FICHE 2 : Apport théorique pour GIT et les techniques de gestion de versions.

Sources : Cisco Netacad (DevAsc)

## Types de systèmes de contrôle de version

Le contrôle de version, également appelé systèmes de contrôle de version, contrôle de révision ou contrôle de source, est un moyen de gérer les modifications apportées à un ensemble de fichiers afin de conserver un historique de ces modifications. Pensez à toutes les fois où vous avez fait une copie d'un fichier avant de le modifier, juste au cas où vous voulez revenir à l'original. Le contrôle de version gère tout cela pour vous.

Les systèmes de contrôle de version stockent l'ensemble maître de fichiers et l'historique des modifications dans un référentiel, également connu sous le nom de repo. Pour apporter une modification à un fichier, une personne doit obtenir une copie de travail du référentiel sur son système local. La copie de travail est la copie personnelle des fichiers de la personne, où elle peut apporter des modifications sans affecter d'autres personnes. Certains des avantages du contrôle de version sont les suivants:

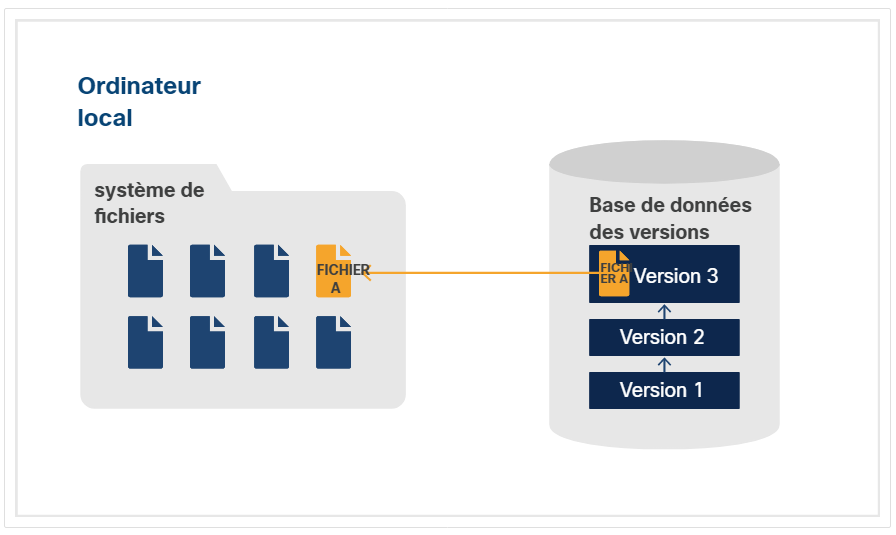
* **Il permet la collaboration** - Plusieurs personnes peuvent travailler sur un projet (un ensemble de fichiers) en même temps sans supplanter les changements de l'autre.
* **Responsabilité et visibilité** - Sachez qui a apporté les changements, quand ils ont apporté ces changements et pourquoi.
* **Travailler isolément** - Créez de nouvelles fonctionnalités indépendamment sans affecter le logiciel existant.
* **Sécurité** - Les fichiers peuvent être annulés lorsqu'une erreur est commise.
* **Travailler n'importe où** - Les fichiers sont stockés dans un référentiel, de sorte que n'importe quel périphérique peut avoir une copie de travail.

**Types de systèmes de contrôle de version**

Il existe trois types de systèmes de contrôle de version:

* Local
* Centralisé
* Applications

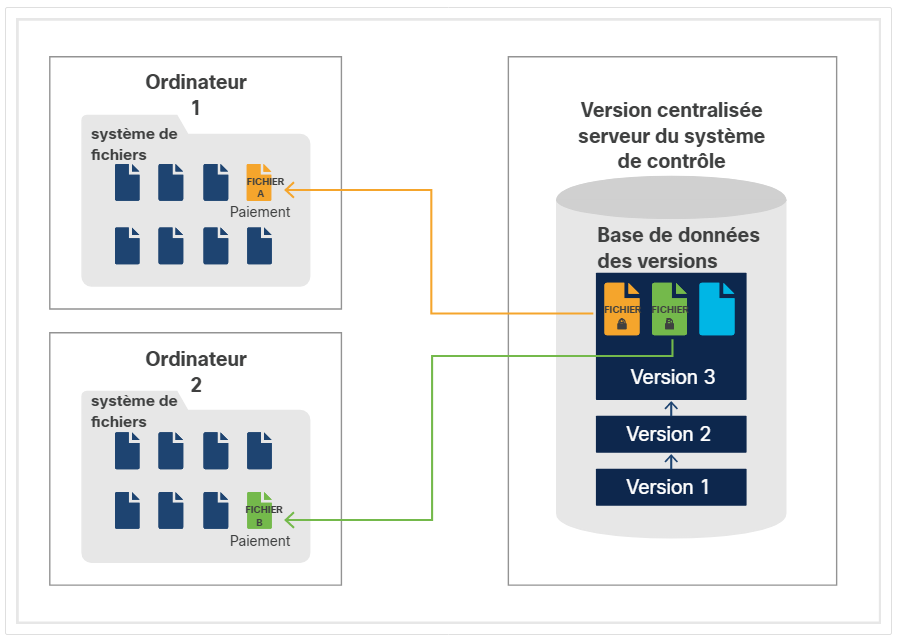
## Système de contrôle de version local



Tout comme le nom indique, un système de contrôle de version local (LVCS) suit les fichiers dans un système local. Un système de contrôle de version local remplace le scénario "faire une copie du fichier avant d'éditer d'autres". L'objectif d'un système de contrôle de version local est principalement de pouvoir revenir à une version précédente. Ce type de contrôle de version n'est pas destiné à traiter la plupart des avantages énumérés ci-dessus.

Les systèmes de contrôle de version locaux utilisent une base de données simple pour suivre toutes les modifications apportées au fichier. Dans la plupart des cas, le système stocke le delta entre les deux versions du fichier, par opposition au fichier lui-même. Lorsque l'utilisateur veut rétablir le fichier, le delta est inversé pour accéder à la version demandée.

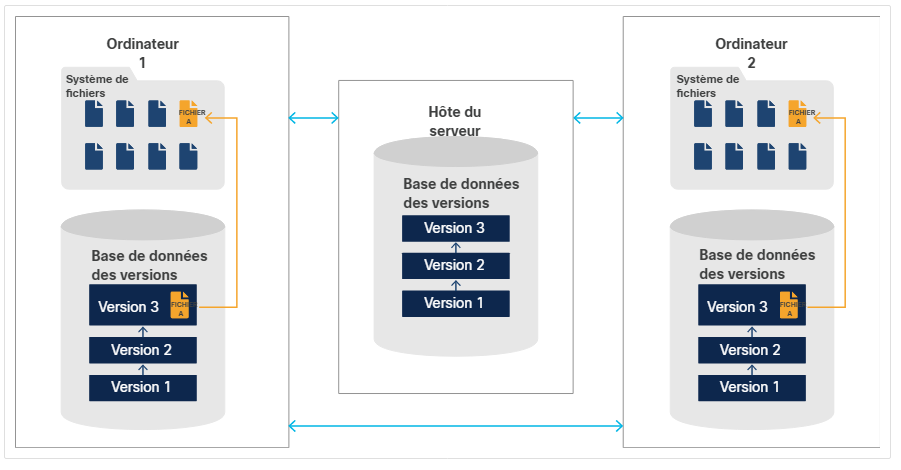
## Système de contrôle de version local



Un système de contrôle de version centralisé (CVCS) utilise un modèle serveur-client. Le référentiel (également connu sous le nom de repo), qui est la seule copie de l'ensemble des fichiers et de l'historique, est stocké à un emplacement centralisé, sur un serveur. Chaque fois qu'une personne souhaite apporter une modification à un fichier, elle doit d'abord obtenir une copie de travail du fichier du référentiel vers son propre système, le client.

Dans un système de contrôle de version centralisé, une seule personne à la fois peut travailler sur un fichier particulier. Afin d'appliquer cette restriction, une personne doit extraire le fichier, ce qui verrouille le fichier et l'empêche d'être modifié par quelqu'un d'autre. Lorsque l'individu a terminé d'apporter des modifications, il doit enregistrer le fichier, ce qui applique les modifications de l'individu à la copie principale dans le repo, balise une nouvelle version et déverrouille le fichier pour que d'autres personnes puissent apporter des modifications.

## Système de contrôle de version distribuée



Un système de contrôle de version distribué (DVCS) est un modèle peer-to-peer. Le référentiel peut être stocké sur un système client, mais il est généralement stocké dans un service d'hébergement de référentiel. Lorsqu'une personne souhaite apporter une modification à un fichier, elle doit d'abord cloner le référentiel complet sur son propre système. Cela inclut l'ensemble des fichiers ainsi que l'ensemble de l'historique des fichiers. L'avantage de ce modèle est que le référentiel complet sera sur plusieurs systèmes et peut être utilisé pour restaurer le référentiel dans le service d'hébergement du référentiel si un événement tel que la corruption des données se produit.

Dans un système de contrôle de version distribué, chaque individu peut travailler sur n'importe quel fichier, même en même temps, car le fichier local dans la copie de travail est ce qui est modifié. Par conséquent, le verrouillage du fichier n'est pas nécessaire. Lorsque la personne a terminé d'apporter les modifications, elle pousse le fichier vers le référentiel principal qui se trouve dans le service d'hébergement du référentiel, et le système de contrôle de version détecte tout conflit entre les modifications de fichiers.

## Git

Au moment de cette écriture, le système de contrôle de version le plus populaire utilisé est Git. Git est une implémentation open source d'un système de contrôle de version distribué qui est actuellement la dernière tendance dans le développement de logiciels. Git:

* Est facile à apprendre
* Peut gérer tous les types de projets, y compris les projets de grande entreprise
* A des performances rapides
* Est conçu pour les projets collaboratifs
* Est flexible
* A une petite empreinte
* A tous les avantages d'un système de contrôle de version distribué
* Est gratuit

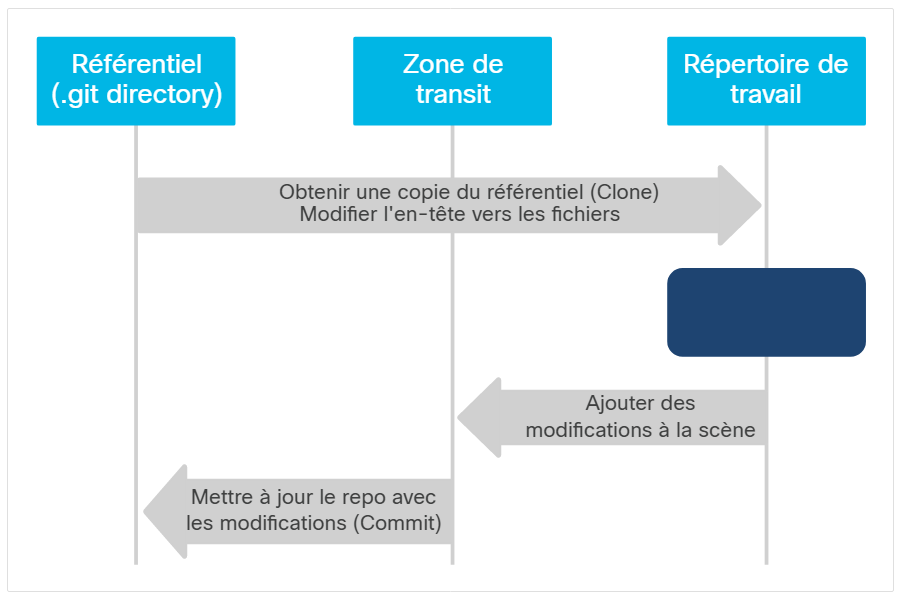
Un client Git doit être installé sur une machine cliente. Il est disponible pour macOS, Windows et Linux/Unix. Bien que certains clients Git viennent avec une interface graphique de base, Git se concentre sur l'interface de ligne de commande, à propos de laquelle nous reviendrons plus tard.

Une différence clé entre Git et les autres systèmes de contrôle de version est que Git stocke les données sous forme d'instantanés au lieu de différences (le delta entre le fichier actuel et la version précédente). Si le fichier ne change pas, git utilise un lien de référence vers le dernier instantané du système au lieu de prendre un instantané nouveau et identique.

La figure montre la vue fichier à gauche et la vue version à droite. Dans la vue des fichiers, il y a une flèche qui va de gauche à droite en haut avec le mot checkins au-dessus de la ligne. Sous la ligne se trouvent quatre blocs colorés: version 1 version 2, version 3 et version 4. La ligne suivante est Fichier A (A1) sous la version 1 et flèche allant à une zone de texte (A2) sous la version 2. Une flèche va à la zone de texte suivante, A3, qui se trouve sous la version 4. Sur la ligne suivante se trouve la zone de texte Fichier B (B1) et une flèche qui va à la zone de texte B2 située sous la version 3. Une flèche va à la zone de texte B3 qui se trouve sous la version 4. La dernière ligne est Fichier C (C1) qui a une flèche qui s'étend sur toutes les quatre versions. Dans la vue de version, il y a quatre blocs colorés étiquetés version 1, version 2, version 3 et version 4. Chaque bloc de la vue de version est soit un instantané, soit un lien de référence. Seuls les liens de référence seront appelés. Les autres versions sont des instantanés. Dans la première ligne se trouve le fichier A (A1) sous la version 1, A2 sous la version 2, le lien de référence A2 sous la version 3, et A3 sous la version 4. Dans la deuxième ligne se trouve le fichier B (B1) sous la version 1, le lien de référence B1 sous la version 2, B2 sous la version 3 et B3 sous la version 4. Dans la dernière ligne se trouve le fichier C (C1) sous la version 1, le lien de référence C1 sous les versions 2, 3 et 4.

## 

## Git Trois étapes

Git est organisé par trois — trois étapes, et trois états.

Il y a trois étapes dans Git:

* référentiel ( the .git directory)
* répertoire de travail
* zone de transit

**Référentiel (.GIT DIRECTORY)﻿**

Étant donné que Git est un système de contrôle de version distribué, chaque client dispose d'une copie complète du dépôt. Lorsqu'un projet devient un référentiel Git, un répertoire .git caché est créé, et c'est essentiellement le référentiel. Le répertoire .git contient des métadonnées telles que les fichiers (compressés), les validations et les journaux (historique de validation).

**RÉPERTOIRE DE TRAVAIL**

Le répertoire de travail est le dossier visible dans le système de fichiers. Il s'agit d'une copie des fichiers dans le référentiel. Ces fichiers peuvent être modifiés et les modifications ne sont visibles que par l'utilisateur du client. Si le système de fichiers du client est corrompu, ces modifications seront perdues, mais le dépôt principal reste intact.

**ZONE DE TRANSIT**

La zone de transit stocke les informations sur ce que l'utilisateur souhaite ajouté/mettre à jour/supprimer dans le référentiel. L'utilisateur n'a pas besoin d'ajouter tous ses fichiers modifiés au stage/repo; il peut sélectionner des fichiers spécifiques. Bien qu'il soit appelé zone, il s'agit en fait d'un fichier d'index situé dans le répertoire .git.

**Trois États**

Comme il y a trois étapes dans Git, il y a trois états correspondants pour un fichier Git:

* **Commis (committed)** - Ceci est la version du fichier a été enregistrée dans le référentiel (.git directory).
* **modifié (modified)** - Le fichier a changé mais n'a pas été ajouté à la zone de transit ou validé dans le référentiel.
* **Mis en scène (staged)** - Le fichier modifié est prêt à être validé dans le référentiel.

## Référentiels locaux et distants

Git a deux types de référentiels, locaux et distants.

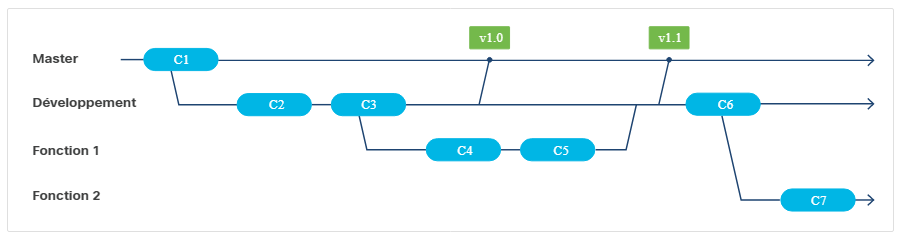
Un référentiel local est stocké sur le système de fichiers d'une machine cliente, qui est le même sur lequel les commandes git sont exécutées.

Un référentiel distant est stocké ailleurs que l'ordinateur client, généralement un serveur ou un service d'hébergement de référentiel. Les référentiels distants sont facultatifs et sont généralement utilisés lorsqu'un projet nécessite une collaboration entre une équipe avec plusieurs utilisateurs et machines clientes.

Les référentiels distants peuvent être considérés comme le référentiel "centralisé" pour Git, mais cela n'en fait pas un CVCS. Un référentiel distant avec Git continue d'être un DVCS car le référentiel distant contiendra le référentiel complet, qui inclut le code et l'historique des fichiers. Lorsqu'une machine cliente clone le référentiel, elle obtient le référentiel complet sans avoir besoin de le verrouiller, comme dans un CVCS.

Une fois le référentiel local cloné à partir du référentiel distant ou le référentiel distant créé à partir du référentiel local, les deux référentiels sont indépendants l'un de l'autre jusqu'à ce que les modifications de contenu soient appliquées à l'autre branche par l'exécution manuelle d'une commande Git.

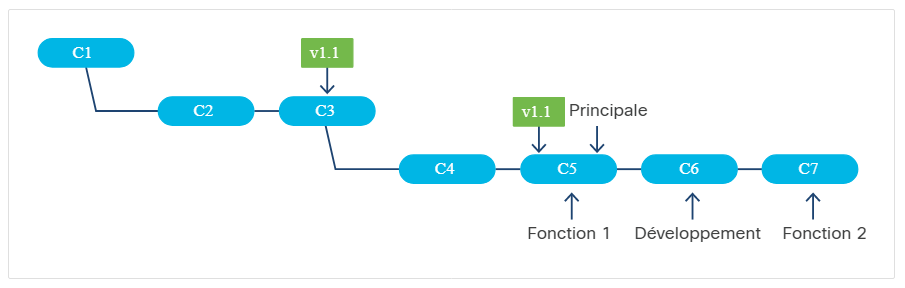
## Qu’est-ce que les branches ?

Le branchage permet aux utilisateurs de travailler sur le code indépendamment sans affecter le code principal dans le référentiel. Lorsqu'un référentiel est créé, le code est automatiquement placé sur une branche appelée Master. Les utilisateurs peuvent avoir plusieurs branches et celles-ci sont indépendantes les unes des autres. Le branchement permet aux utilisateurs de:

* Travailler sur une fonctionnalité indépendamment tout en bénéficiant d'un système de contrôle de version distribué
* Travailler simultanément sur plusieurs entités
* Expérimenter des idées de code
* Gardez le code de production, de développement et de fonctionnalité séparément
* Gardez la ligne principale de code stable

Les branches peuvent être locales ou distantes, et elles peuvent être supprimées. Les branches locales facilitent l'essai de différentes implémentations de code car une branche peut être utilisée si elle réussit et supprimée si elle ne l'est pas. La fusion d'une branche vers la branche parent n'est pas obligatoire.

Contrairement à d'autres systèmes de contrôle de version, la création de branche de Git est légère, et le basculement entre les branches est presque instantané. Bien que les branches soient souvent visuellement dessinées sous forme de chemins séparés, les branches Git ne sont essentiellement que des pointeurs vers le commit approprié.

 Les branches sont comme un carrefour sur la route, où il commence par le code et l'histoire au point de détournement, puis construit son propre chemin avec de nouveaux engagements de manière indépendante. En conséquence, les branches ont leur propre historique, zone de transit et répertoire de travail. Lorsqu'un utilisateur passe d'une branche à une autre, le code dans son répertoire de travail et les fichiers de la zone de transit changent en conséquence, mais les répertoires du dépôt (.git) restent inchangés.

Dans la mesure du possible, vous devriez essayer d'utiliser des branches plutôt que de mettre à jour le code directement vers la branche principale afin d'éviter les mises à jour accidentelles qui cassent le code.

## GitHub et autres fournisseurs ?

Traiter des projets utilisant Git est souvent associé à GitHub, mais Git et GitHub ne sont pas les mêmes. Git est une implémentation du contrôle de version distribué et fournit une interface de ligne de commande. GitHub est un service, fourni par Microsoft, qui implémente un service d'hébergement de référentiel avec Git.

En plus de fournir la fonctionnalité de contrôle de version distribuée et de gestion du code source de Git, GitHub fournit également des fonctionnalités supplémentaires telles que:

* Révision du code
* Documentation
* Gestion de projets
* Suivi des bugs
* Demande de fonction

GitHub a évolué pour prendre en charge de nombreuses formes de codage collaboratif, notamment:

* Repos privés visibles uniquement par les équipes désignées
* Projets de "codage social" qui sont publics, mais dont les contributeurs peuvent être anonymes
* Des efforts open source à grande échelle avec de nombreux contributeurs, parfois en milliers

Pour permettre aux propriétaires de projets de gérer dans des scénarios aussi largement disparates, GitHub a introduit le concept de "demande d'extraction". Une demande de retrait est un moyen d'officialiser une demande d'un contributeur pour examiner les changements tels que le nouveau code, les modifications apportées au code existant, etc., dans la branche du contributeur pour l'inclure dans la branche principale du projet ou dans d'autres succursales. L'idiome de requête pull est désormais universellement implémenté dans les services d'hébergement Git.

GitHub n'est pas le seul service d'hébergement de référentiel utilisant Git, d'autres incluent Gitlab et Bitbucket.

## Commandes GIT

**Configuration de Git**

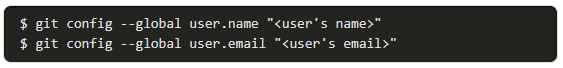
Après avoir installé Git sur la machine cliente, vous devez le configurer. Git fournit une commande git config pour obtenir et définir les paramètres globaux de Git, ou les options d'un dépôt.

Pour configurer Git, utilisez l'option —global pour définir les paramètres globaux initiaux.

**Commande**: git config --global key value

L'utilisation de l'option —global écrira dans le fichier global ~/.gitconfig.

Pour que chaque utilisateur soit responsable de ses modifications de code, chaque installation de Git doit définir le nom et l'e-mail de l'utilisateur. Pour ce faire, utilisez les commandes suivantes:



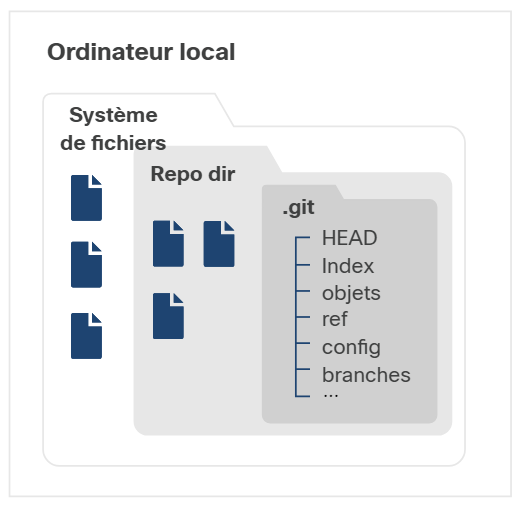
où <user's name>et <user's email> sont respectivement le nom et l'adresse e-mail de l'utilisateur.

**Créer un nouveau dépôt Git**

Tout projet (dossier) dans le système de fichiers local d'un client peut devenir un dépôt Git. Git fournit une commande git init pour créer un dépôt Git vide ou faire d'un dossier existant un dépôt Git. Lorsqu'un projet nouveau ou existant devient un dépôt Git, un répertoire .git masqué est créé dans ce dossier de projet. Rappelez-vous que le répertoire .git est le référentiel qui contient les métadonnées telles que les fichiers compressés, l'historique de validation et la zone de transit. En plus de créer le répertoire .git, Git crée également la branche maître.

**Commande**: git init

Pour faire d'un projet nouveau ou existant un dépôt Git, utilisez la commande suivante:



où le <project directory> est le chemin absolu ou relatif vers le projet nouveau ou existant. Pour un nouveau dépôt Git, le répertoire dans le chemin fourni sera créé en premier, suivi de la création du répertoire .git.

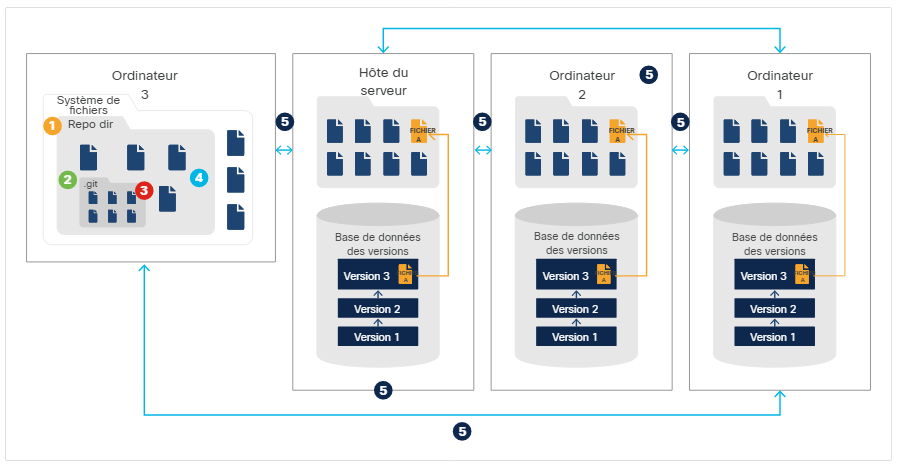
La création d'un dépôt Git ne suit pas automatiquement les fichiers du projet. Les fichiers doivent être explicitement ajoutés au nouveau référentiel pour être suivis. Les détails sur la façon d'ajouter des fichiers à un référentiel seront abordés ultérieurement.

**Obtenir un référentiel Git existant**

Avec Git, il est facile d'obtenir une copie des référentiels existants et de contribuer à ces référentiels. Git fournit une commande git clone qui clone un dépôt existant sur le système de fichiers local. Git étant un DVCS, il clone le dépôt complet, qui inclut l'historique des fichiers et les branches de suivi à distance.

**Commande** : git clone <repository> [target directory]

où <repository> est l'emplacement du dépôt à cloner. Git prend en charge quatre protocoles de transport principaux pour accéder au <repository> : Local, Secure Shell (SSH), Git et HTTP. Le [target directory] est facultatif et représente le chemin absolu ou relatif de l'emplacement de stockage des fichiers clonés. Si vous ne fournissez pas le répertoire du projet, git copie le référentiel à l'emplacement où vous avez exécuté la commande.



Lorsque vous exécutez la commande git clone, Git:

1. Crée le répertoire de travail sur le système de fichiers local avec le nom du référentiel ou le nom spécifié, le cas échéant.
2. Crée un répertoire .git dans le dossier nouvellement créé.
3. Copie les métadonnées du référentiel dans le répertoire .git nouvellement créé.
4. Crée la copie de travail de la dernière version des fichiers de projet.
5. Duplique la structure de branche du référentiel distant cloné et permet le suivi des modifications apportées à chaque branche, localement et à distance — cela inclut la création et l'extraction d'une branche active locale, "fourchée" à partir de la branche active actuelle du référentiel cloné.

Veuillez consulter la documentation officielle git clone pour plus de détails et les options de ligne de commande.

**Afficher les fichiers modifiés dans le répertoire de travail**

Qu'est-ce qui a été changé dans le répertoire de travail? Quels fichiers ont été ajoutés dans la zone de transit? Git fournit une commande git status pour obtenir une liste de fichiers qui ont des différences entre le répertoire de travail et la branche parent. Cela inclut les fichiers non suivis nouvellement ajoutés et les fichiers supprimés. Il fournit également une liste des fichiers qui se trouvent dans la zone de transit. Notez que la différence est calculée en fonction du dernier commit que le clone local a copié à partir de la branche parent dans le dépôt Git, pas nécessairement de la dernière version dans le dépôt distant. Si des modifications ont été apportées depuis que le dépôt a été cloné, Git ne tiendra pas compte de ces modifications.

**Command** : git status

En plus de fournir la liste des fichiers, la sortie de la commande git status fournit des informations supplémentaires sur l’état du dépôt telles que:

* Branche actuelle du répertoire de travail
* Nombre de validations dans le répertoire de travail se trouve derrière la dernière version de la branche parent
* Instructions sur la mise à jour du référentiel local et comment stage/déstage des fichiers

Veuillez consulter la documentation officielle git status pour plus de détails et les options de ligne de commande.

**Comparer les modifications entre les fichiers**

Vous voulez savoir ce qui a été modifié dans un fichier, ou la différence entre deux fichiers? Git fournit une commande git diff qui est essentiellement un outil générique de comparaison de fichiers.

**Commande** : git diff

Étant donné que cette commande est un outil générique de différence de fichier, elle inclut de nombreuses options pour la comparaison de fichiers. Lorsque vous utilisez cette commande, le fichier n'a pas besoin d'être un fichier suivi Git.

Par exemple, vous pouvez:

1. Afficher les modifications entre la version du fichier dans le répertoire de travail et le dernier commit copié par le clone local depuis la branche parent dans le dépôt Git:$ git diff <file path>

2. Afficher les modifications entre la version du fichier dans le répertoire de travail et un commit particulier à partir de l'historique des fichiers:$ git diff <commit id> <file path

3. Afficher les modifications entre les deux validations d'un fichier à partir de l'historique du fichier. <file path> est le chemin absolu ou relatif du fichier à comparer et <commit id> `est l'identifiant de la version du fichier à comparer.  
$ git diff <commit id 1> <commit id 2> [<file path>]  
$ git diff <commit id 1> <commit id 2>

4.Afficher les modifications entre deux fichiers dans le répertoire de travail ou sur le disque.  
$ git diff <file path 1> <file path 2>  
Veuillez consulter la documentation officielle git diff pour plus de détails et les options de ligne de commande.

## Ajout et suppression de fichiers

**Ajout de fichiers à la zone intermédiaire**

Une fois que des modifications ont été apportées à un fichier dans le répertoire de travail, il doit d'abord aller dans la zone de transit avant de pouvoir être mis à jour dans le dépôt Git. Git fournit une commande git add pour ajouter des fichiers à la zone de transit. Ces fichiers ajoutés au transfert peuvent inclure des fichiers nouvellement non suivis, des fichiers suivis existants qui ont été modifiés ou même des fichiers suivis qui doivent être supprimés du référentiel. Les fichiers modifiés n'ont pas besoin d'être ajoutés au répertoire de travail, sauf si les modifications doivent être ajoutées au référentiel.

**Commande** : git add

Cette commande peut être utilisée plusieurs fois avant la mise à jour du dépôt Git (en utilisant commit). En outre, le même fichier peut être ajouté à la scène plusieurs fois avant un commit. Seuls les fichiers spécifiés dans la commande git add sont ajoutés à la zone de transit.

Pour ajouter un seul fichier à la zone de transit:

Pour ajouter plusieurs fichiers à la zone intermédiaire où le <file path> est le chemin absolu ou relatif du fichier à ajouter à la zone intermédiaire et peut accepter des caractères génériques.



Pour ajouter tous les fichiers modifiés à la zone de transit:

Rappelez-vous que Git a trois étapes, donc ajouter des fichiers à la zone de transit n'est que la première étape du processus en deux étapes pour mettre à jour le dépôt Git.

**Suppression de fichiers du référentiel Git**

Il existe deux façons de supprimer des fichiers du dépôt Git.

**OPTION 1**

Git fournit une commande git rm pour supprimer des fichiers du dépôt Git. Cette commande ajoute la suppression du (des) fichier (s) spécifié (s) à la zone de transit. Il n'effectue pas la deuxième étape de la mise à jour du dépôt Git lui-même.

**Commande** : git rm

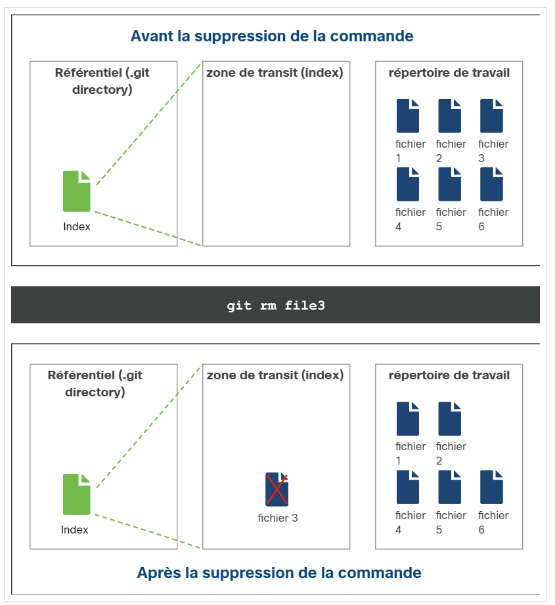
Pour supprimer le ou les fichiers spécifiés du répertoire de travail et ajouter cette modification à la zone de transit, utilisez la commande suivante:

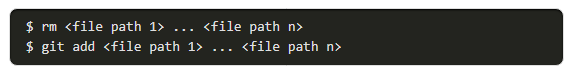


où <file path> est le chemin absolu ou relatif du fichier à supprimer du dépôt Git.

Pour ajouter le(s) fichier(s) spécifié(s) à supprimer à la zone de transit sans supprimer le(s) fichier(s) lui-même du répertoire de travail, utilisez la commande suivante:

Cette commande ne fonctionnera pas si le fichier se trouve déjà dans la zone de transit avec des modifications.

**OPTION 2**

Cette option est un processus en deux étapes. Tout d'abord, utilisez la commande standard du système de fichiers pour supprimer le ou les fichiers. Ensuite, ajoutez le fichier à la scène en utilisant la commande Git add qui a été discuté plus tôt.

Ce processus en deux étapes équivaut à l'utilisation de la commande git rm <file path 1> ... <file path n>. L'utilisation de cette option ne permet pas de conserver le fichier dans le répertoire de travail.

## Mise à jour des référentiels

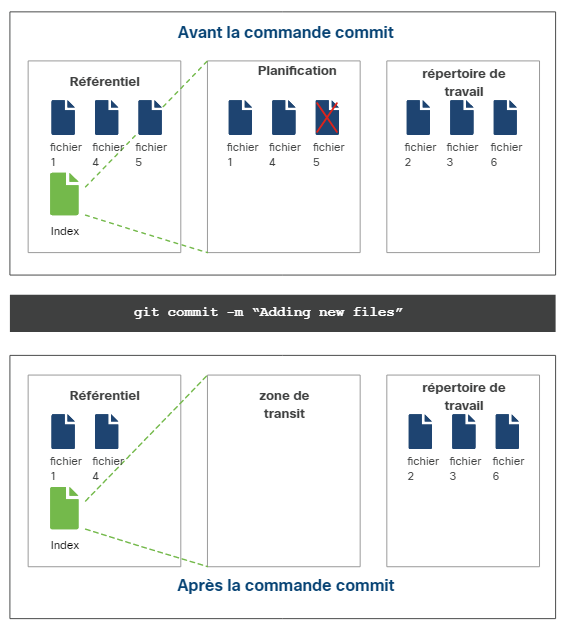
**Mise à jour du référentiel local avec les modifications apportées à la zone de transit**

Rappelez-vous que dans Git, les modifications apportées à un fichier passent par trois étapes: répertoire de travail, zone de transit et dépôt. L'obtention des modifications de contenu du répertoire de travail vers la zone de transit peut être effectuée avec la commande git add, mais comment les mises à jour parvient-elles au référentiel? Git fournit une commande git commit pour mettre à jour le référentiel local avec les modifications qui ont été ajoutées dans la zone de transit.

**Command** : git commit

Cette commande combine toutes les modifications de contenu dans la zone de transit en un seul commit et met à jour le référentiel Git local. Ce nouveau commit devient le dernier changement dans le dépôt Git. S'il existe un dépôt Git distant, il n'est pas modifié avec cette commande.

Pour valider les modifications à partir de la zone de transit, utilisez la commande suivante:

Il est une bonne pratique de développement logiciel d'ajouter une note à la validation pour expliquer la raison des changements. Pour valider les modifications de la zone de transit avec un message, utilisez la commande suivante:

Si la commande git commit est exécutée sans contenu dans la zone de transit, Git retournera un message, et rien ne se passera au dépôt Git. Cette commande met à jour uniquement le référentiel Git avec le contenu de la zone de transit. Il ne prendra aucune modification du répertoire de travail.

Veuillez consulter la documentation officielle git commit pour plus de détails et les options de ligne de commande.

**Mise à jour du référentiel distant**

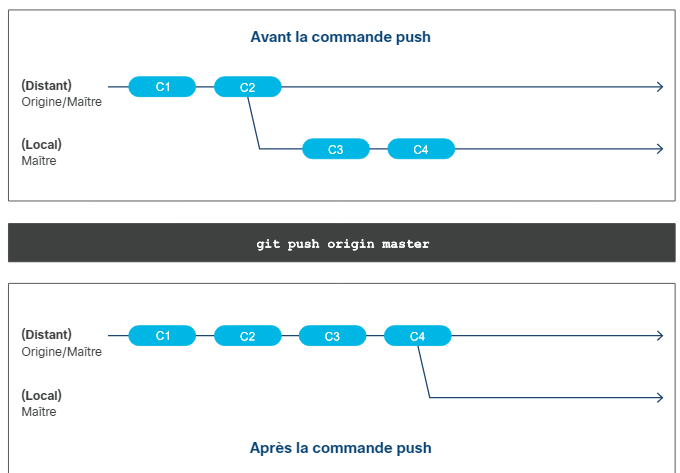
Afin de partager les modifications de contenu du dépôt Git local avec d'autres personnes, le dépôt Git distant doit être mis à jour manuellement. Git fournit une commande git push pour mettre à jour le dépôt Git distant avec les modifications de contenu du dépôt Git local.

**Commande** : git push

Cette commande ne s'exécutera pas correctement s'il y a un conflit avec l'ajout des modifications du dépôt Git local au dépôt Git distant. Des conflits se produisent lorsque deux personnes modifient la même partie du même fichier. Par exemple, si vous clonez le référentiel et que quelqu'un d'autre envoie les modifications avant vous, votre push peut créer un conflit. Les conflits doivent d'abord être résolus avant que le git push ne réussisse.

Pour mettre à jour le contenu du référentiel local vers une branche particulière du référentiel distant, utilisez la commande suivante:

Pour mettre à jour le contenu du référentiel local vers la branche maître du référentiel distant, utilisez la commande suivante: 



**Mise à jour de votre copie locale du référentiel**

Les copies locales du dépôt Git ne sont pas automatiquement mises à jour lorsqu'un autre contributeur effectue une mise à jour du dépôt Git distant. La mise à jour de la copie locale du référentiel est une étape manuelle. Git fournit une commande git pull pour obtenir les mises à jour d'une branche ou d'un référentiel. Cette commande peut également être utilisée pour intégrer la copie locale à une branche non parente.

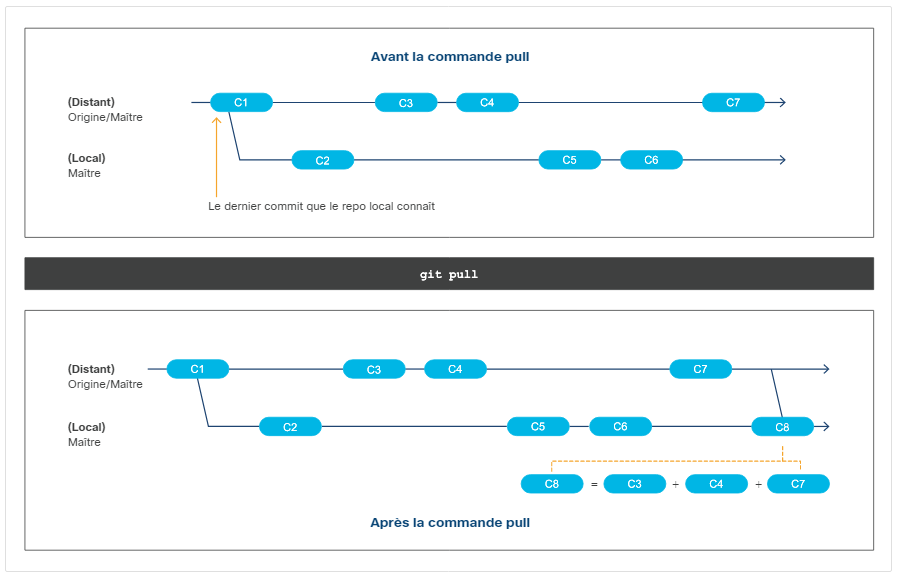
**Commande** : git pull

Pour plus de détails sur la commande git pull, lors de l'exécution de la commande, les étapes suivantes se produisent:

1. Le référentiel local (répertoire .git) est mis à jour avec le dernier commit, l'historique des fichiers, etc. à partir du dépôt Git distant. (Ceci est équivalent à la commande Git git fetch.)
2. Le répertoire de travail et la branche sont mis à jour avec le dernier contenu de l'étape 1. (Ceci est équivalent à la commande Git git merge.)
3. Un seul commit est créé sur la branche locale avec les modifications de l'étape 1. S'il y a un conflit de fusion, il devra être résolu.
4. Le répertoire de travail est mis à jour avec le contenu le plus récent.

Pour mettre à jour la copie locale du dépôt Git à partir de la branche parent, utilisez la commande suivante:

OU 



## Mise à jour des référentiels

**Création et suppression d'une branche**

Les branches sont une fonctionnalité très utile de Git. Comme indiqué précédemment, l'utilisation de branches présente de nombreux avantages, mais un avantage majeur est qu'elle permet aux fonctionnalités et aux modifications de code d'être effectuées indépendamment du code principal (la branche maître).

Il existe deux options pour créer une branche.

**OPTION 1**

Git fournit une commande git branch pour lister, créer ou supprimer une branche.

**Commande** : git branch

Pour créer une branche, utilisez la commande suivante:

où <parent branch> est la branche à débrancher et le<branch name> est le nom pour appeler la nouvelle branche.

Lorsque vous utilisez cette commande pour créer une branche, Git crée la branche, mais il ne bascule pas automatiquement le répertoire de travail vers cette branche. Vous devez utiliser la commande git switch <branch name> pour basculer le répertoire de travail vers la nouvelle branche.

**OPTION 2**

Git fournit une commande git checkout pour changer de branche en mettant à jour le répertoire de travail avec le contenu de la branche.

**Commande** : git checkout

Pour basculer vers une branche, utiliser la commande suivante



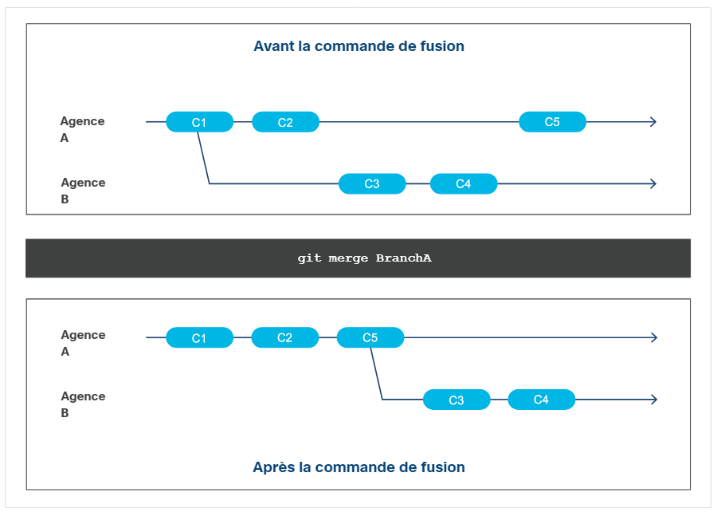
Pour créer une branche et basculer le répertoire de travail vers cette branche, utilisez la commande suivante:

où <parent branch> est la branche à débrancher et le<branch name> est le nom pour appeler la nouvelle branche.

**Suppression d'une branche**Pour supprimer une branche, utilisez la commande suivante:

**Obtenir une liste de toutes les branches**Pour obtenir une liste de toutes les branches locales, utilisez la commande suivante:Ou

**Fusionner des branches**

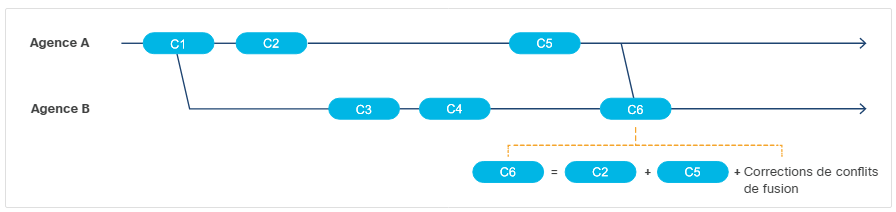
Les branches divergent les unes des autres lorsqu'elles sont modifiées après leur création. Pour obtenir les modifications d'une branche (source) à une autre (cible), vous devez fusionner la branche source dans la branche cible. Lorsque Git fusionne la branche, il prend les modifications/validations de la branche source et l'applique à la branche cible. Lors d'une fusion, seule la branche cible est modifiée. La branche source est intacte et reste la même.

1. Au commit #1, la Branche B s'est branchée de la Branche A.
2. Une fois les branches divergent, quelqu'un ajoute commit #2 à la branche A. La branche B n'obtient pas ces modifications.
3. Quelqu'un ajoute commit #3 et commit #4 à la branche B. La branche A n'obtient pas ces modifications.
4. Quelqu'un ajoute commit #5 à la branche A. La branche B n'obtient pas ces modifications.
5. Maintenant, la branche A et la branche B ont divergé par deux validations chacune.
6. Disons que la branche B veut les changements de la branche A car elle a divergé (commit #2 et commit #5). Ainsi, la branche A est la branche source et la branche B est la branche cible. Dans cet exemple, déclarons que les commits étaient des changements dans différents fichiers. Par conséquent, commit #2 et commit #5 sont appliqués à la branche B et la branche A reste la même. C'est ce qu'on appelle une fusion rapide.

**Fusion rapide**Une fusion rapide est lorsque l'algorithme Git est capable d'appliquer automatiquement et sans conflits les modifications/commits de la branche source à la branche cible. Ceci est généralement possible lorsque différents fichiers sont modifiés dans les branches en cours de fusion. Il est toujours possible lorsque le même fichier est modifié, mais généralement lorsque différentes lignes du fichier ont été modifiées. Une fusion rapide est le meilleur scénario lors de l'exécution d'une fusion.

Dans une fusion rapide, Git intègre les différents commits de la branche source dans la branche cible. Parce que les branches ne sont essentiellement que des pointeurs vers des validations dans le backend, une fusion rapide déplace simplement le pointeur qui représente la HEAD de la branche cible, plutôt que d'ajouter un nouveau commit.

Notez que pour faire une fusion rapide, Git doit pouvoir fusionner tous les validations existantes sans rencontrer de conflits.

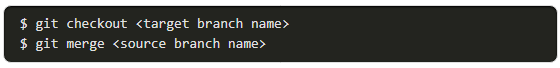
**Fusion des conflits**La modification du même fichier sur différentes branches à fusionner augmente les chances d'un conflit de fusion. Un conflit de fusion est lorsque Git n'est pas en mesure d'effectuer une fusion rapide car il ne sait pas comment appliquer automatiquement les modifications des branches ensemble pour le ou les fichiers. Lorsque cela se produit, l'utilisateur doit corriger manuellement ces conflits avant que les branches puissent être fusionnées ensemble. La correction manuelle du conflit ajoute une nouvelle validation à la branche cible contenant les validations de la branche source, ainsi que le ou les conflits de fusion fixes.

**Exécution de la fusion**Git fournit une commande git merge pour joindre deux ou plusieurs branches ensemble.

**Commande** : git merge  
Pour fusionner une branche dans la branche/référentiel actuelle du client, utilisez la commande suivante:

où <branch name> est la branche source qui est fusionnée dans la branche actuelle

Lorsque vous utilisez la commande git merge, la branche cible doit être la branche/référentiel en cours, donc pour fusionner une branche dans une branche qui n'est pas la branche/référentiel en cours du client, utilisez les commandes suivantes:

  
où<target branch name>est la branche cible et le<source branch name>est la branche source.

Pour fusionner plusieurs branches dans la branche/référentiel actuelle du client, utilisez la commande suivante:



## Fichiers .diff

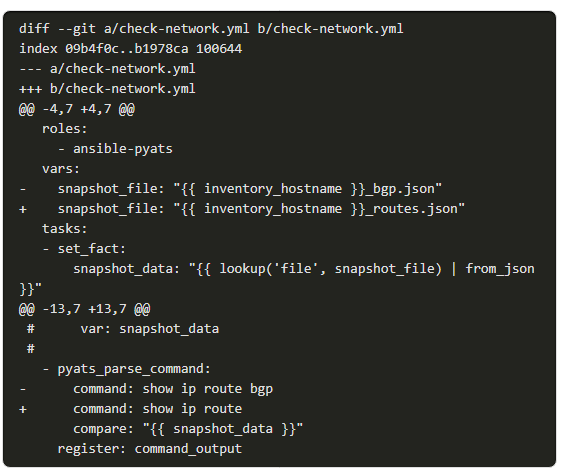
**Qu' est-ce qu'un fichier .diff?**

Les développeurs utilisent un fichier .diff pour montrer comment deux versions différentes d'un fichier ont changé. En utilisant des symboles spécifiques, ce fichier peut être lu par d'autres systèmes pour interpréter la manière dont les fichiers peuvent être mis à jour. La différence est utilisée pour implémenter les modifications en comparant et fusionnant les deux versions. Certains projets nécessitent que des modifications soient soumises via un fichier .diff sous forme de correctif. Parce que tout cela est dans un seul fichier, il est appelé diff unifié.

Les symboles et les significations d'un fichier diff unifié sont présentés ci-dessous:

* + : Indique que la ligne a été ajoutée.
* - : Indique que la ligne a été supprimée.
* /dev/null : Indique qu'un fichier a été ajouté ou supprimé.
* "blank": Donne des lignes de contexte autour des lignes modifiées.
* @@ : Indicateur visuel que le prochain bloc d'informations commence. Dans les modifications d'un fichier, il peut y avoir plusieurs.
* index : Affiche les validations comparées.

Exemple diff pour un fichier nommé check-network.yml:



Le signal peut être un "+" ou un "-" selon l'ordre des hachages.

Dans ce format, il y a trois lignes affichées au-dessus et au-dessous de la ligne modifiée exacte pour le contexte, mais vous pouvez repérer les différences en comparant la ligne - avec la ligne +. L'une des modifications apportées à ce correctif consiste à modifier le nom du fichier instantané, en remplaçant... bgp.json par... routes.json.



Vous pouvez toujours regarder la différence entre deux fichiers d'une requête GitHub Pull Request comme un diff unifié en ajoutant .diff à l'URL GitHub.