Git

SEPTEMBRE-OCTOBRE 2025

Programme

01 **Configuration**

02 **Versionner**

03 **Collaborer**

04 **Pour aller (un peu) plus loin**

01

Configuration (purement git)

Configuration

Installation de Git

Configuration

- Bien que la configuration par défaut soit souvent adéquate, il est nécessaire d'ajuster certaines informations personnelles. Ces détails apparaîtront dans l'historique du projet et permettront d'identifier clairement les contributeurs à chaque modification.
- Pour configurer ces informations, on utilise les commandes suivantes :

```
git config --global user.name "Votre Nom"
git config --global user.email "votre@email.com"
```

- L'option --global indique que cette configuration s'appliquera à l'ensemble de vos futurs projets Git.

Configuration

- Lorsque nous utiliserons Gitlab ou Github, il y aura des configurations supplémentaires à effectuer.
- Nous y reviendrons à ce moment-là.

02

Versionner

init

- La commande **git init** permet de créer un dépôt Git vide dans un répertoire. En exécutant cette commande, un dossier caché nommé .git est généré, contenant toutes les informations nécessaires pour suivre l'historique de version. Sans ce dossier, il serait impossible de gérer les versions de votre projet.
- Voici comment procéder :

```
cd chemin/vers/mon/projet
git init
```

- Cela aura pour effet de rendre le répertoire « prêt » pour le versionnement avec Git.

status

- La commande **git status** permet de connaître l'état actuel du suivi des versions dans un dépôt Git. Elle fournit un résumé rapide des fichiers suivis, non suivis, et ceux qui sont en staging (prêts à être validés). C'est un excellent moyen de savoir où vous en êtes dans le processus de versionnement.

```
git status
```

- Il est recommandé d'utiliser cette commande régulièrement pour éviter des erreurs ou des oublis qui pourraient entraîner des complications dans l'historique du projet.

add

- L'une des particularités de Git est son système de **staging**, qui permet de sélectionner les fichiers à inclure dans le prochain commit. Vous pouvez considérer cette étape comme une "zone d'attente", où les fichiers que vous souhaitez sauvegarder sont listés avant de finaliser le commit.
- Voici quelques exemples de commandes pour ajouter des fichiers à cette zone :

```
git add <fichier>     # Ajoute un fichier spécifique à la zone de staging
git add <dossier>     # Ajoute tout le contenu d'un dossier
git add *.html        # Ajoute tous les fichiers avec l'extension .html
git add --all         # Ajoute tous les fichiers modifiés, ajoutés ou supprimés
git add *             # Ajoute tous les fichiers (modifications, ajouts, suppressions)
```

- Nous verrons plus tard comment retirer des éléments de cette zone de staging si nécessaire.

commit

- Une fois que la zone de staging est prête, vous pouvez réaliser votre premier **commit**. Un commit correspond à une étape dans l'historique de votre projet, et il est identifié par un message qui décrit les modifications apportées.
- Voici comment effectuer un commit :

```
git commit  # ouvre un éditeur pour écrire le message du commit
git commit -m "Message pour le commit"
```

L'option -m est souvent utilisée pour inclure un message directement dans la ligne de commande, sans ouvrir l'éditeur de texte (rédigez toujours un message à chaque commit).

log

- Sauvegarder les modifications est essentiel, mais pouvoir consulter l'historique l'est tout autant. La commande **git log** permet d'afficher des informations sur les différents commits de votre projet.
- Exemple de commande de base :

```
git log  # Affiche les derniers commits
```

- **Options utiles pour git log :**

--oneline	Affiche chaque commit sur une seule ligne pour une meilleure lisibilité.
-n <nombre>	Limite l'affichage aux <nombre> derniers commits.
-p <fichier>	Affiche l'historique des commits qui ont modifié un fichier spécifique.
--author <motif>	Affiche les commits correspondant à un auteur spécifique.

- Chaque commit dans le projet est identifié par une clé unique SHA-1. Cette clé assure l'intégrité du commit et permet de référencer un commit particulier dans différentes commandes.

diff

- La commande **git diff** permet de visualiser les différences entre les fichiers modifiés et leur dernière version dans le dépôt.
- **Utilisation basique**

```
git diff  # Affiche les différences pour tous les fichiers modifiés depuis le dernier commit
git diff <fichier>  # Affiche les différences spécifiques pour un fichier donné
```

- **Comparer avec ce qui est "stagé"**

  Si vous souhaitez comparer les fichiers dans la zone de staging (c'est-à-dire ceux qui sont prêts à être commités), vous devez utiliser l'option --cached ou --staged :

```
git diff --cached  # Compare les fichiers stagés avec le dernier commit
```

- **Comparer avec un ou plusieurs commits**

  Il est également possible de comparer des commits spécifiques ou de comparer deux commits entre eux :

```
git diff <commit>             # Compare l'état actuel avec un commit spécifique
git diff <commit1>..<commit2> # Compare deux commits entre eux
```

- Ces commandes vous permettent d'analyser les différences de manière précise, soit pour un fichier, soit pour l'ensemble du projet, avant de valider ou de modifier les modifications.

checkout

- La commande **git checkout** a plusieurs fonctionnalités :
  - Passer d'une branche à une autre (on reviendra sur ce point plus tard).
  - Restaurer un fichier à son état à un moment donné, correspondant à un commit spécifique.
  - Revenir à un commit précédent.

- ```
  git checkout <commit> <fichier>
  ```

  Cette commande transforme le fichier pour qu'il corresponde à son état au moment du commit et l'ajoute à la zone de staging.

- ```
  git checkout <commit>
  ```

  Cette commande modifie tous les fichiers pour reproduire l'état qu'ils avaient lors de ce commit. Cela vous place dans un état appelé detached HEAD, ce qui signifie que vous pouvez voir le projet tel qu'il était à ce moment-là, sans apporter de modifications à la branche actuelle. Vous pouvez observer d'anciens commits, mais pour vraiment revenir en arrière, on utilisera la commande reset.

revert

- La commande **git revert** permet d'annuler un commit spécifique.

```
git revert <commit>
```

- Cette commande défait les modifications apportées par le commit en créant un nouveau commit. Contrairement à la commande reset (juste après), elle ne modifie pas l'historique du projet, mais ajoute un commit d'inversion. Par exemple, les lignes ajoutées par le commit d'origine seront supprimées, et les fichiers supprimés seront recréés.
- Cela permet de maintenir un historique exhaustif tout en annulant les effets d'un commit précédent.

reset

Tout comme la commande checkout, la commande **git reset** permet aussi de revenir sur des modifications, de différentes manières. Cependant, elle modifie l'historique du dépôt et peut dans certains cas entraîner la perte de modifications, c'est pourquoi il faut l'utiliser avec précaution (surtout lorsque vous voyez l'option --hard).

```
git reset <fichier>
```

retire un fichier de la zone de staging, mais conserve les modifications locales dans le fichier. Il n'y a aucune perte de travail non commité. Préférer néanmoins la nouvelle syntaxe : git restore --staged <fichier>

```
git reset
```

enlève tous les fichiers de la zone de staging, sans affecter les modifications locales dans le répertoire de travail. Les changements ne sont pas supprimés, juste désindexés.

```
git reset --hard
```

ramène l'état de votre répertoire de travail au dernier commit enregistré. Cette commande doit être utilisée avec une **grande vigilance**. Toute modification non commité sera perdue définitivement.

```
git reset <commit>
```

recule à un commit spécifique en réinitialisant la zone de staging. L'historique est effacé après ce commit (les commits ultérieurs seront perdus, mais les modifications locales ne seront pas touchées). Cela permet de "nettoyer" l'historique avant de soumettre un nouveau commit consolidé.

```
git reset <commit> --hard
```

réinitialise tout (la zone de staging et le répertoire de travail) pour correspondre à l'état du commit spécifié. Toute modification non enregistrée ainsi que tous les commits postérieurs à ce commit seront effacés. À utiliser avec une **extrême prudence**.

reset

Il est important de ne **jamais utiliser** la commande **reset** après avoir publié vos modifications (après un **push**), car elle peut causer des incohérences dans le dépôt distant. Cependant, elle est très utile pour nettoyer l'historique local avant de le publier en ligne.

Documentation

En cas de doute, n’hésitez pas à consulter la documentation officielle de Git !

https://git-scm.com/doc

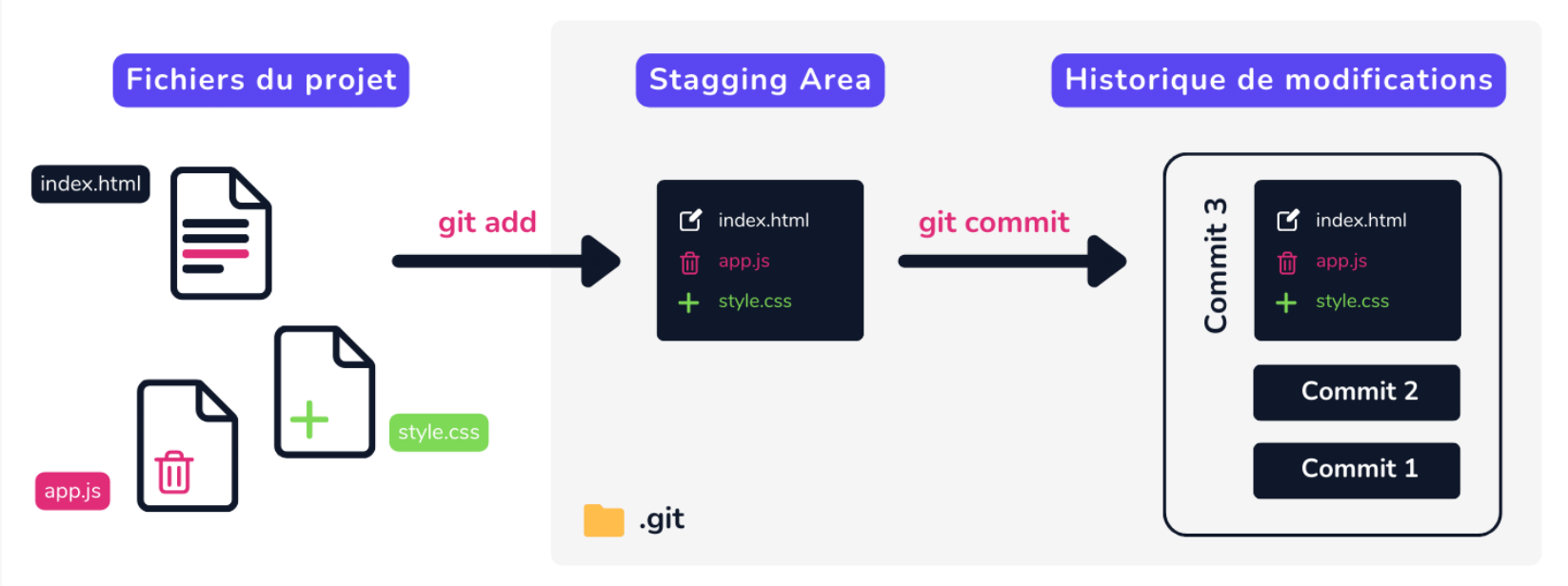
Versionner

TP

A la recherche d’un secret

Discussion

Choisir les éléments à inclure dans un commit...



.gitignore

- Le fichier **.gitignore** joue un rôle essentiel dans un projet Git en indiquant **quels fichiers ou répertoires Git doit ignorer**.
- Pour ajouter un fichier à la liste des ignorés, vous créez simplement un fichier .gitignore **à la racine de votre projet** et y indiquez les fichiers ou motifs à exclure. Ce fichier sera ensuite lu par Git à chaque opération pour déterminer quels fichiers ignorer.
- Le fichier .gitignore permet de spécifier les fichiers ou dossiers que vous ne souhaitez pas suivre dans votre dépôt Git. Cela est utile pour éviter d'inclure des fichiers temporaires, de configuration locale, ou générés automatiquement (comme des fichiers compilés, logs, ou cache) dans l'historique de votre projet. Il va également permettre d'exclure les fichiers contenant des informations qu'on ne souhaite pas diffuser, comme les clefs et les mots de passe (en gardant en tête que git va aussi servir à collaborer avec d'autres personnes).
- En excluant les fichiers non pertinents, .gitignore aide à garder un historique des commits propre et pertinent. En ignorant certains fichiers lourds ou non nécessaires, .gitignore peut même améliorer les performances du dépôt, car moins de fichiers sont suivis et analysés à chaque commit.

Un template de .gitignore pour Python : https://github.com/github/gitignore/blob/main/Python.gitignore

03

Collaborer

remote

- Le remote dans Git désigne un **dépôt distant**, c'est-à-dire une copie de votre dépôt Git hébergée sur un serveur externe ou une plateforme comme GitHub, GitLab, Bitbucket, ou un serveur privé.
- Un dépôt distant permet de collaborer avec d'autres développeurs, de sauvegarder le travail à distance, et de synchroniser les modifications entre plusieurs machines.

Rôles d'un remote dans Git

- **Collaboration** : Partager le code avec d'autres développeurs.
- **Sauvegarde** : Conserver une copie du code à un endroit sûr.
- **Synchronisation** : Mettre à jour le dépôt local avec les changements d'autres contributeurs.
- **Déploiement** : Permettre d'envoyer le code vers des serveurs pour des environnements de production ou de staging

remote

Afficher les remotes

```
git remote -v
```

➡ Liste tous les dépôts distants associés au projet.

```
origin  https://github.com/user/repo.git (fetch)
origin  https://github.com/user/repo.git (push)
```

remote

Ajouter/supprimer/renommer un remote

Ajouter un remote :

```
git remote add <nom> <url>
```

➡ origin est le nom donné au remote, et l'URL est celle du dépôt distant

```
git remote add origin https://github.com/user/repo.git
```

Supprimer un remote :

```
git remote remove <nom>
```

```
git remote remove origin
```

Renommer un remote :

```
git remote rename <ancien_nom> <nouveau_nom>
```

```
git remote rename origin upstream
```

remote

Changer l'URL d'un remote

```
git remote set-url <nom> <nouvelle_url>
```

```
git remote set-url origin https://github.com/user/new-repo.git
```

remote

Conventions de nommage

- `origin` : Par convention, c'est le nom attribué par défaut au premier dépôt distant que vous configurez. Ce n'est qu'un alias, et vous pouvez le renommer si besoin.
- `upstream` : Souvent utilisé pour désigner le dépôt principal (par exemple, un projet open source) lorsqu'un utilisateur travaille sur un fork.

remote

Récupérer les modifications depuis un remote

Fetch

```
git fetch <nom>
```

➡ Cette commande télécharge les nouvelles branches et les modifications du dépôt distant mais ne les fusionne pas dans le dépôt local

```
git fetch origin
```

Pull

```
git pull <nom> <branche>
```

➡ Cette commande télécharge les modifications et les fusionne directement avec la branche locale.

```
git pull origin main
```

remote

Envoyer (pousser) des modifications vers un remote

Push

```
git push <nom_du_remote> <branche>
```

```
git push origin main
```

➡ Envoie les modifications locales vers la branche `main` du dépôt distant `origin`.

remote

Que se passe-t-il quand il y a plusieurs remotes ?

Il faut explicitement spécifier sur quel remote agit si ce n'est pas celui qui est configuré par défaut pour la branche actuelle (souvent origin)

```
git fetch upstream
```

```
git push origin main
```

➡ Il est possible de consulter les configurations de remote spécifiques d'une branche avec :

```
git branch -vv
```

remote

Configuration des URLs pour fetch et push et cas d'usage avec plusieurs remotes

Il est possible de configurer des URLs distinctes pour les actions de fetch et de push :

```
git remote set-url --add --fetch origin <fetch_url>
```

```
git remote set-url --add --push origin <push_url>
```

C'est utile dans certains cas particuliers :

- Accès différencié entre lecture et écriture, dépôts multiples (forks, privés/publics).
- Performances (miroirs locaux).
- Automatisation ou contrôle des flux de contributions.

De manière générale, on peut travailler avec plusieurs remotes si on travaille sur un **fork** ou sur plusieurs copies d'un dépôt hébergées sur **différents serveurs**.

04

Pour aller (un peu) plus loin

Les branches

- Branche = « pointeur mobile qui pointe vers un commit spécifique »
- Les branches permettent de travailler sur différentes versions du code en parallèle, facilitant le développement collaboratif et l'expérimentation sans affecter la version principale du projet.
- `main` / `master` : la branche par défaut créée lors de l'initialisation d'un dépôt Git. Elle va (généralement) contenir la version stable (de production).
- Autres branches : développer de nouvelles fonctionnalités, corriger des bugs, expérimenter… sans interférer avec la branche principale. Zoomons sur ces autres branches classiques.

Les branches

Branche principale

Contient le code en production ou stable.

Nom conventionnel : `main` ou `master`

Branches de fonctionnalités

Utilisées pour développer une nouvelle fonctionnalité sans affecter la base de code principale.

Nom conventionnel : `feature/nom-feature`

Branches de correction de bugs

Utilisées pour corriger des bugs spécifiques.

Nom conventionnel : `bugfix/nom-bug`

Branches de développement

Point d'intégration pour plusieurs fonctionnalités ou contributions.

Nom conventionnel : `develop`

Branches de version ou de release

Utilisées pour préparer une version stable.

Nom conventionnel : `release/x.y.z`

Branches de hotfix

Utilisées pour corriger rapidement un problème critique en production.

Nom conventionnel : `hotfix/nom-correctif`

Les branches

Travailler avec des branches distantes

Créer une branche locale basée sur une branche distante

```
git checkout -b nom-branche origin/nom-branche
```

Pousser une branche locale vers un dépôt distant

```
git push origin nom-branche
```

Suivre une branche distante

```
git branch --set-upstream-to=origin/nom-branche
```

Récupérer les branches distantes

```
git fetch origin
```

Les branches

Workflow typique avec des branches

1. Créer une branche pour développer une fonctionnalité

```
git checkout -b feature/new-feature
```

2. Travailler sur la branche et commit les changements

```
git add .
git commit -m "Add new feature"
```

3. Pousser la branche sur le dépôt distant

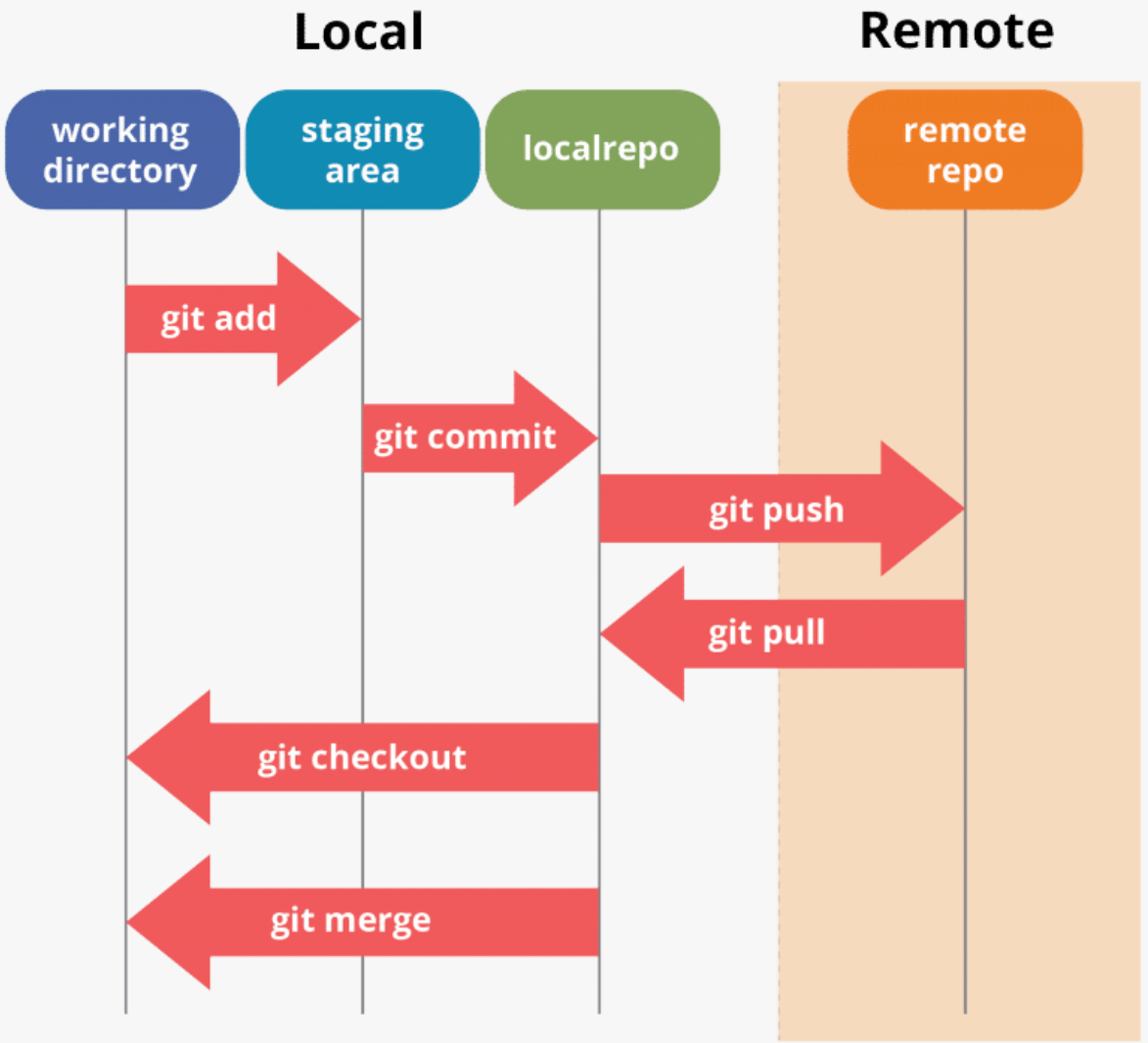
```
git push origin feature/new-feature
```

4. Ouvrir une Pull/Merge Request pour fusionner les changements dans main

5. Fusionner et supprimer la branche après validation

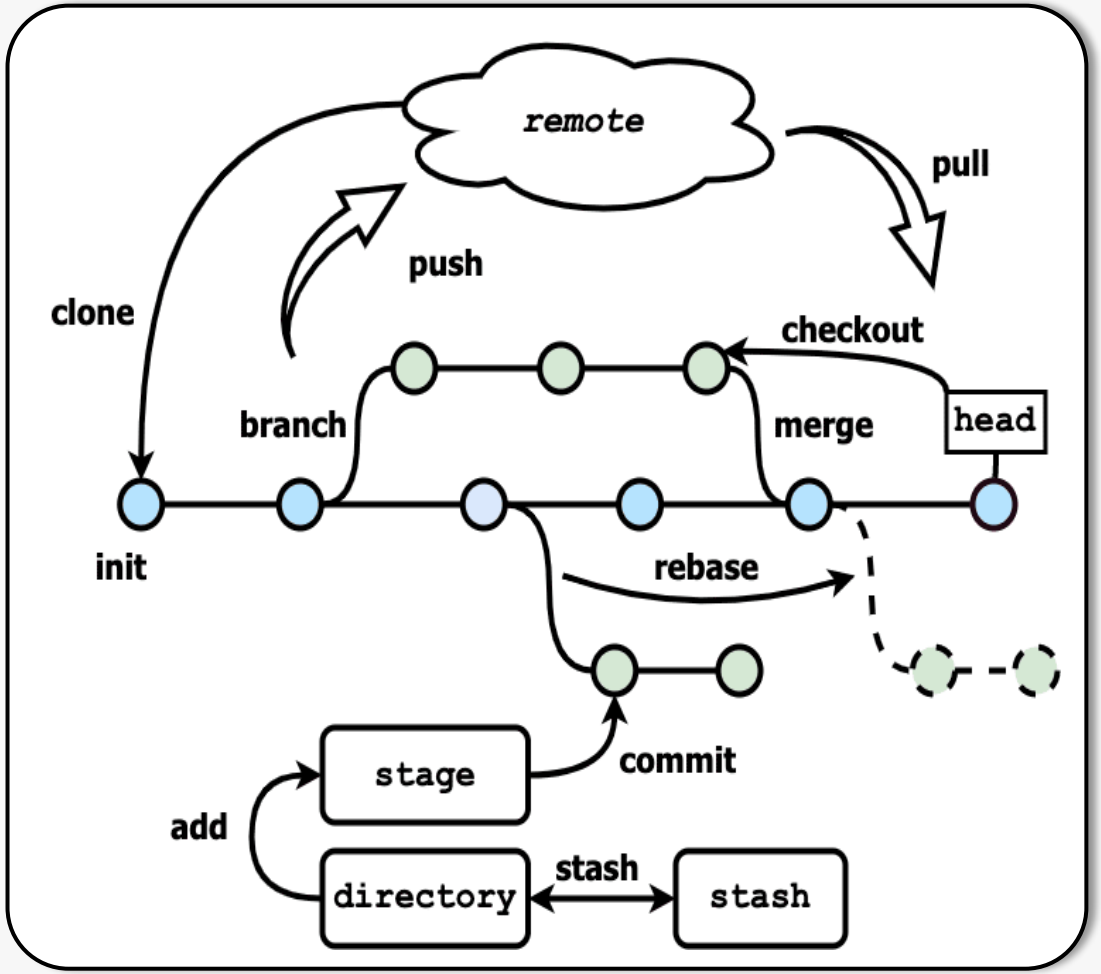
Synthèse

Un schéma (simple) d'ensemble



Synthèse

Un autre schéma d'ensemble



Retour sur le pointeur HEAD

- Le pointeur HEAD représente le pointeur qui indique la branche ou le commit actuellement actif dans le dépôt Git. Si on effectue des modifications ou des commits, ils seront liés à ce point.
- Il est possible de déplacer le HEAD pour explorer l'historique ou corriger des erreurs :
  - `git checkout` (désormais remplacé par git switch) pour naviguer entre branches.
  - `git reset` pour déplacer le HEAD à un commit précédent, modifiant ainsi l'état de l'historique.
- Exemples courants :
  - `git reset --hard HEAD~1` : revient au commit précédent et supprime les modifications actuelles.
  - `git checkout [commit_hash]` : examine un commit spécifique sans changer de branche.

Tags

- Un `tag` est un pointeur immuable associé à un commit spécifique, utilisé pour marquer une version importante du projet (exemple : v1.0, release-2024).
- Les tags facilitent le suivi des versions ou des étapes-clefs dans un projet, en particulier les déploiements et les releases.
- Il existe des tags « légers » (`git tag <tag_name>`), contenant une simple référence, et des tags « annotés » (`git tag -a <tag_name> -m "message"`), incluant des métadonnées (auteur, date, …).

Rebase

- Le rebase réécrit l'historique Git en appliquant les commits d'une branche sur une autre.
- Contrairement à merge, qui crée un commit de fusion, rebase intègre les modifications en réorganisant les commits dans une ligne d'historique plus linéaire.
- Utiliser rebase permet de maintenir un historique propre et linéaire, surtout pour des projets collaboratifs. Par exemple, si une branche est en retard sur main, utiliser `git rebase main` applique les commits de la branche en question après les commits les plus récents de main.

Gérer les conflits

Quand ?

Lors d'une tentative de `merge` ou de `rebase`, si deux branches ont modifié les mêmes lignes de code ou fichiers de manière incompatible.

Comment ?

1. Git marque les conflits dans les fichiers concernés avec des délimiteurs (`<<<<<<<`, `=======`, `>>>>>>>`).
2. L'utilisateur choisit quelle version conserver ou combine les changements.
3. Une fois le confilt résolu, il faut « terminer » l'opération (avec git add <file> suivi de git commit, ou avec git rebase --continue).

Bonnes pratiques :

Résoudre les conflits rapidement pour éviter un blocage dans le développement.

Communiquer avec l'équipe si les conflits concernent des modifications majeures.

Fin de la partie Git

Merci !