un état où vous n'êtes pas sur une branche spécifique, mais plutôt sur un commit spécifique. Cela peut être utile pour inspecter ou travailler sur un commit spécifique sans être sur une branche. Voici comment détacher le HEAD en utilisant la ligne de commande Git :

1. **Trouver l'identifiant du commit :** Tout d'abord, vous devez connaître l'identifiant (hash) du commit vers lequel vous souhaitez détacher le HEAD. Vous pouvez l'obtenir à partir de la sortie de la commande **git log** ou d'autres moyens.
2. **Détacher le HEAD :** Utilisez la commande **git checkout** suivi de l'identifiant du commit pour détacher le HEAD :

**git checkout <identifiant\_du\_commit>**

Cela placera votre dépôt Git dans un état détaché où HEAD pointe directement vers le commit spécifié. Vous n'êtes pas sur une branche et ne pouvez pas faire de nouveaux commits sans créer une nouvelle branche.

Lorsque vous avez terminé avec l'état détaché, pour revenir à une branche normale, vous pouvez simplement utiliser **git checkout** suivi du nom de la branche :

**git checkout ma\_branche**

Assurez-vous de ne pas faire de nouvelles modifications importantes ou de créer de nouveaux commits dans un état détaché sans enregistrer les références appropriées.

* 1. **Référence Relative (^) et (~)**
     1. **Définition**

En Git, les références relatives **~** et **^** sont utilisées pour se déplacer dans l'historique des commits à partir d'un commit spécifique. Elles permettent de référencer des commits parents ou ancêtres d'un commit donné. Voici comment elles fonctionnent :

1. **~ :** Le symbole **~** suivi d'un nombre (ou sans nombre) est utilisé pour accéder aux commits ancêtres d'un commit donné. Par défaut, **~** représente le premier parent du commit. Si un nombre suit **~**, cela signifie que vous souhaitez vous déplacer d'un certain nombre de niveaux d'ancêtres.

Par exemple :

* + **HEAD~1** : Accède au parent du commit actuel (commit précédent).
  + **HEAD~2** : Accède au grand-parent du commit actuel (deux commits en arrière).
  + **abcd123~3** : Accède au 3ème ancêtre du commit avec l'identifiant **abcd123**.

1. **^ :** Le symbole **^** est utilisé pour référencer les parents d'un commit qui a plusieurs parents (comme dans le cas d'une fusion, appelée commit de fusion). Si un nombre suit **^**, cela permet de spécifier le parent spécifique lorsque le commit a plus de deux parents.

Par exemple :

* + **abcd123^** : Accède au premier parent du commit avec l'identifiant **abcd123**.
  + **efgh456^2** : Accède au deuxième parent du commit avec l'identifiant **efgh456** (dans le cas d'une fusion).

Ces références relatives sont particulièrement utiles pour se déplacer dans l'historique et référencer des commits spécifiques dans des opérations telles que la création de nouvelles branches, la révision de code et d'autres tâches de gestion des versions. N'oubliez pas que les références relatives sont basées sur la topologie de l'arbre des commits, il est donc important de comprendre comment fonctionne l'arbre des commits dans Git pour les utiliser correctement.

* + 1. **Example**

Bien sûr, voici des exemples concrets pour illustrer l'utilisation des références relatives **~** et **^** :

Supposons que nous ayons l'historique de commits suivant :

A - B - C - D - E - F

* **A**, **B**, **C**, **D**, **E** et **F** représentent les identifiants (hash) des commits.

1. Utilisation de **~** :
   * **HEAD~1** fait référence au commit précédent à **HEAD**, qui est le commit **E**.
   * **HEAD~2** fait référence au deuxième ancêtre avant **HEAD**, qui est le commit **D**.
   * **C~3** fait référence au troisième ancêtre avant le commit **C**, mais comme **C** a moins de trois ancêtres, cela n'a pas de sens dans ce cas.
2. Utilisation de **^** :
   * **B^** fait référence au premier parent du commit **B**, qui est le commit **A**.
   * **F^2** fait référence au deuxième parent du commit **F**, mais cela a du sens uniquement si **F** est un commit de fusion.

Supposons maintenant que nous ayons un historique de commits avec une fusion :

  G

/ \

A - B - F

\ /

  C - D - E

* **A**, **B**, **C**, **D**, **E**, **F** et **G** représentent les identifiants (hash) des commits.

1. Utilisation de **^** avec un commit de fusion :
   * **F^1** fait référence au premier parent du commit de fusion **F**, qui est le commit **B**.
   * **F^2** fait référence au deuxième parent du commit de fusion **F**, qui est le commit **C**.
   * **G^** fait référence au premier parent du commit **G**, qui est le commit **B**.

Ces exemples illustrent comment les références relatives **~** et **^** peuvent être utilisées pour se déplacer dans l'arbre des commits en fonction de la topologie de l'historique.

1. **Annuler les changements en utilisant git reset et git revert**
   1. **Git reset**

La commande **git reset** est utilisée pour déplacer la branche courante (HEAD) et/ou l'index (zone de préparation) vers un commit spécifique, ce qui peut avoir un effet sur l'historique du dépôt et sur l'état des fichiers dans le répertoire de travail. Il existe trois principaux modes de réinitialisation : **--soft**, **--mixed** (ou par défaut) et **--hard**.

1. **git reset --soft** : Lorsque vous utilisez cette option, Git déplace simplement la branche courante (HEAD) vers le commit spécifié sans modifier l'index ni le répertoire de travail. Cela signifie que les modifications de l'index et du répertoire de travail restent intactes et non confirmées. Cette option est souvent utilisée si vous voulez refaire un commit en utilisant les mêmes modifications. Vous pouvez alors réindexer les modifications et créer un nouveau commit.
2. **git reset --mixed (par défaut)** : C'est le mode par défaut si vous n'indiquez pas explicitement l'option. Lorsque vous utilisez **--mixed**, Git déplace la branche courante (HEAD) vers le commit spécifié et réinitialise l'index pour correspondre à l'état du commit, mais il ne touche pas au répertoire de travail. Cela signifie que les modifications dans le répertoire de travail ne sont pas supprimées, mais elles ne sont pas incluses dans l'index pour le prochain commit. Vous devrez réindexer les modifications et créer un nouveau commit si vous le souhaitez.
3. **git reset --hard** : Cette option est la plus drastique. Lorsque vous utilisez **--hard**, Git déplace la branche courante (HEAD) vers le commit spécifié, réinitialise l'index pour correspondre à l'état du commit et réinitialise également le répertoire de travail pour correspondre à l'état de l'index. Cela signifie que toutes les modifications non enregistrées dans l'index et le répertoire de travail seront perdues. Utilisez cette option avec précaution, car elle peut entraîner la perte de données non sauvegardées.

Voici comment vous pourriez utiliser la commande **git reset** avec les différentes options :

* **git reset --soft HEAD~1**: Annule le dernier commit tout en conservant les modifications dans l'index et le répertoire de travail.
* **git reset --mixed HEAD~1** (ou simplement **git reset HEAD~1**): Annule le dernier commit et déplace les modifications de l'index vers le répertoire de travail.
* **git reset --hard HEAD~1**: Supprime complètement le dernier commit et toutes les modifications qu'il contenait.

Assurez-vous de comprendre les conséquences de chaque option avant d'utiliser la commande **git reset**, car elle peut avoir un impact significatif sur votre historique et vos données.

* 1. **Git revert**

**git revert** est une commande Git qui permet de créer un nouveau commit qui annule les modifications apportées par un ou plusieurs commits antérieurs. Contrairement à **git reset** qui modifie l'historique existant en déplaçant les branches, **git revert** agit en ajoutant de nouveaux commits pour annuler les changements spécifiques. Cela permet de conserver l'historique d'origine du dépôt tout en indiquant clairement les modifications annulées.

Voici comment fonctionne **git revert** en détail :

1. **Identifier les commits à annuler** : Tout d'abord, vous devez identifier les commits que vous souhaitez annuler. Vous pouvez spécifier un seul commit ou une plage de commits à l'aide de leurs identifiants de commit (hash) ou d'autres références.
2. **Exécution de la commande** : Une fois les commits ciblés identifiés, vous pouvez exécuter la commande **git revert**. Par exemple : git revert <commit1> <commit2> ...

Remplacez **<commit1>**, **<commit2>**, etc., par les identifiants des commits que vous souhaitez annuler.

1. **Résolution de conflits** : Il est possible que des conflits surviennent lorsque Git tente d'appliquer l'annulation des modifications sur le code actuel. Si cela se produit, vous devrez résoudre les conflits manuellement en modifiant les fichiers concernés. Après la résolution des conflits, vous devez utiliser **git add** pour indiquer à Git que les conflits sont résolus.
2. **Création du commit de réversion** : Une fois les conflits résolus et les modifications prêtes à être appliquées, vous pouvez finaliser la réversion en créant un nouveau commit. Git créera un nouveau commit qui contient les modifications nécessaires pour annuler les effets des commits spécifiés. Vous serez invité à fournir un message de commit décrivant la réversion.

Le résultat final est que vous aurez un nouveau commit dans l'historique du dépôt qui contient les modifications inverses des commits ciblés. Cela signifie que les changements apportés par les commits spécifiés seront annulés dans la branche actuelle. Cependant, l'historique des commits d'origine reste intact, ce qui permet de conserver une trace claire de ce qui a été fait.

En résumé, **git revert** est utilisé pour créer de nouveaux commits qui annulent spécifiquement les modifications apportées par des commits antérieurs, tout en conservant l'historique existant du dépôt.

* 1. **La différence entre git reset et git revert**

Voici une comparaison concise entre **git reset** et **git revert** :

**git reset** :

* Modifie l'historique en déplaçant les branches ou en supprimant des commits.
* Les commits ciblés sont retirés de l'historique.
* Peut provoquer la perte irréversible de données.
* Utilisé pour réorganiser l'historique, annuler des commits, ou réinitialiser les index et les répertoires de travail.
* Les commits supprimés peuvent être récupérés si vous avez sauvegardé leurs références.

**git revert** :

* Crée de nouveaux commits pour annuler spécifiquement les modifications de commits antérieurs.
* Les commits ciblés restent dans l'historique, mais leurs effets sont annulés.
* Préserve l'historique d'origine et permet d'annuler des changements de manière sûre.
* Utilisé pour annuler des modifications tout en maintenant la traçabilité de l'historique.
* N'entraîne généralement pas la perte de données, mais peut entraîner des conflits à résoudre.

En résumé, **git reset** modifie l'historique en supprimant ou déplaçant des commits, ce qui peut être risqué, tandis que **git revert** crée de nouveaux commits pour annuler les modifications spécifiques tout en conservant l'historique d'origine. La choix entre les deux dépend de vos besoins en gestion de l'historique et de la manière dont vous souhaitez traiter les modifications indésirables.

1. **Git cherry-pick** 
   1. **Definition**

La commande **git cherry-pick** est utilisée pour appliquer un commit spécifique à une branche différente. Voici comment utiliser cette commande :

1. **Assurez-vous d'être sur la branche cible :** Avant d'utiliser **git cherry-pick**, assurez-vous d'être sur la branche dans laquelle vous souhaitez appliquer le commit. Vous pouvez utiliser la commande **git checkout** pour changer de branche si nécessaire : git checkout branche\_cible
2. **Utilisez la commande git cherry-pick :** Exécutez la commande **git cherry-pick** suivi de l'identifiant (hash) du commit que vous souhaitez appliquer : git cherry-pick <identifiant\_du\_commit>

Git appliquera automatiquement les modifications du commit spécifié à la branche actuelle.

1. **Résolution des conflits (si nécessaire) :** Si des conflits surviennent pendant le processus de cherry-pick, vous devrez les résoudre manuellement. Utilisez **git status** pour voir quels fichiers sont en conflit, éditez ces fichiers pour résoudre les conflits, puis utilisez **git add** pour marquer les fichiers comme résolus. Une fois les conflits résolus, continuez avec le cherry-pick en utilisant **git cherry-pick --continue**.
2. **Terminer le cherry-pick :** Après avoir résolu les conflits, utilisez **git cherry-pick --continue** pour finaliser le processus de cherry-pick.
3. **Pousser les modifications (si nécessaire) :** Si vous souhaitez pousser les modifications du cherry-pick vers le dépôt distant, utilisez **git push** pour mettre à jour la branche distante avec les modifications que vous avez appliquées.

N'oubliez pas que **git cherry-pick** crée un nouveau commit avec les modifications du commit choisi. Cela peut être utile lorsque vous souhaitez intégrer sélectivement des modifications spécifiques d'une branche à une autre. Cependant, soyez conscient que cela peut parfois introduire des conflits ou des incohérences, surtout si le commit que vous choisissez dépend d'autres modifications non présentes dans la branche cible.

1. **Example**

Supposons que nous ayons deux branches : **branche\_source** et **branche\_cible**. Vous voulez appliquer trois commits spécifiques de **branche\_source** à **branche\_cible**.

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

* **A**, **B**, **C**, **D**, **E**, **F** représentent les commits de la branche source.
* **X**, **Y**, **Z** représentent les commits de la branche cible.

Pour appliquer les commits **C**, **E** et **F** de **branche\_source** à **branche\_cible**, vous pouvez utiliser la commande :

**Une image contenant texte, Police, capture d’écran, horloge

Description générée automatiquement**

Après le cherry-pick, votre historique pourrait ressembler à ceci :

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

* **C'**, **E'**, **F'** représentent les nouveaux commits créés par **git cherry-pick** sur la branche cible.

N'oubliez pas que chaque **git cherry-pick** crée un nouveau commit avec les modifications du commit choisi. Dans cet exemple, vous avez appliqué sélectivement les commits **C**, **E** et **F** de la branche source sur la branche cible. Assurez-vous de comprendre l'impact de cette opération, en particulier si les commits dépendent d'autres modifications qui ne sont pas présentes sur la branche cible.

1. **Patche Tar9i3e**
2. **Git rebase**
   1. **Git rebase**

**git rebase** est une commande de Git utilisée pour réappliquer les commits d'une branche sur le dessus d'une autre branche. L'objectif principal du rebase est de maintenir un historique de commits linéaire et cohérent, en replaçant les commits d'une branche à un autre point de départ.

En d'autres termes, le rebase vous permet de déplacer les commits d'une branche vers le sommet d'une autre branche, en modifiant l'historique de commits. Cela peut être particulièrement utile pour intégrer proprement des modifications d'une branche de fonctionnalité sur une branche principale, tout en maintenant un historique linéaire plus propre et en évitant les commits de fusion.

La syntaxe de base de la commande **git rebase** est la suivante :

**git rebase branche-cible**

Cela signifie que vous êtes sur la branche que vous souhaitez rebaser (habituellement une branche de fonctionnalité) et vous souhaitez réappliquer ses commits sur le sommet de **branche-cible** (habituellement la branche principale).

Il est important de noter que le rebase peut potentiellement créer des conflits lorsque les commits sont réappliqués sur la nouvelle base, en particulier si des modifications similaires ont été apportées sur les deux branches. Dans ce cas, vous devrez résoudre les conflits manuellement comme vous le feriez avec un merge.

L'utilisation de **git rebase** peut rendre l'historique des commits plus lisible et linéaire, mais il est recommandé de l'utiliser avec prudence, en particulier sur des branches partagées avec d'autres membres de l'équipe, car elle modifie l'historique des commits et peut provoquer des problèmes si elle est mal utilisée.

* 1. **La différence entre git merge et git rebase**

La commande **git rebase** est utilisée pour intégrer les modifications d'une branche sur une autre en réécrivant l'historique de commits de manière linéaire, plutôt que de créer un nouveau commit de fusion comme avec la commande **git merge**. Cela peut parfois donner une vue plus propre et linéaire de l'historique des commits, mais cela nécessite une approche plus soignée pour éviter les conflits.

Voici la différence fondamentale entre **git merge** et **git rebase** :

1. **Git Merge :**
   * Crée un nouveau commit de fusion qui combine les changements de deux branches.
   * Préserve l'historique d'origine de chaque branche, ce qui signifie que vous pouvez voir quand chaque branche a été fusionnée dans l'autre.
   * Peut générer une structure d'historique de commits plus complexe, avec des branches de fusion visibles.
2. **Git Rebase :**
   * Réapplique les commits d'une branche au sommet d'une autre branche en les replaçant dans l'ordre.
   * Produit un historique de commits linéaire et plus propre, car les commits de la branche rebasée semblent avoir été créés directement après les commits de la branche de base.
   * Peut simplifier l'historique des commits, en particulier lorsque vous avez fait plusieurs petits commits et que vous souhaitez les condenser en un seul commit significatif.
   * Peut créer des conflits plus facilement, car chaque commit est réappliqué individuellement.

En résumé, **git merge** crée un nouveau commit de fusion qui combine les changements des branches source et cible, tandis que **git rebase** réapplique les commits de la branche source sur la branche cible, produisant un historique de commits linéaire. Le choix entre **git merge** et **git rebase** dépend du flux de travail de votre équipe, de vos préférences et de la manière dont vous souhaitez organiser votre historique de commits.

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

Une image contenant texte, écriture manuscrite, diagramme, ligne

Description générée automatiquement

**2. git rebase -i**

La commande **git rebase -i** est une commande puissante de Git qui permet de réorganiser, éditer ou fusionner des commits dans l'historique d'une branche. L'option **-i** signifie "interactive", ce qui signifie que vous pouvez interagir avec l'éditeur de texte pour spécifier les actions que vous souhaitez effectuer sur les commits.

L'utilisation de **git rebase -i** vous permet de :

1. **Réorganiser l'historique :** Vous pouvez changer l'ordre des commits, les fusionner ou même les supprimer pour obtenir un historique plus propre et cohérent.
2. **Éditer les messages de commit :** Vous pouvez modifier les messages de commit pour les rendre plus clairs ou corriger des erreurs.
3. **Séparer un commit en plusieurs commits :** Vous pouvez diviser un commit en plusieurs commits plus petits.
4. **Fusionner des commits :** Vous pouvez fusionner plusieurs commits en un seul, ce qui est utile pour combiner des modifications liées.
5. **Squasher des commits :** Vous pouvez fusionner plusieurs commits en un seul, en écrasant les messages de commit précédents, ce qui est utile pour nettoyer l'historique avant de le pousser vers un dépôt distant.

Voici comment utiliser **git rebase -i** :

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

1. **Git tag**

En Git, un tag est une référence statique à un commit spécifique dans l'historique. Les tags sont généralement utilisés pour marquer des points importants dans l'historique d'un projet, tels que des versions stables, des sorties majeures ou des jalons importants. Les tags sont immuables, ce qui signifie qu'ils pointent toujours vers le même commit et ne changent pas avec les nouveaux commits.

Il existe deux types de tags en Git : les tags légers (lightweight tags) et les tags annotés (annotated tags).

1. **Tags légers (Lightweight Tags) :** Les tags légers sont simplement des pointeurs vers un commit spécifique, sans informations supplémentaires telles que l'auteur, la date et le message. Pour créer un tag léger, utilisez la commande **git tag** suivi du nom du tag et de l'identifiant du commit :

**git tag nom\_du\_tag identifiant\_du\_commit**

1. **Tags annotés (Annotated Tags) :** Les tags annotés sont similaires aux tags légers, mais ils sont associés à des métadonnées supplémentaires, comme le nom de l'auteur, la date et un message de description. Les tags annotés sont recommandés lorsque vous souhaitez ajouter des informations supplémentaires à un tag. Pour créer un tag annoté, utilisez la commande **git tag** suivi de l'option **-a** pour annoter le tag, et de l'identifiant du commit :

**git tag -a nom\_du\_tag -m "Description du tag" identifiant\_du\_commit**

1. **Lister les tags :** Pour afficher la liste des tags dans le dépôt, utilisez la commande **git tag** sans aucun argument :

**git tag**

1. **Afficher les détails d'un tag annoté :** Pour afficher les détails d'un tag annoté, utilisez la commande **git show** suivi du nom du tag :

**git show nom\_du\_tag**

1. **Partager des tags :** Par défaut, les tags ne sont pas automatiquement partagés avec les dépôts distants lors d'un push. Pour pousser des tags vers un dépôt distant, utilisez la commande **git push** avec l'option **--tags** :

**git push origin –tags**

Les tags sont utiles pour marquer des points clés dans l'historique, identifier des versions stables ou importantes, et pour créer des repères dans le développement du projet. Utilisez les tags pour simplifier la référence à des commits spécifiques dans l'histoire de votre projet.

1. **Git config**

La commande **git config** est utilisée pour configurer diverses options de configuration dans Git. Ces options incluent des informations telles que le nom d'utilisateur, l'adresse e-mail, les alias de commandes et d'autres paramètres spécifiques à Git. Voici comment utiliser la commande **git config** avec différentes options :

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

Ces exemples illustrent quelques-unes des utilisations courantes de la commande **git config**. Vous pouvez consulter la documentation de Git ou exécuter **git help config** pour obtenir plus d'informations sur les options et les paramètres disponibles.

1. **Remote (Push & Pull) –Depots Git distants**
   1. **Definition**

Les dépôts distants (remote repositories) ne sont pas si compliqués que ça. Dans l'ère actuelle du cloud computing il est facile de croire qu'il y a un peu de magie derrière les branches distantes, mais elles sont en fait juste des copies de votre dépôt local sur un autre ordinateur. Vous pouvez donc vous connecter à cet ordinateur par Internet, ce qui vous permet de transférer les commits en retard et en avance.

Cela dit, les branches distantes ont plusieurs propriétés puissantes :

* Tout d'abord, le dépôt distant sert de sauvegarde ! Le dépôt local de Git a la capacité de restaurer des fichiers à un état précédent (comme vous le savez), mais toutes les informations sont stockées localement. En ayant des copies de votre dépôt Git sur d'autres ordinateurs, vous pouvez perdre vos données et toujours repartir de là où vous en étiez resté.
* Plus important encore, les dépôts distants sociabilisent le projet ! Maintenant qu'il est hébergé quelque part ailleurs, vos amis peuvent y contribuer facilement (ou récupérer vos derniers changements).

Il est devenu courant de visualiser l'activité sur dépôt distant via des sites internet (comme [**GitHub**](https://github.com/)), mais les dépôts distants servent toujours de colonne vertébrale à ce système. Il est donc important de les comprendre !

* 1. **Git CLone**

"git clone" est une commande utilisée dans Git, un système de gestion de version, pour créer une copie locale d'un dépôt distant. Elle récupère l'intégralité de l'historique et des fichiers du dépôt, permettant aux utilisateurs de collaborer, de travailler et de versionner les fichiers localement.

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **Les Branche distaant c’est quoi**

**Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement**

**Exécuter la cmd git checkout o/main ; git commit**

**Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement**

1. **gitIgnore**

Le fichier **.gitignore** est un fichier utilisé dans les dépôts Git pour spécifier les fichiers et les répertoires qui doivent être ignorés par Git lors du suivi des modifications. En d'autres termes, les fichiers et les répertoires listés dans le fichier **.gitignore** ne seront pas inclus dans les commits ni pris en compte par Git lors de l'exécution de certaines commandes, comme **git add** et **git status**.

Voici comment fonctionne le fichier **.gitignore** :

1. **Création du fichier :** Vous créez un fichier nommé **.gitignore** à la racine de votre dépôt Git.
2. **Ajout de règles :** Dans ce fichier, vous spécifiez les règles pour les fichiers et les répertoires que vous souhaitez ignorer. Chaque règle est une ligne distincte.
3. **Syntaxe des règles :** Les règles peuvent inclure des motifs (patterns) pour spécifier les fichiers à ignorer. Les motifs peuvent inclure des caractères génériques comme **\*** (match n'importe quel caractère) et **/** (délimiteur de répertoire).

Exemple de contenu pour un fichier **.gitignore** :

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

Dans cet exemple, le fichier **config.ini** ainsi que tous les fichiers avec les extensions **.o** et **.class** seront ignorés. De plus, les répertoires **node\_modules** et **vendor** (et tout leur contenu) seront également ignorés.

Il est important d'utiliser un fichier **.gitignore** pour éviter d'inclure accidentellement des fichiers sensibles, volumineux ou temporaires dans vos commits. Cela contribue à garder l'historique de votre dépôt propre et concentré sur les fichiers pertinents pour votre projet.

Lorsque vous modifiez le fichier **.gitignore**, assurez-vous de faire un commit de ce fichier afin que les règles soient prises en compte par tous les contributeurs du projet.