

SUPPORT GIT

GIT - VERSIONING

INTRODUCTION

DÉFINITION

Un gestionnaire de versions distribuées est un outil essentiel dans la programmation moderne, facilitant la collaboration sur les projets logiciels en conservant un historique de toutes les modifications apportées au code source. Les systèmes comme Git, Mercurial et Bazaar sont des exemples de ces outils.

COMPRENDRE LES SYSTÈMES DE CONTRÔLE DE VERSION

Un système de contrôle de version (VCS, Version Control System) est un outil qui aide les développeurs à suivre et à gérer les modifications apportées à un code source au fil du temps. Il conserve un historique de toutes les modifications, permettant ainsi aux développeurs de revenir à une version précédente si nécessaire. Il existe deux types principaux de VCS : centralisés et distribués.

SYSTÈMES DE CONTRÔLE DE VERSION CENTRALISÉS (CVCS)

Dans un CVCS, comme SVN ou CVS, il y a un seul "référentiel" central de tout le code et de son historique. Les développeurs obtiennent une copie de travail du code, apportent des modifications, puis "commettent" ces modifications dans le référentiel central.

SYSTÈMES DE CONTRÔLE DE VERSION DISTRIBUÉS (DVCS)

Dans un DVCS, comme Git, chaque développeur possède une copie complète du référentiel, y compris l'historique des modifications. Cela offre de nombreux avantages, comme la possibilité de travailler hors ligne, une réduction de la dépendance à un serveur central, et une flexibilité accrue pour les différentes méthodologies de développement.

GIT : UN EXEMPLE DE GESTIONNAIRE DE VERSIONS DISTRIBUÉES

Git est le DVCS le plus largement utilisé.

Il a été créé par Linus Torvalds pour le développement du noyau Linux.

CONCEPTS CLÉS DE GIT

- **Référentiel (repository)** : Un référentiel Git est un ensemble de fichiers et de dossiers que vous souhaitez suivre, dossiers dans lesquels vous effectuez des modifications, et que vous souhaitez enregistrer au fil du temps.
- **Commit** : Un commit est un instantané d'un ensemble de modifications apportées aux fichiers dans votre référentiel. Chaque commit est doté d'un identifiant unique (un "hash") que vous pouvez utiliser pour référencer ce commit spécifique.
- **Branches** : Les branches sont essentiellement des pointeurs vers un commit. Elles sont utilisées pour créer des flux de travail isolés sur le même référentiel, permettant aux développeurs de travailler sur différentes fonctionnalités simultanément sans interférer entre eux.
- **Merge** : L'opération de fusion permet de combiner les modifications de différentes branches ensemble.

WORKFLOW BASIQUE AVEC GIT

1. **Clone** : Cette opération vous permet d'obtenir une copie locale d'un référentiel existant.
2. **Pull** : Cette opération vous permet de récupérer les dernières modifications du référentiel distant.
3. **Branch** : Créez une nouvelle branche sur laquelle effectuer vos modifications.
4. **Checkout** : Cette commande vous permet de passer d'une branche à une autre.
5. **Add/Commit** : Ajoutez des fichiers à votre commit et enregistrez vos modifications dans le référentiel.
6. **Push** : Cette commande vous permet d'envoyer vos commits vers un référentiel distant. Cela permet de partager votre travail avec d'autres et de le sauvegarder dans un emplacement externe.
7. **Merge** : Si vous avez fini de travailler sur une branche et que vous souhaitez intégrer vos modifications dans la branche principale (souvent appelée 'master' ou 'main'), vous utiliserez la commande merge.
8. **Pull Request** : Dans certains systèmes comme GitHub, avant de fusionner votre branche avec la branche principale, vous pouvez créer une "Pull Request". C'est une façon de proposer vos modifications à l'équipe, de demander des commentaires et d'effectuer des révisions si nécessaire avant de fusionner.

AVANTAGES D'UN GESTIONNAIRE DE VERSIONS DISTRIBUÉES

- Il protège contre le fait qu'une seule machine devienne le SPOF (Point Unique De Défaillance)
- Les contributeurs du projet peuvent continuer à travailler sur leur projet sans avoir à être en ligne
- Il permet de participer à un projet sans nécessiter d'autorisations par un chef de projet (les droits de validation/soumission peuvent donc être donnés après avoir démontré son travail et pas avant)
- La plupart des opérations sont plus rapides car effectuées localement.
- Vous permet de créer des brouillons sans publier les modifications ni déranger les autres contributeurs.
- Permet de conserver un référentiel de référence contenant les versions livrées d'un projet.

LES OBJETS GIT

INTRODUCTION

Avant de commencer à utiliser git, nous allons d'abord explorer son fonctionnement interne. Comme vous le savez, Git est un outil de gestion de versions qui garde en mémoire toutes les modifications apportées aux fichiers, avec des informations telles que le message, la date et l'heure. Pour stocker de manière efficace ces fichiers et informations, Git utilise sa propre structure de fichiers basée sur les objets "blob", "tree" et "commit".

BLOB

Un objet "blob" est le plus simple des objets dans Git. Il représente un jeu de données. Dans la plupart des cas, cet ensemble de données est un fichier.

Lorsque vous ajoutez un fichier à votre dépôt Git et que vous validez cette modification, Git crée un objet blob qui contient les données de votre fichier. Chaque blob est identifié par une empreinte SHA-1, qui est générée à partir du contenu du blob.

EXEMPLE

Supposons que vous ayez un fichier appelé "hello.txt" avec le contenu "Hello, World!".

Lorsque vous ajoutez ce fichier à votre dépôt Git et que vous effectuez un commit, Git crée un blob pour ce fichier. Le blob contiendra les données "Hello, World!", et aura un hash SHA-1 unique basé sur ces données.

TREE

Un tree dans Git est un objet qui représente un répertoire dans votre système de fichiers. Il fait le lien entre le nom des fichiers et les blobs. Il peut également lier à d'autres trees pour représenter une structure de répertoire.

Chaque objet tree contient une ou plusieurs entrées, chacune d'elles étant une référence à un blob ou à un autre tree (pour les sous-répertoires), un nom de fichier, et des permissions de fichier.

EXEMPLE

Supposons que vous ayez un répertoire avec deux fichiers : "hello.txt" et "world.txt".

Lorsque vous ajoutez ces fichiers à votre dépôt Git et que vous effectuez un commit, Git crée un tree qui représente ce répertoire. Ce tree contiendra deux entrées : une pour "hello.txt" et une pour "world.txt". Chaque entrée pointera vers le blob correspondant à ce fichier et aura le nom du fichier.

COMMIT

Un commit est l'objet le plus complexe dans Git. Il représente un point dans l'histoire de votre projet.

Un commit pointe vers un tree qui représente l'état de votre dépôt à ce point précis. Il contient également des métadonnées, comme le nom de l'auteur, l'adresse e-mail, la date du commit, et un message de commit.

Un commit pointe également vers zéro ou plusieurs commits parents. Un commit initial n'a aucun parent, un commit normal en a un, et un commit qui est le résultat d'une fusion de deux branches peut avoir deux parents.

EXEMPLE

Supposons que vous avez ajouté plusieurs fichiers à votre dépôt Git et que vous êtes prêt à effectuer un commit.

Lorsque vous effectuez un commit, Git crée un objet commit qui contient vos informations (nom, e-mail), la date et heure actuelle, votre message de commit, et un pointeur vers la tree qui représente l'état actuel de votre dépôt. Si c'est votre premier commit, il n'aura pas de parents. Si ce n'est pas votre premier commit, il pointera vers le commit que vous avez effectué juste avant

RECAPITULATIF

BLOB

- Un "blob" (ou "binary large object") est un fichier qui est stocké dans git, il permet de stocker un contenu tel qu'une image ou un document.

ANALOGIE

Imaginez un blob comme un coffre-fort qui contient un précieux document (votre fichier). Une fois que le document est à l'intérieur, le coffre-fort est scellé et il ne peut plus être modifié. Le coffre-fort est ensuite marqué avec un numéro de série unique (l'empreinte SHA-1), basé sur le contenu du document à l'intérieur. Vous pouvez avoir de nombreux coffres-forts (blobs) avec différents documents, chacun avec son propre numéro de série unique.

TREE

- Un "tree" est comme une arborescence de fichier, il contient des références aux "blobs" et aux "trees" enfants, c'est un peu comme un dossier qui contient des fichiers et d'autres dossiers.

ANALOGIE

Maintenant, imaginez que vous avez une pièce (un répertoire) où vous stockez tous ces coffres-forts. Dans cette pièce, chaque coffre-fort est associé à une étiquette (le nom du fichier). C'est ce qu'est un tree. C'est comme une pièce qui contient plusieurs coffres-forts (blobs), chacun associé à une étiquette (nom de fichier). Et cette pièce peut aussi contenir d'autres petites pièces (sous-répertoires), qui ont aussi leurs propres coffres-forts et leurs propres étiquettes.

COMMIT

- Un "commit" est comme un instantané, il contient une référence à un "tree" particulier, ainsi que des informations comme la date, l'auteur, le message de commit. Cela permet de retrouver un état précis d'un projet à un moment donné.

COMMIT

Un commit est donc simplement un objet stockant :

- Une référence vers un tree
- Un message de commit
- Le nom de l'auteur du commit
- La date et l'heure du commit
- Une référence vers son commit parent (le commit qui le précède)

ANALOGIE

Enfin, imaginez un agent de sécurité qui fait des rondes dans le bâtiment où se trouve cette pièce. Chaque fois que l'agent fait sa ronde, il note l'état actuel de la pièce, y compris l'emplacement de chaque coffre-fort et de chaque étiquette, ainsi que son propre nom, la date et l'heure, et un bref rapport sur ce qui a changé depuis sa dernière ronde. C'est ce qu'est un commit. C'est comme un enregistrement de l'état de la pièce (l'état de votre projet) à un certain moment dans le temps, avec des métadonnées supplémentaires comme l'agent de sécurité (l'auteur du commit), la date et l'heure, et un rapport sur ce qui a changé (le message de commit).

VISUALISATION

SCHEMA

EXPLICATIONS

Dans la slide précédente, nous pouvons voir que chaque tree contient une référence vers un blob. Cependant, un tree peut également contenir une référence vers un autre tree. Cela permet de créer des dossiers contenant d'autres dossiers et fichiers.

Chaque objet git est stocké dans un fichier dont le nom est le hash SHA-1 de l'objet. Donc, les noms des fichiers sont stockés dans le tree, et les noms des dossiers sont stockés dans le tree parent.

SCHEMA

EXPLICATIONS

Si on modifie le contenu d'un fichier, Git va créer un nouveau blob avec le nouveau contenu, et modifier le tree pour pointer vers ce nouveau blob. puisque le contenu du tree a changé, Git va créer un nouveau tree avec les modifications. Et ce, pour tous ses parents.

SCHEMA

EXPLICATIONS

Un commit pointe donc vers un tree, qui pointe vers des blobs et d'autres trees.

C'est ainsi qu'il garde un snapshot de l'état du projet à un moment donné.

HEAD

Attardons maintenant notre attention sur le pointeur HEAD.

HEAD est un pointeur vers la branche courante.

Par défaut, HEAD pointe vers la branche main. La branche main est la branche principale du projet.

SCHEMA

EXPLICATIONS

Sur notre schéma, nous avons deux branches: MAIN et TEST.

HEAD pointe vers la branche TEST.

MAIN et TEST pointent vers le même commit.

Si nous faisons un commit sur la branche TEST, HEAD va pointer vers TEST, et TEST va pointer vers ce commit.

SCHEMA

EXPLICATIONS

Si nous nous déplaçons sur la branche MAIN, et faisons un commit, HEAD va pointer vers MAIN, et MAIN va pointer vers ce commit.

SCHEMA

INSTALLATION DE GIT

INSTALLATION WINDOWS

- Téléchargez le fichier d'installation de git pour Windows à partir du [site web de git](#)
- Exécutez le fichier d'installation et suivez les instructions à l'écran pour installer git.
- Vérifiez que git est installé correctement en ouvrant une invite de commande et en exécutant la commande `git --version`.

INSTALLATION MAC

- Téléchargez le fichier d'installation de git pour Mac à partir du [site web de git](#)
- Exécutez le fichier d'installation et suivez les instructions à l'écran pour installer git.
- Vérifiez que git est installé correctement en ouvrant Terminal et en exécutant la commande `git --version`.

INSTALLATION LINUX

- Utilisez votre gestionnaire de paquets préféré pour installer git. Les commandes ci-dessous montrent comment installer git en utilisant apt sur Ubuntu ou Debian :

```
sudo apt update  
sudo apt install git
```

- Vérifiez que git est installé correctement en ouvrant un terminal et en exécutant la commande git --version.

CONFIGURATION DE VOTRE IDENTITÉ

Une fois que Git est installé, vous devez configurer votre identité. Git utilise cette information pour associer vos commits à votre nom et votre adresse e-mail. Pour configurer votre nom, tapez :

```
git config --global user.name "Votre Nom"
```

Pour configurer votre adresse e-mail, tapez :

```
git config --global user.email "votre-email@example.com"
```

Remplacez "Votre Nom" et "**votre-email@example.com**" par votre nom et votre adresse e-mail.

VÉRIFICATION DE LA CONFIGURATION

Après avoir configuré Git, vous pouvez vérifier votre nom et votre adresse mail en utilisant la commande suivante :

```
git config --list
```

REFERENTIEL

CRÉER VOTRE PREMIER RÉFÉRENTIEL

Nous allons maintenant créer un nouveau référentiel git et y ajouter un fichier. Nous allons entrer dans les détails de chaque étape dans les prochaines sections.

INITIALISATION D'UN DÉPÔT

- Ouvrez un terminal ou une invite de commande et naviguez jusqu'au dossier de votre projet.
- Utilisez la commande `git init` pour initialiser un nouveau dépôt git dans ce dossier.

```
git init
```

- Vérifiez que le dépôt a été correctement initialisé en utilisant la commande `git status`. Vous devriez voir un message indiquant que le dépôt est vide.

AJOUT DE FICHIERS

- Ajoutez les fichiers de votre projet à l'index en utilisant la commande `git add` suivie du nom du fichier ou des fichiers. Par exemple :

```
git add main.c
```

- Vérifiez que les fichiers ont été ajoutés en utilisant la commande `git status`. Vous devriez voir une liste des fichiers ajoutés en rouge, signifiant qu'ils sont prêts à être commités.

EFFECTUER UN COMMIT

- Utilisez la commande `git commit` pour enregistrer les modifications dans le dépôt. Vous devriez inclure un message de commit expliquant les modifications apportées. Par exemple :

```
git commit -m "Initial commit"
```

- Vérifiez que le commit a été effectué en utilisant la commande `git log`. Vous devriez voir une entrée avec votre message de commit et les détails de l'auteur.

GIT ADD

Cette commande permet d'ajouter des fichiers ou des modifications à l'index, préparant les fichiers pour le prochain commit. Elle peut être utilisée de différentes manières.

GIT ADD

- Pour ajouter un fichier spécifique :

```
git add main.c
```

GIT ADD

- Pour ajouter tous les fichiers d'un répertoire :

```
git add myfolder/
```

GIT ADD

- Pour ajouter tous les fichiers modifiés ou nouveaux du répertoire courant :

```
git add .
```

Il est important de noter que l'ajout de fichiers à l'index ne les commit pas automatiquement, il faut utiliser la commande `git commit` pour enregistrer les modifications dans le dépôt.

INDEX/STAGING AREA

L'index de Git est un tampon entre votre répertoire de travail et le référentiel. Il permet de suivre les modifications de fichiers avant de les valider avec une commande `git commit`. Il est possible d'ajouter des fichiers à l'index avec la commande `git add` et de les retirer avec la commande `git reset`.

SCHÉMA INDEX/STAGING AREA

INDEX/STAGING AREA

AJOUTER DES FICHIERS À L'INDEX

```
$ git add file1.txt file2.txt
```

RETIKER DES FICHIERS DE L'INDEX

```
$ git reset file3.txt
```

INDEX/STAGING AREA

Points à retenir:

- L'index de Git est un tampon entre votre répertoire de travail et le référentiel.
- Il permet de suivre les modifications de fichiers avant de les valider avec une commande `git commit`.
- La commande `git add` permet d'ajouter des fichiers à l'index.
- La commande `git reset` permet de retirer des fichiers de l'index.

GIT COMMIT

Cette commande permet d'enregistrer les modifications de l'index dans le dépôt en créant un nouveau commit. Il est important de noter qu'il faut ajouter des fichiers à l'index avant de les commit, sinon, il n'y aura rien à commit.

GIT COMMIT

Il est important d'utiliser un message de commit clair et concis pour décrire les modifications apportées aux fichiers. Par exemple :

```
git commit -m "Ajout de la fonctionnalité X"
```

GIT COMMIT

Il est également possible de rajouter des commentaires additionnels dans un éditeur de texte pour écrire des commentaires plus étayés:

```
git commit
```

LA GESTION DES COMMITS

Il est important de rédiger des messages de commit significatifs qui décrivent clairement les modifications apportées dans chaque commit. Les messages de commit doivent être concis mais suffisamment détaillés pour que les autres développeurs puissent comprendre les modifications apportées et les raisons qui les ont motivées.

CONVENTIONS DE NOMMAGE DES COMMITS

- Utiliser un verbe à l'infinitif pour décrire l'action effectuée
- Utiliser des majuscules et des minuscules appropriées
- Eviter les phrases trop longues
- Utiliser des tirets pour séparer les mots
- Ajouter des références aux numéros de tickets de suivi de projet ou de bugs si nécessaire

FAIRE DES COMMITS ATOMIQUES

Il est important de faire des commits le plus atomique possible. Cela signifie que chaque commit devrait contenir des modifications liées à une seule fonctionnalité ou correction de bug. Cela permet de faciliter le suivi des modifications et de faciliter la résolution de problèmes éventuels.

FAIRE DES COMMITS FRÉQUEMMENT

Il est important de faire des commits fréquemment au lieu de faire des commits massifs. Cela permet de suivre l'historique des modifications de manière plus détaillée et de faciliter la résolution de problèmes.

UTILISER DES MESSAGES DE COMMIT SIGNIFICATIFS

Il est important de rédiger des messages de commit significatifs qui décrivent clairement les modifications apportées dans chaque commit. Les messages de commit doivent être concis mais suffisamment détaillés pour que les autres développeurs puissent comprendre les modifications apportées et les raisons qui les ont motivées.

QUELQUES EXEMPLES DE MESSAGES DE COMMIT SIGNIFICATIFS

```
git commit -m "Ajouter une fonctionnalité de recherche"  
git commit -m "Modifier le design de la page d'accueil"  
git commit -m "Supprimer les fichiers inutiles"  
git commit -m "Résoudre un bug de calcul de total"  
git commit -m "Ajouter une validation des champs de formulaire"  
git commit -m "Ajouter une documentation pour la fonctionnalité X"
```

CE QU'IL YA A RETENIR

- Il est important de faire des commits le plus atomique possible.
- Il est important de faire des commits fréquemment au lieu de faire des commits massifs.
- Il est important de rédiger des messages de commit significatifs qui décrivent clairement les modifications apportées dans chaque commit.

LES BRANCHES DANS GIT

INTRODUCTION

Les branches dans Git permettent de gérer différentes versions de votre projet de manière indépendante. Chaque branche peut avoir son propre historique de commits et de modifications, ce qui permet de travailler sur des fonctionnalités ou des correctifs de bugs sans affecter les autres branches.

CRÉER UNE NOUVELLE BRANCHE

- Utilisez la commande `git branch` suivie du nom de la nouvelle branche pour créer une nouvelle branche à partir de la branche courante. Par exemple :

```
git branch new-feature
```

Cela crée une nouvelle branche appelée "new-feature" à partir de la branche courante.

SUPPRIMER UNE BRANCHE EXISTANTE

- Utilisez la commande `git branch` avec l'option `--delete` pour supprimer une branche existante. Par exemple :

```
git branch --delete new-feature
```

PASSER À UNE BRANCHE EXISTANTE

- Utilisez la commande `git checkout` suivie du nom de la branche pour passer à une branche existante.

Par exemple :

```
git checkout new-feature
```

Vous changez ainsi branche courante pour "new-feature", vous permettant de travailler sur cette branche.

FUSIONNER UNE BRANCHE À L'AUTRE

- Utilisez la commande `git merge` suivie du nom de la branche à fusionner pour fusionner les modifications d'une branche à l'autre. Par exemple, pour fusionner les modifications de la branche "new-feature" dans la branche "master" :

```
git checkout master  
git merge new-feature
```

FUSIONNER UNE BRANCHE À L'AUTRE

Cela fusionne les modifications de la branche "new-feature" dans la branche "master". Il est important de noter que cela peut causer des conflits si les fichiers modifiés dans les deux branches sont différents. Il faudra alors les résoudre avant de continuer. Nous en verrons plus plus tard dans cette présentation.

CE QU'IL YA A RETENIR

- Les branches permettent de gérer différentes versions de votre projet de manière indépendante.
- Chaque branche a son propre historique de commits et modifications.
- Utilisez `git branch <nom-de-la-branche>` pour créer une nouvelle branche.
- Utilisez `git checkout <nom-de-la-branche>` pour passer à une branche existante.
- Utilisez `git merge <nom-de-la-branche>` pour fusionner les modifications d'une branche à l'autre.
- Utilisez `git branch -d <nom-de-la-branche>` pour supprimer une branche fusionnée, ou `git branch -D <nom-de-la-branche>` pour forcer la suppression d'une branche non fusionnée.

git rebase

rebase est une commande puissante qui vous permet de modifier votre base de code de manière propre et ordonnée. Elle vous permet de fusionner les commits et de vous déplacer d'une branche à une autre.

COMPRENDRE GIT REBASE

Prenons l'exemple de deux branches, **master** et **feature**. Vous avez effectué un travail sur la branche **feature** tandis que d'autres ont également effectué des modifications sur **master**. Maintenant, vous voulez mettre à jour votre branche **feature** avec les dernières modifications de **master**. Ici, vous pouvez utiliser **rebase**.

COMMENT UTILISER GIT REBASE

1. Commutez-vous sur la branche que vous souhaitez réorganiser. Dans notre exemple, ce serait **feature**.

```
$ git checkout feature
```

1. Exécutez la commande **rebase** avec la branche à partir de laquelle vous souhaitez appliquer les modifications. Dans notre exemple, ce serait **master**.

```
$ git rebase master
```

Cela prend tous les commits de la branche **feature** et les "rejoue" sur la branche **master**. Si des conflits se produisent pendant le rebase, Git vous le signalera et vous demandera de les résoudre avant de continuer.

RÉSOLUTION DES CONFLITS DE REBASE

Si vous rencontrez des conflits pendant un rebase, vous devez les résoudre manuellement. Git vous indique quel fichier est en conflit. Une fois que vous avez résolu ces conflits, vous pouvez continuer le rebase avec **git rebase --continue**.

Si vous voulez arrêter le processus de rebase, vous pouvez utiliser **git rebase --abort**.

GIT REBASE INTERACTIF

git rebase -i ou **git rebase --interactive** vous permet de modifier les commits alors qu'ils sont appliqués. Par exemple, si vous voulez modifier le message de commit, fusionner plusieurs commits en un seul ou supprimer certains commits, vous pouvez le faire avec un rebase interactif.

ATTENTION AVEC LE REBASE !

Rebase est une méthode de réécriture de l'historique, il faut donc être prudent lorsqu'on l'utilise sur des branches partagées avec d'autres développeurs. Si vous rebasé et forcez le push sur une branche partagée, cela pourrait perturber le travail des autres. Vous devriez donc éviter de faire un rebase sur des branches qui ne sont pas uniquement locales à votre répertoire.

TRAVAILLER EN ÉQUIPE VIA UN RÉPERTOIRE DISTANT

INTRODUCTION À GIT FETCH

- La commande `git fetch` permet de récupérer les commits d'un dépôt distant dans votre dépôt local.
- `git fetch` ne fusionne pas automatiquement les commits récupérés dans votre branche locale, il faut utiliser `git merge` ou `git cherry-pick` pour cela.

RÉCUPÉRER UN RÉPERTOIRE DISTANT AVEC GIT FETCH

- Ajouter le dépôt distant à votre projet local avec la commande `git remote add`
- Récupérer les commits du dépôt distant avec `git fetch <nom_depot>`
- Vérifier les commits récupérés avec `git log <nom_depot>/<nom_branche>`

FUSIONNER LES COMMITS RÉCUPÉRÉS DANS VOTRE BRANCHE LOCALE

Une fois les commits récupérés avec `git fetch`, il y a deux options pour les intégrer à votre projet.

L'une est de fusionner automatiquement tous les commits récupérés dans votre branche locale avec la commande `git merge`, l'autre est de sélectionner les commits spécifiques que vous souhaitez intégrer à votre branche locale avec la commande `git cherry-pick`.

FUSIONNER LES COMMITS RÉCUPÉRÉS DANS VOTRE BRANCHE LOCALE

- Utiliser la commande `git merge` pour fusionner automatiquement tous les commits récupérés dans votre branche locale:

```
git merge <nom_depot>/<nom_branche>
```

FUSIONNER LES COMMITS RÉCUPÉRÉS DANS VOTRE BRANCHE LOCALE

- Utiliser la commande `git cherry-pick` pour sélectionner les commits spécifiques que vous souhaitez intégrer à votre branche locale :

```
git cherry-pick <SHA1> <SHA2> <SHA3> ...
```

Les SHA1, SHA2, SHA3 sont les identifiants des commits que vous souhaitez intégrer à votre branche locale.

CE QU'IL YA A RETENIR

- La commande `git fetch` permet de récupérer les commits d'un dépôt distant dans votre dépôt local.
- `git fetch` ne fusionne pas automatiquement les commits récupérés dans votre branche locale, il faut utiliser `git merge` ou `git cherry-pick` pour cela.
- Utilisez `git remote add <nom_depot> <url_depot>` pour ajouter un dépôt distant à votre projet local.
- Utilisez `git fetch <nom_depot>` pour récupérer les commits d'un dépôt distant dans votre dépôt local.
- Utilisez `git log <nom_depot>/<nom_branche>` pour vérifier les commits récupérés.
- Utilisez `git merge <nom_depot>/<nom_branche>` pour fusionner automatiquement tous les commits récupérés dans votre branche locale.

git clone

La commande `git clone` permet de créer une copie exacte d'un dépôt distant sur votre ordinateur. Elle permet de télécharger tout le contenu du dépôt, y compris l'historique des commits, les branches et les étiquettes.

git clone

Pour utiliser `git clone`, il suffit de spécifier l'URL du dépôt distant que vous souhaitez cloner :

```
git clone <url_depot>
```

git clone

Par défaut, `git clone` crée un nouveau répertoire qui porte le nom du dépôt distant, et place tout le contenu du dépôt dans ce répertoire. Il est également possible de spécifier un nom de répertoire pour la copie locale du dépôt distant :

```
git clone <url_depot> <nom_reperoire>
```

`git clone` va également créer une copie locale de la branche principale (généralement la branche `main`) et la rendre active.

git pull

La commande `git pull` permet de récupérer les commits d'un dépôt distant et de les fusionner automatiquement dans votre branche locale. Elle est utilisée pour synchroniser votre dépôt local avec le dépôt distant.

git pull

Pour utiliser `git pull`, il faut d'abord ajouter le dépôt distant à votre projet local avec la commande `git remote add`, puis utiliser `git pull` suivi du nom du dépôt distant et de la branche que vous souhaitez récupérer :

```
git pull <nom_depot> <nom_branche>
```

git pull

git pull est en fait un raccourci pour les commandes git fetch suivi de git merge. Il permet donc de récupérer les commits d'un dépôt distant et de les fusionner automatiquement dans votre branche locale en un seul pas.

git pull

Il est important de noter que `git pull` peut causer des conflits si des modifications en conflit ont été apportées sur le dépôt distant et votre branche locale. Il est donc important de résoudre les conflits éventuels avant de valider la fusion.

CE QU'IL YA A RETENIR

- La commande `git clone` permet de créer une copie exacte d'un dépôt distant sur votre ordinateur.
- La commande `git pull` permet de récupérer les commits d'un dépôt distant et de les fusionner automatiquement dans votre branche locale.
- `git pull` est en fait un raccourci pour les commandes `git fetch` suivi de `git merge`.
- Vous devrez résoudre les conflits éventuels avant de valider la fusion.

LES STRATÉGIES DE FUSION

FAST-FORWARD MERGE

Lorsqu'on parle de fusion (merge) dans Git, plusieurs stratégies peuvent être utilisées. L'une d'entre elles est le "fast-forward merge".

QU'EST-CE QU'UN FAST-FORWARD MERGE ?

Un "fast-forward merge" est possible lorsque la branche sur laquelle vous voulez fusionner n'a pas de nouveaux commits depuis la création de la branche que vous voulez y fusionner. En d'autres termes, si aucune modification n'a été apportée à la branche de base pendant que vous travaillez sur une autre branche, Git peut effectuer une fusion "fast-forward".

Dans une fusion "fast-forward", au lieu de créer un nouveau commit de fusion, Git déplace simplement (ou "avance rapidement") la référence de la branche de base vers le dernier commit de la branche que vous fusionnez.

COMMENT EFFECTUER UN FAST-FORWARD MERGE ?

Supposons que vous ayez une branche **main** et que vous ayez créé une nouvelle branche **feature** à partir de celle-ci pour travailler sur une nouvelle fonctionnalité. Après avoir effectué plusieurs commits sur la branche **feature**, vous êtes prêt à fusionner cette branche dans **main**.

Si aucun nouveau commit n'a été effectué sur **main** depuis la création de **feature**, vous pouvez effectuer un "fast-forward merge" en suivant ces étapes :

1. Assurez-vous d'être sur la branche **main** :

```
git checkout main
```

2. Fusionnez la branche **feature** :

```
git merge feature
```

Si une fusion "fast-forward" est possible, Git la fera automatiquement. Vous verrez un message indiquant que la fusion a été effectuée en mode "fast-forward" et la référence de la branche **main** sera déplacée vers le dernier commit de la branche **feature**.

QUE FAIRE SI VOUS VOULEZ ÉVITER UN FAST-FORWARD MERGE ?

Il peut y avoir des cas où vous ne voulez pas d'un "fast-forward merge", par exemple pour conserver explicitement l'historique de la branche dans laquelle vous avez travaillé. Dans ce cas, vous pouvez forcer Git à créer un commit de fusion en utilisant l'option **--no-ff** lors de la fusion :

```
git merge --no-ff feature
```

Cela créera un nouveau commit de fusion même si une fusion "fast-forward" aurait été possible. Cela force le "three way merge".

CONCLUSION

Le "fast-forward merge" est une stratégie de fusion efficace qui simplifie l'historique de votre projet lorsque cela est possible. Cependant, il est important de comprendre quand et comment l'utiliser pour gérer efficacement vos projets avec Git.

THREE-WAY MERGE

Le Three-Way Merge est une stratégie de fusion que Git utilise lorsque deux branches ont divergé, c'est-à-dire lorsque des modifications ont été apportées sur les deux branches après leur point de divergence commun.

Dans un Three-Way Merge, Git utilise trois points : le dernier commit de la branche que vous êtes en train de fusionner (appelez-le B), le dernier commit de la branche dans laquelle vous fusionnez (appelez-le A) et leur ancêtre commun le plus proche (appelez-le C). Git compare les modifications de A à C et de B à C, et tente de résoudre automatiquement les modifications qui ne sont pas en conflit.

Si Git peut résoudre tous les conflits, il crée un nouveau commit qui contient le résultat de la fusion. Ce commit a deux parents : A et B. Si Git ne peut pas résoudre tous les conflits, il vous demande de les résoudre manuellement.

COMMENT EFFECTUER UN THREE-WAY MERGE ?

Supposons que vous ayez une branche **main** et une branche **feature** qui ont toutes deux été modifiées depuis leur point de divergence commun. Vous voulez fusionner **feature** dans **main** en utilisant un Three-Way Merge. Voici comment vous pouvez le faire :

1. Assurez-vous d'être sur la branche **main** :

```
git checkout main
```

2. Fusionnez la branche **feature** :

```
git merge feature
```

Si une fusion "fast-forward" n'est pas possible, Git effectuera automatiquement un Three-Way Merge.

COMMENT RÉSOUDRE LES CONFLITS DE FUSION ?

<>

Si Git ne peut pas résoudre automatiquement tous les conflits lors d'une fusion, vous devrez les résoudre manuellement. Voici comment vous pouvez le faire :

1. Pour voir quels fichiers ont des conflits, utilisez la commande suivante :

```
git status
```

2. Ouvrez chaque fichier avec des conflits dans un éditeur de texte. Vous verrez des blocs de conflit marqués par <<<<<<, ===== et >>>>>. Le contenu entre <<<<<< et ===== vient de la branche **HEAD** (la branche dans laquelle vous fusionnez), et le contenu entre ===== et >>>>> vient de la branche que vous fusionnez.
3. Pour chaque bloc de conflit, décidez si vous voulez garder le contenu de **HEAD**, le contenu de la branche que vous fusionnez, ou une combinaison des deux. Modifiez le fichier pour qu'il contienne le contenu final que vous voulez, et supprimez les marqueurs de conflit.
4. Une fois que vous avez résolu tous les conflits dans un fichier, ajoutez le fichier à l'index avec la commande suivante :

```
git add <filename>
```

 salesforce authorized training provider



5. Répétez les étapes 2 à 4 pour tous les fichiers en conflit.

6. Une fois que tous les conflits ont été résolus et que tous les fichiers ont été ajoutés à l'index, validez la

DIFFÉRENCE CONCRÈTE ENTRE FAST-FORWARD ET THREE-WAY

La différence entre un "fast-forward merge" et un "three-way merge" dans Git réside principalement dans la façon dont les commits sont traités lors de la fusion et dans l'état de l'historique de commit après la fusion.

FAST-FORWARD MERGE

Dans un scénario de "fast-forward merge", la branche que vous souhaitez fusionner n'a pas divergé de la branche cible (aucune modification n'a été faite sur la branche cible depuis la création de la branche source). Git peut donc simplement "avancer" la référence de la branche cible pour pointer vers le dernier commit de la branche source, comme si les commits avaient été faits directement sur la branche cible. C'est pourquoi on l'appelle "fast-forward" : Git avance simplement la branche cible.

EXEMPLE DE FAST-FORWARD MERGE



Après la fusion "fast-forward", cela ressemblera à ceci :

A - B - C - D - E (main, feature)

THREE-WAY MERGE

Dans un scénario de "three-way merge", des modifications ont été faites à la fois sur la branche cible et sur la branche source depuis leur point de divergence commun. Dans ce cas, Git ne peut pas simplement avancer la branche cible car cela écrasera les modifications faites sur celle-ci. À la place, Git crée un nouveau commit qui combine les modifications des deux branches. Ce commit de fusion a deux parents : le dernier commit de chaque branche. C'est pourquoi on l'appelle "three-way merge" : Git utilise trois commits (les deux derniers commits et leur ancêtre commun) pour créer le commit de fusion.

EXEMPLE DE THREE-WAY MERGE



Après la fusion "three-way", cela ressemblera à ceci :



Dans cet exemple, **G** est le commit de fusion.

LES TAGS

L'UTILISATION DES TAGS

Les tags dans Git permettent de marquer des versions spécifiques de votre projet. Ils peuvent être utilisés pour marquer des versions stables qui ont été testées et approuvées pour une mise en production, ou pour marquer des versions importantes dans l'historique de développement de votre projet.

Il existe deux types de tags dans Git : les tags légers et les tags annotés.

TAG LÉGER

Un tag léger est simplement un alias pour un commit spécifique. Il est créé en utilisant la commande `git tag` suivi du nom du tag et de l'identifiant de commit :

```
git tag <nom_tag> <identifiant_commit>
```

TAG ANNOTÉ

Un tag annoté est similaire à un tag léger, mais il contient également des informations supplémentaires telles qu'un message de tag et l'identité de la personne qui a créé le tag. Les tags annotés sont créés en utilisant la commande `git tag` avec l'option `-a` suivie du nom du tag, de l'identifiant de commit et d'un message de tag :

```
git tag -a <nom_tag> -m <message_tag> <identifiant_commit>
```

POUSSER UN TAG

Une fois créés, les tags peuvent être poussés sur le dépôt distant en utilisant la commande `git push`:

```
git push <nom_depot> <nom_tag>
```

POUSSER TOUS LES TAGS

Il est également possible de pousser tous les tags d'un coup :

```
git push <nom_depot> --tags
```

SUPPRIMER UN TAG

Pour supprimer un tag, vous pouvez utiliser la commande `git tag` avec l'option `-d` :

```
git tag -d <nom_tag>
```

SUPPRIMER UN TAG SUR LE DÉPÔT DISTANT

Pour supprimer un tag sur le dépôt distant, vous pouvez utiliser la commande `git push` avec l'option `:refs/tags/<nom_tag>`:

```
git push <nom_depot> :refs/tags/<nom_tag>
```

CE QU'IL YA A RETENIR

- Les tags dans Git permettent de marquer des versions spécifiques de votre projet.
- Il existe deux types de tags dans Git : les tags légers et les tags annotés.
- Les tags annotés sont créés en utilisant la commande `git tag` avec l'option `a` suivie du nom du tag, de l'identifiant de commit et d'un message de tag.
- Une fois créés, les tags peuvent être poussés sur le dépôt distant en utilisant la commande `git push`.

CRÉATION ET APPLICATION DE PATCH

QU'EST-CE QU'UN PATCH ?

Un patch est un fichier qui contient les différences (ou les "patches") entre deux versions d'un fichier ou d'un ensemble de fichiers. Il est généralement utilisé pour partager des modifications avec d'autres développeurs ou pour appliquer des modifications à des fichiers qui ne sont pas sous le contrôle de version.

CRÉER ET APPLIQUER UN PATCH

Il existe plusieurs manières de créer et d'appliquer des patches dans Git. La commande `git diff` permet de générer un patch à partir des modifications non encore committées d'un fichier ou d'un ensemble de fichiers. La commande `git format-patch` permet de générer des patches à partir de commits spécifiques.

CRÉER ET APPLIQUER UN PATCH

Pour appliquer un patch, vous pouvez utiliser la commande `git apply`. Il est également possible d'utiliser `git am` pour appliquer un patch et l'intégrer directement dans le dépôt en créant un commit.

COMMENT CRÉER UN PATCH ?

- Utilisez la commande `git diff > nom_du_patch.patch` pour générer un patch à partir des modifications non encore commitées d'un fichier ou d'un ensemble de fichiers.

```
git diff > mon_patch.patch
```

COMMENT CRÉER UN PATCH À PARTIR DE COMMITS SPÉCIFIQUES ?

- Utilisez la commande `git format-patch [nom_du_commit]` pour générer des patches à partir de commits spécifiques.

```
git format-patch HEAD~3..HEAD
```

COMMENT APPLIQUER UN PATCH ?

- Utilisez la commande `git apply nom_du_patch.patch` pour appliquer un patch.

```
git apply mon_patch.patch
```

COMMENT APPLIQUER UN PATCH ET L'INTÉGRER DIRECTEMENT DANS LE DÉPÔT ?

- Utilisez la commande `git am nom_du_patch.patch` pour appliquer un patch et l'intégrer directement dans le dépôt en créant un commit.

```
git am mon_patch.patch
```

CE QU'IL YA A RETENIR

- Un patch est un fichier qui contient les différences (ou les "patches") entre deux versions d'un fichier ou d'un ensemble de fichiers.
- La commande `git diff` permet de générer un patch à partir des modifications non encore commitées d'un fichier ou d'un ensemble de fichiers.
- La commande `git format-patch` permet de générer des patches à partir de commits spécifiques.
- La commande `git apply` permet d'appliquer un patch.
- La commande `git am` permet d'appliquer un patch et l'intégrer directement dans le dépôt en créant un commit.

RECHERCHE DICHOTOMIQUE

QU'EST-CE QUE LA RECHERCHE DICHOTOMIQUE AVEC GIT BISECT ?

La recherche dichotomique avec git bisect est un outil intégré à Git qui permet de trouver rapidement quel commit a introduit un bug ou une régression dans votre code. Il fonctionne en divisant la recherche en deux parties à chaque étape, jusqu'à ce qu'il trouve le commit spécifique qui a introduit le bug.

COMMENT UTILISER GIT BISECT ?

Pour utiliser git bisect, vous devez d'abord définir un état "good" (sans bug) et un état "bad" (avec bug) dans votre dépôt. Git bisect va alors automatiquement sélectionner un commit au milieu de l'intervalle entre ces deux états, et vous demander de vérifier si ce commit est "good" ou "bad". Selon votre réponse, git bisect va continuer à diviser l'intervalle de recherche en deux, jusqu'à ce qu'il trouve le commit spécifique qui a introduit le bug.

COMMENT UTILISER GIT BISECT ?

Pour utiliser git bisect, vous devez d'abord vous rendre dans votre dépôt local :

```
cd mon_dépôt
```

COMMENT UTILISER GIT BISECT ?

Ensuite, vous devez lancer la commande git bisect pour démarrer la recherche dichotomique

```
git bisect start
```

COMMENT UTILISER GIT BISECT ?

Ensuite vous devez définir l'état "bad" (où le bug est présent) en utilisant la commande :

```
git bisect bad
```

COMMENT UTILISER GIT BISECT ?

Et enfin vous devez définir l'état "good" (où le bug n'est pas présent) en utilisant la commande :

```
git bisect good [nom_du_commit]
```

COMMENT UTILISER GIT BISECT ?

Git bisect va alors automatiquement sélectionner un commit au milieu de l'intervalle entre ces deux états, et vous demander de vérifier si ce commit est "good" ou "bad". Selon votre réponse, git bisect va continuer à diviser l'intervalle de recherche en deux, jusqu'à ce qu'il trouve le commit spécifique qui a introduit le bug.

CE QU'IL YA A RETENIR

- La recherche dichotomique avec git bisect est un outil intégré à Git qui permet de trouver rapidement quel commit a introduit un bug ou une régression dans votre code.
- Pour utiliser git bisect, vous devez d'abord définir un état "good" (sans bug) et un état "bad" (avec bug) dans votre dépôt.
- Git bisect va alors automatiquement sélectionner un commit au milieu de l'intervalle entre ces deux états, et vous demander de vérifier si ce commit est "good" ou "bad". Selon votre réponse, git bisect va continuer à diviser l'intervalle de recherche en deux, jusqu'à ce qu'il trouve le commit spécifique qui a introduit le bug.

PATTERNS ET WORKFLOWS

LE GIT FLOW

QU'EST-CE QUE LE GIT FLOW ?

Le Git flow est un type de workflow qui se base sur l'utilisation de branches spécifiques pour gérer les différentes étapes de développement d'un projet. Il a été développé par Vincent Driessen pour gérer les flux de travail d'équipes de développement. Il est souvent utilisé pour les projets de grande taille ou les projets avec plusieurs équipes de développement simultanément.

COMMENT FONCTIONNE LE GIT FLOW ?

Avec le Git flow, il y a une branche principale `main` qui accueille les versions stables de votre projet et une branche `develop` qui accueille les modifications en cours de développement. Il y a également des branches pour les fonctionnalités, les corrections de bugs et les versions de production. Les modifications sont intégrées dans ces branches spécifiques et ensuite fusionnées dans la branche principale après validation.

Il est important de noter que le Git flow nécessite une bonne organisation et une bonne communication entre les membres de l'équipe pour fonctionner efficacement.

AVANTAGES

- Organisation : Le Git flow permet une bonne organisation des modifications et une bonne visibilité sur l'état de développement d'un projet.
- Séparation des modifications : Il permet une séparation des modifications entre les fonctionnalités, les corrections de bugs et les versions de production.
- Possibilité de gérer plusieurs équipes : Il est adapté pour les projets de grande taille ou les projets avec plusieurs équipes de développement simultanément.

INCONVÉNIENTS

- Complexité : Il nécessite une bonne compréhension des branches et des merge pour être utilisé efficacement, il est donc plus complexe à mettre en place que d'autres types de workflow.
- Communication et coordination : Il nécessite une bonne communication et coordination entre les membres de l'équipe pour fonctionner efficacement.

QUAND UTILISER LE GIT FLOW ?

- Lorsque vous travaillez sur un projet de grande taille ou avec plusieurs équipes de développement simultanément.
- Lorsque vous voulez une bonne organisation des modifications et une bonne visibilité sur l'état de développement d'un projet.
- Lorsque vous voulez une séparation des modifications entre les fonctionnalités, les corrections de bugs et les versions de production.

QUAND ÉVITER LE GIT FLOW ?

- Lorsque vous travaillez sur un projet de petite ou moyenne taille, il est préférable d'utiliser un autre type de workflow comme un workflow basé sur le tronc pour éviter la complexité supplémentaire liée à la gestion des branches.
- Lorsque vous voulez éviter la complexité supplémentaire liée à la gestion des branches et des merge.
- Lorsque la communication et la coordination entre les membres de l'équipe sont difficiles.

CE QU'IL Y A A RETENIR :

- Le Git flow est un type de workflow qui se base sur l'utilisation de branches spécifiques pour gérer les différentes étapes de développement d'un projet.
- Il permet une bonne organisation des modifications et une bonne visibilité sur l'état de développement d'un projet.
- Il permet une séparation des modifications entre les fonctionnalités, les corrections de bugs et les versions de production.
- Il est adapté pour les projets de grande taille ou les projets avec plusieurs équipes de développement simultanément.
- Il nécessite une bonne compréhension des branches et des merge pour être utilisé efficacement, il est donc plus complexe à mettre en place que d'autres types de workflow.
- Il nécessite une bonne communication et coordination entre les membres de l'équipe pour fonctionner efficacement.
- Il n'est pas adapté pour les projets de petite ou moyenne taille, ou lorsque la communication et la coordination entre les membres de l'équipe sont difficiles.

TRUNK BASED WORKFLOW

QU'EST-CE QU'UN WORKFLOW BASÉ SUR LE TRONC (TRUNK-BASED WORKFLOW) ?

Un workflow basé sur le tronc (ou trunk-based workflow) est un type de workflow qui se base sur l'utilisation d'une seule branche principale (ou tronc) pour accueillir toutes les modifications. Il est souvent utilisé pour les projets de petite ou moyenne taille, ou pour les projets qui nécessitent une intégration rapide des modifications.

QU'EST-CE QU'UN WORKFLOW BASÉ SUR LE TRONC (TRUNK-BASED WORKFLOW) ?

Avec un workflow basé sur le tronc, tous les développeurs travaillent sur la même branche principale, et les modifications sont intégrées directement dans cette branche, sans passer par des branches de fonctionnalités ou des branches de bugfix. Cela permet une intégration rapide des modifications, mais il est important de noter qu'il n'y a pas de séparation des modifications et qu'il peut y avoir des risques de conflits. Il est possible de mettre en place des stratégies de validation des modifications avant intégration pour minimiser les risques de conflits.

AVANTAGES

- Intégration rapide des modifications : Les modifications sont intégrées directement dans la branche principale, sans passer par des branches de fonctionnalités ou des branches de bugfix.
- Simplicité : Ce type de workflow est plus simple à mettre en place et à comprendre que d'autres types de workflow.
- S'intègre bien avec les outils de CI/CD : Ce type de workflow s'intègre bien avec les outils de CI/CD, car il n'y a pas de branches de fonctionnalités ou de branches de bugfix.

INCONVÉNIENTS

- Risques de conflits : Il n'y a pas de séparation des modifications, il y a donc des risques de conflits.
- Pas de séparation des modifications : Il n'y a pas de séparation des modifications, ce qui peut rendre difficile de comprendre les modifications apportées.

QUAND UTILISER UN WORKFLOW BASÉ SUR LE TRONC ?

- Lorsque vous travaillez sur un projet de petite ou moyenne taille.
- Lorsque vous voulez une intégration rapide des modifications.

QUAND ÉVITER UN WORKFLOW BASÉ SUR LE TRONC ?

- Lorsque vous travaillez sur un projet de grande taille ou avec plusieurs équipes de développement simultanément, il est préférable d'utiliser un autre type de workflow comme un Gitflow ou un GitHub flow pour minimiser les risques de conflits.
- Lorsque vous voulez une séparation des modifications pour faciliter la compréhension des modifications apportées.

RÉSUMÉ DES ÉLÉMENTS À RETENIR :

- Le workflow basé sur le tronc est un type de workflow qui se base sur l'utilisation d'une seule branche principale pour accueillir toutes les modifications.
- Il permet une intégration rapide des modifications, mais il y a des risques de conflits.
- Il est important de mettre en place des stratégies de validation des modifications pour minimiser les risques de conflits.
- Il est adapté pour les projets de petite ou moyenne taille ou pour les projets qui nécessitent une intégration rapide des modifications
- Il est plus simple à mettre en place et à comprendre que d'autres types de workflow.
- Il n'y a pas de séparation des modifications, ce qui peut rendre difficile de comprendre les modifications apportées.

FORK BASED WORKFLOW

QU'EST-CE QU'UN FORK-BASED WORKFLOW ?

Un fork-based workflow est un type de workflow utilisé lorsque vous travaillez sur un projet open-source ou lorsque vous voulez utiliser le code d'un autre projet dans le vôtre, sans pour autant le modifier directement. Il se base sur l'utilisation de "forks" (ou copies) de dépôts distants.

QU'EST-CE QU'UN FORK-BASED WORKFLOW ?

Avec un fork-based workflow, vous allez créer une copie (ou fork) du dépôt distant sur lequel vous voulez travailler. Vous pourrez alors y apporter des modifications et envoyer des "pull requests" pour intégrer vos modifications dans le dépôt original. Cela permet aux mainteneurs du dépôt d'accepter ou de refuser les modifications proposées.

Il est important de noter que ce type de workflow est plus adapté aux projets open-source ou aux projets qui nécessitent une validation avant intégration des modifications.

AVANTAGES

- Séparation des modifications : Vous pouvez effectuer vos modifications sur votre fork sans affecter le dépôt original.
- Contrôle des modifications : Les mainteneurs du dépôt peuvent valider les modifications avant de les intégrer dans le dépôt original.

INCONVÉNIENTS

- Complexité supplémentaire : Ce type de workflow est plus complexe que les autres, il nécessite une bonne compréhension des notions de fork et de pull request.
- Temps supplémentaire : Les modifications doivent être approuvées avant d'être intégrées, il faut donc prévoir un temps supplémentaire pour cette validation.

QUAND UTILISER UN FORK-BASED WORKFLOW ?

- Lorsque vous travaillez sur un projet open-source ou lorsque vous voulez utiliser le code d'un autre projet dans le vôtre sans le modifier directement.
- Lorsque vous voulez assurer un niveau de qualité élevé en validant les modifications avant de les intégrer dans le dépôt original.

QUAND ÉVITER UN FORK-BASED WORKFLOW ?

- Lorsque vous travaillez sur un projet personnel ou dans une équipe fermée, il est préférable d'utiliser un autre type de workflow comme un Gitflow ou un GitHub flow.
- Lorsque vous voulez éviter la complexité supplémentaire liée à la gestion des forks et des pull requests.

CE QU'IL Y A À RETENIR :

- Le fork-based workflow est un type de workflow utilisé lorsque vous travaillez sur un projet open-source ou lorsque vous voulez utiliser le code d'un autre projet dans le vôtre, sans pour autant le modifier directement.
- Il se base sur l'utilisation de "forks" (ou copies) de dépôts distants et permet de séparer les modifications et de les valider avant de les intégrer dans le dépôt original.
- Ce type de workflow est plus adapté aux projets open-source ou aux projets qui nécessitent une validation avant intégration des modifications.
- Il nécessite une bonne compréhension des notions de fork et de pull request, et peut être plus complexe et prendre plus de temps que d'autres types de workflow.