La gestion de version décentralisée

Logistique

- repas
- horaires
- contraintes

Tour de table

- Typologie de projet (Techno, environnement) ?
- Gestionnaire de source actuel

De quoi va-t-on parler

- Contexte
- Installation et environnement
- Le fonctionnement de git
- Utilisation au quotidien
- Les dépôts distants
- Utilisation des branches
- Workflow de développement
- Gestion des sous-modules
- Bonnes pratiques et résolution des problèmes courants
- Conclusion

Contexte

La gestion de version

> *Maintient l'ensemble des versions d'un ou plusieurs fichiers*

Que permet la gestion de version ?

Offrir la possibilité de **revenir** facilement à une **version antérieure.**

Garder la trace de modifications : **qui**, **quoi**, **quand** et **pourquoi ?**

Permettre à **plusieurs personnes** de travailler **simultanément** sur le même projet sans se marcher sur les pieds.

> *Si le code n'est pas enregistré dans un logiciel de gestion de version, il n'existe pas.*

2 Grandes familles de logiciels de gestion de version

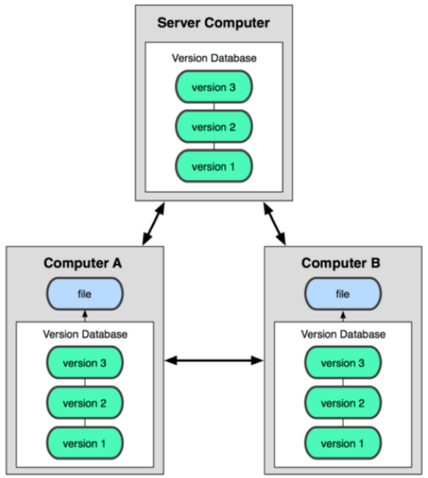
Centralisés (cvs, svn, tfs).

Distribués (git, mercurial, bazaar).

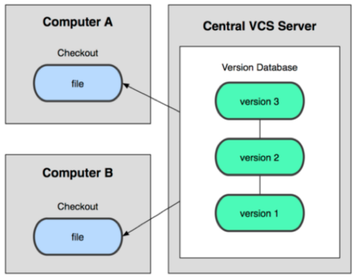
Qu'est-ce qu'un logiciel de gestion de version distribué ?

Logiciel de version distribué

Distribué



Centralisé



Distribué signifie :

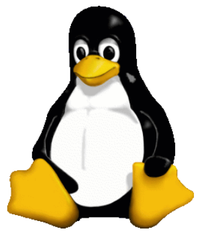
- Chaque développeur dispose d'un **dépôt complet en local**
- Techniquement le dépôt central n'est pas différent du dépôt local
- Facilement utilisable hors ligne
- La création de branche est facile

Git

- Créé en 2005 par Linus Torvald pour le noyau linux.
- Utilisé par Linux, Android, Eclipse...
- Intégration dans Eclipse, XCode, VSCode, Intellij...
- Utilisé par Google, SAP, Twitter.
- Populaire ++

Installation et environnement

En fonction de l'OS



Centos / Fedora : `yum install git`

Debian / Ubuntu : `apt-get install`

Git Pour Windows : Git Bash + Git GUI + Shell Integration

https://gitforwindows.org/

Via l'installeur graphique

https://sourceforge.net/projects/git-osx-installer/

Logiciels

GitKaken : produit par Axesoft, gratuit pour des utilisation non commerciale.

VSCode : open source, avec les extensions GitHistory et GitLens.

Plugin officiel pour Eclipse, intégré mais pas de visualisation de l'arbre git.

`gitk --all` Intégré avec git fournit le minimum.

https://git-scm.com/downloads/guis

Configuration

3 niveaux de configuration possible (avec surcharge)

- Globale à tout le système: `/etc/gitconfig`

```
$ git config --system <key> <value>
$ git config -e --system
```

- Propre à l'utilisateur : `$HOME/.gitconfig`

```
$ git config --global <key> <value>
$ git config -e --global
```

- Spécifique au dépôt : `.git/config`

```
$ git config <key> <value>
$ git config -e
```

Exemples de configuration

Configuration du nom d'utilisateur et de son e-mail :

```
$ git config --global user.name “Jean BON”
$ git config --global user.email jean.bon@exemple.fr
```

Autres configurations possibles :

```
$ git config --global core.editor vim
$ git config --global color.ui true
$ git config --global http.proxy http://hostname:3128
```

Ignorer des fichiers

Par configuration d'un fichier à différents niveau :

Versionné dans le dépôt (la meilleure solution).

`<nom_depot>/.gitignore`

Spécifique au dépôt, non versionné.

`<nom_depot>/.git/info/exclude`

Commun à tous les utilisateurs.

`<HOME>/.gitignore`

Exemple de fichier .gitignore

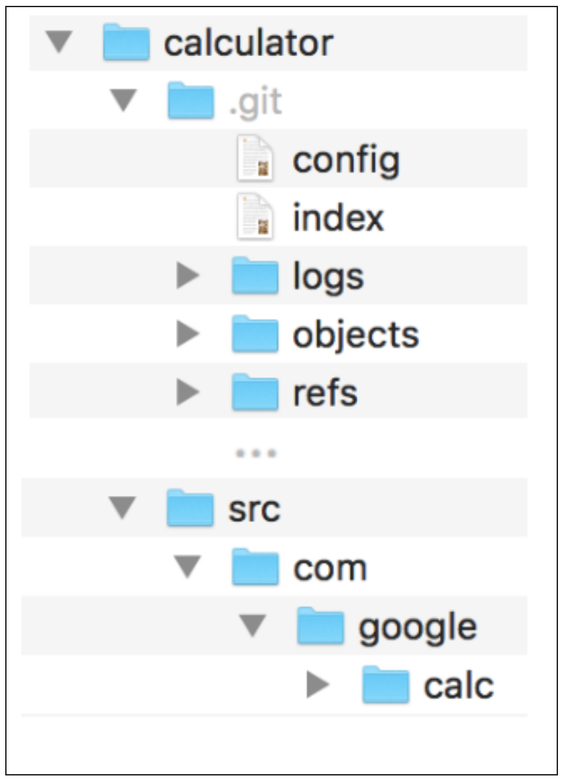
```
node_modules/
dist/
!dist/kdbxweb.js
.log
.idea/
```

Les principes de GIT

Sécurisant

- Les opérations sont locales
- Toutes les modifications sont tracées
  - Système de checksum
  - 1 hash est généré pour chaque opération(sha-1)
- Principalemeent des ajout, peu de suppression
  - Perte d'information difficile
  - Retour arrière très souvent possible
- Snapshot et pas une recopie complète

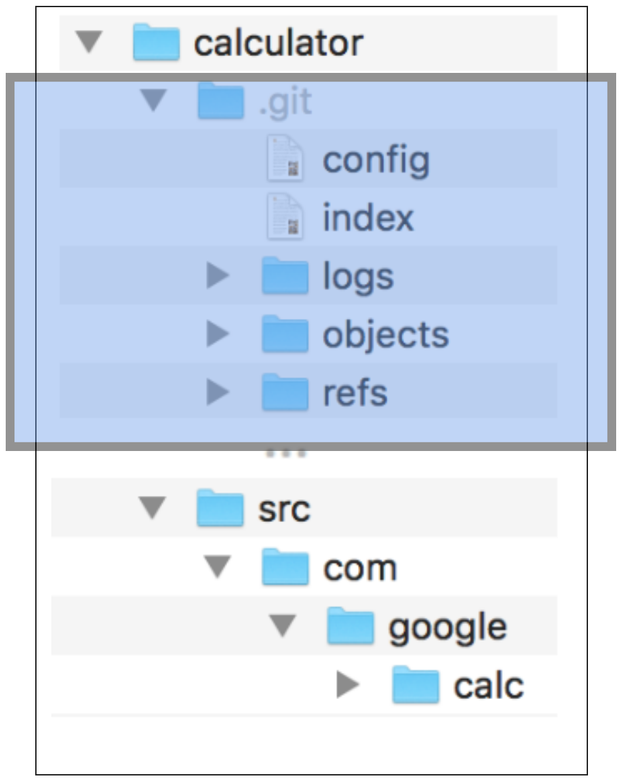
Structure d'un dépôt



Un dépôt Git est créé en local par :

```
$ git init
$ git clone (voir plus loin)
```

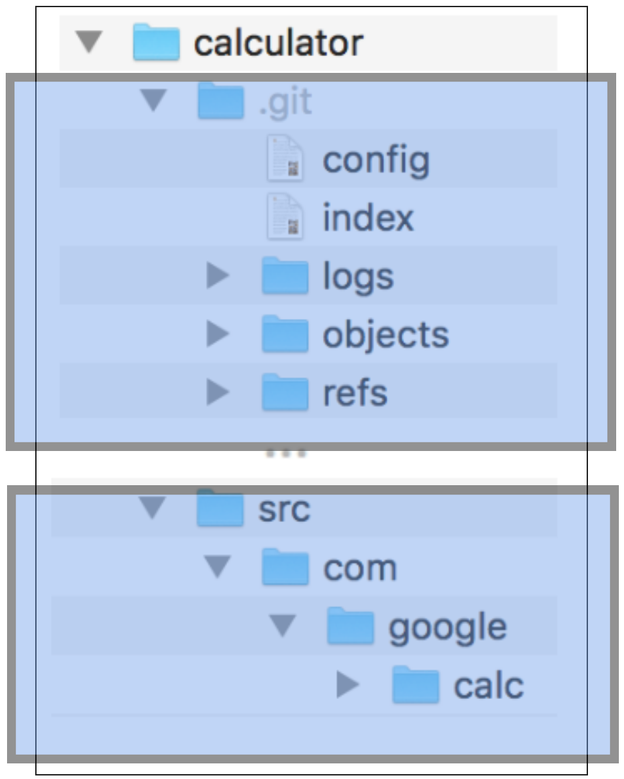
Structure d'un dépôt



Le dossier `.git` correspond au dépôt git.

Il contient une version complète du dépôt (historique, configuration...)

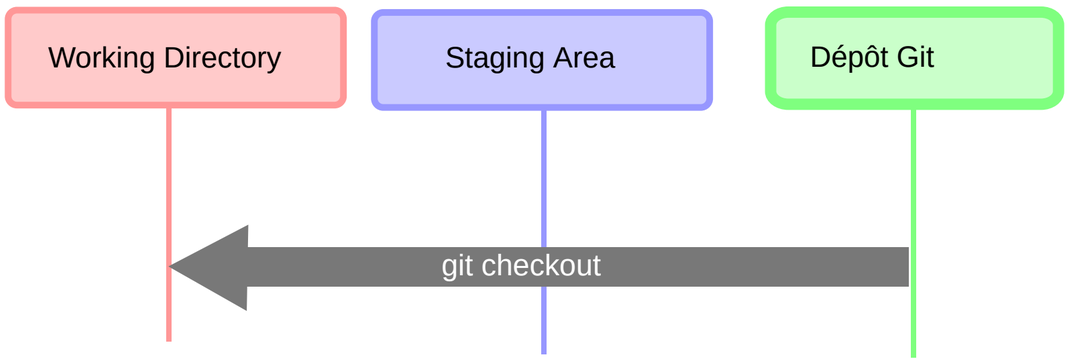
Structure d'un dépôt



Le dossier `.git` correspond au dépôt git.

Les dossiers et fichiers à coté du répertoire `.git` constituent le **working directory**.

Répertoire de travail : checkout



Remplit le **working directory** avec le contenu du **commit** sur lequel on veut travailler.

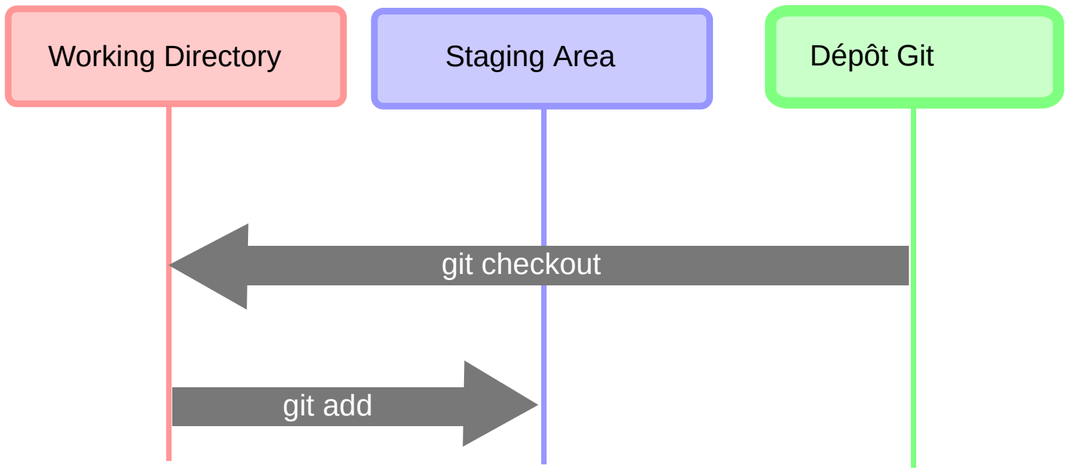
Effectuer des changements

- Commencer à faire des modifications
- pas besoin de dire à GIT sur quels fichiers on travaille
- Il suffit juste de lui dire **quels changements sont à commiter**

```
$ git add <file>
$ git rm <file>
```

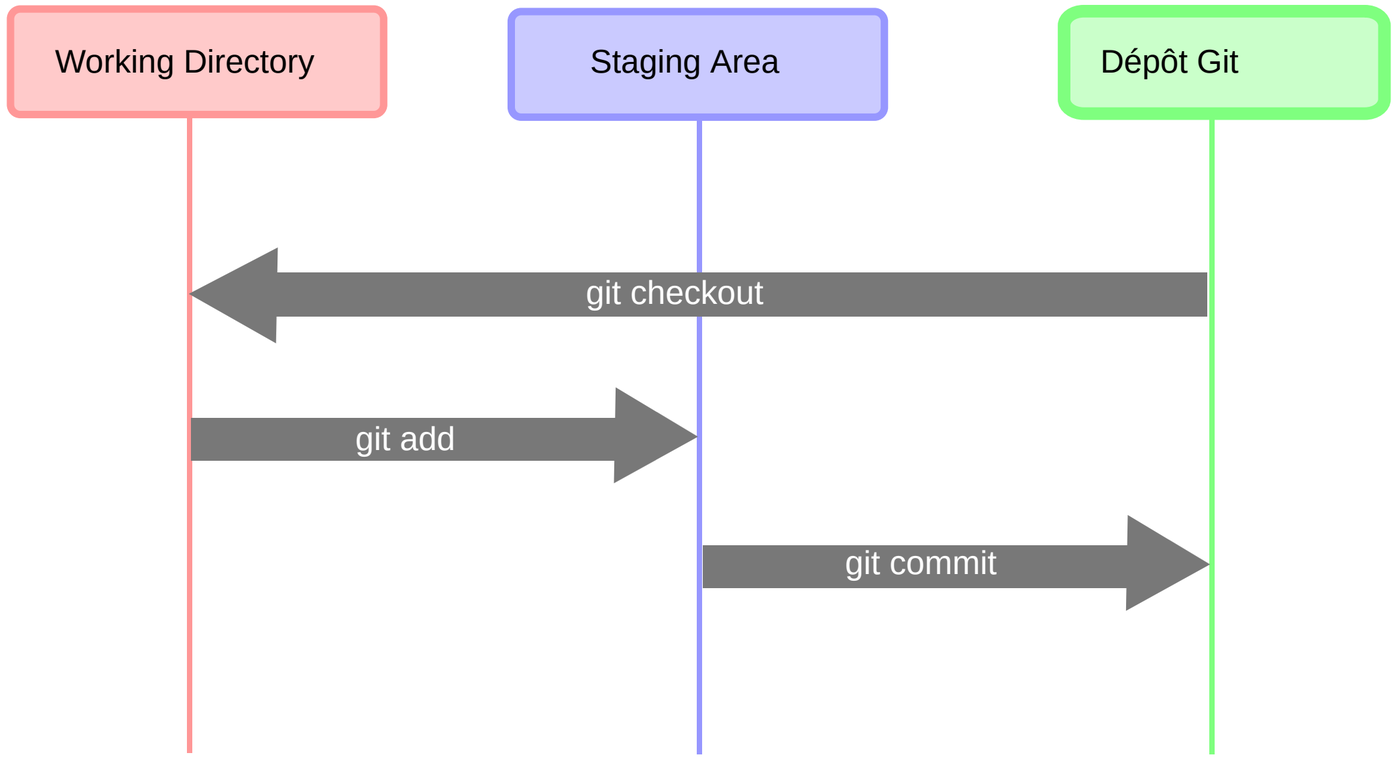
Que se passe-t-il si l'on fait un `git add` ?

Ajout des modifications : add



Staging area ou **index** est l'endroit où le prochain commit est préparé.

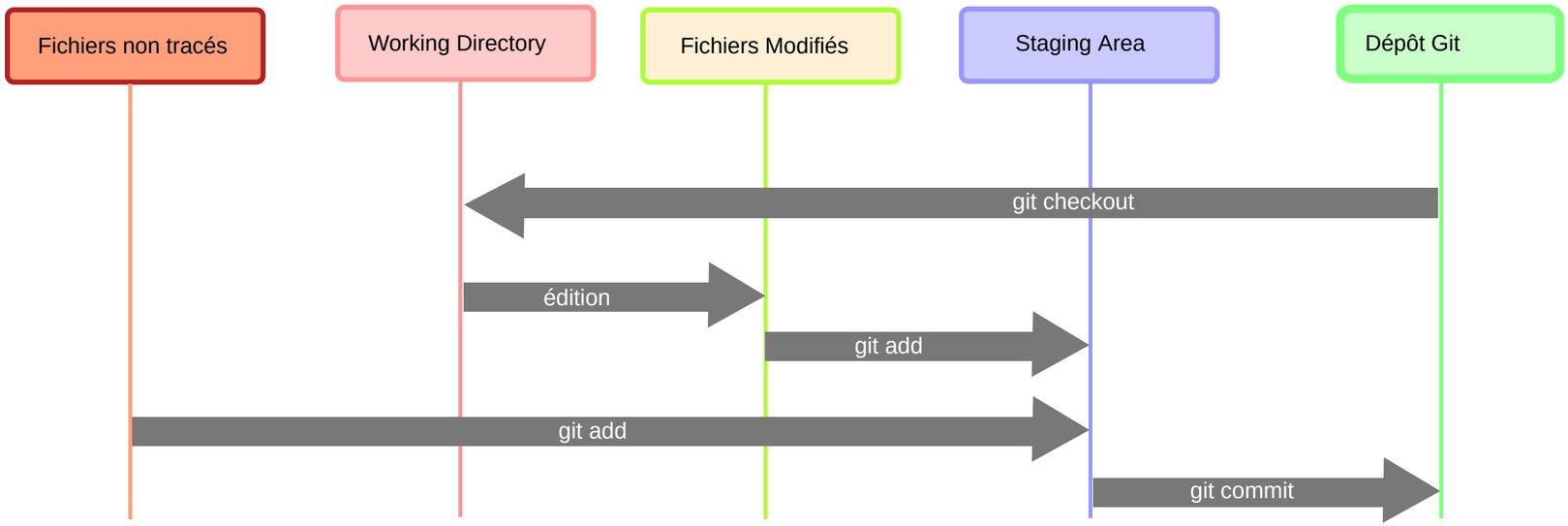
Validation : commit



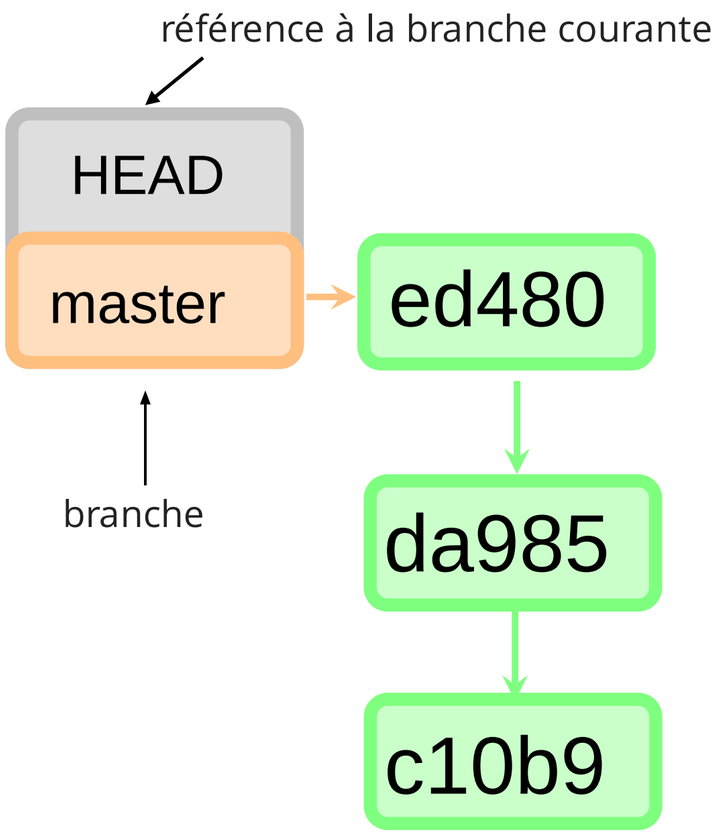
`git commit` publie uniquement les changements présents dans la **staging area**.

Il peut rester des éléments modifiés dans le **working directory** qui ne seront pas inclus dans le **commit**.

En résumé : Cycle de vie des fichiers



Historique des commits



Les **commits** ce succèdent les uns avec les autres.

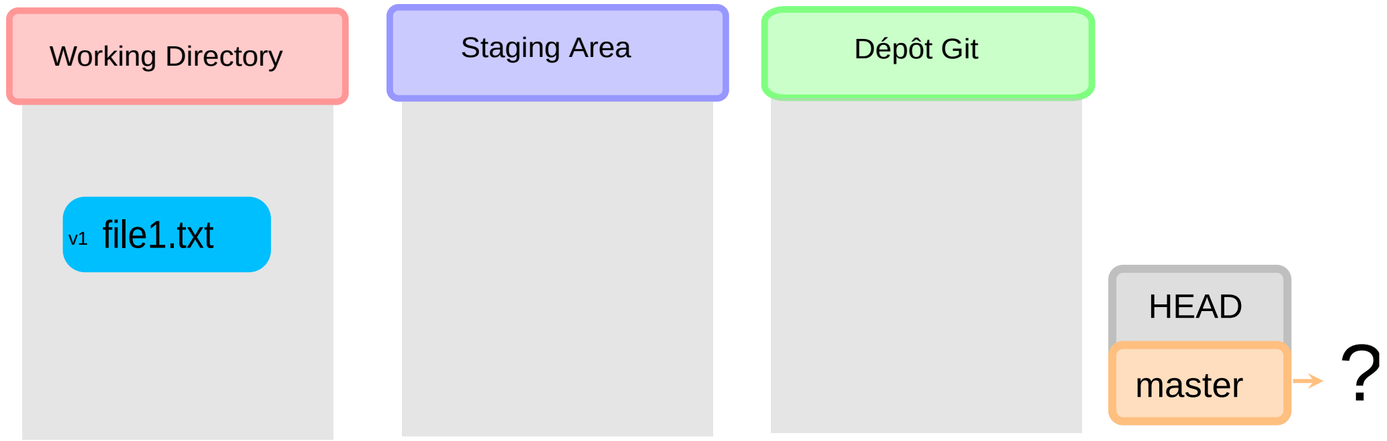
Les flèches représentent les relations entre les commits.

Une façon de consulter l'historique :

```
$ git log
$ git log --oneline
$ git log --graph
$ gitk
```

Utilisation au quotidien

Initialisation

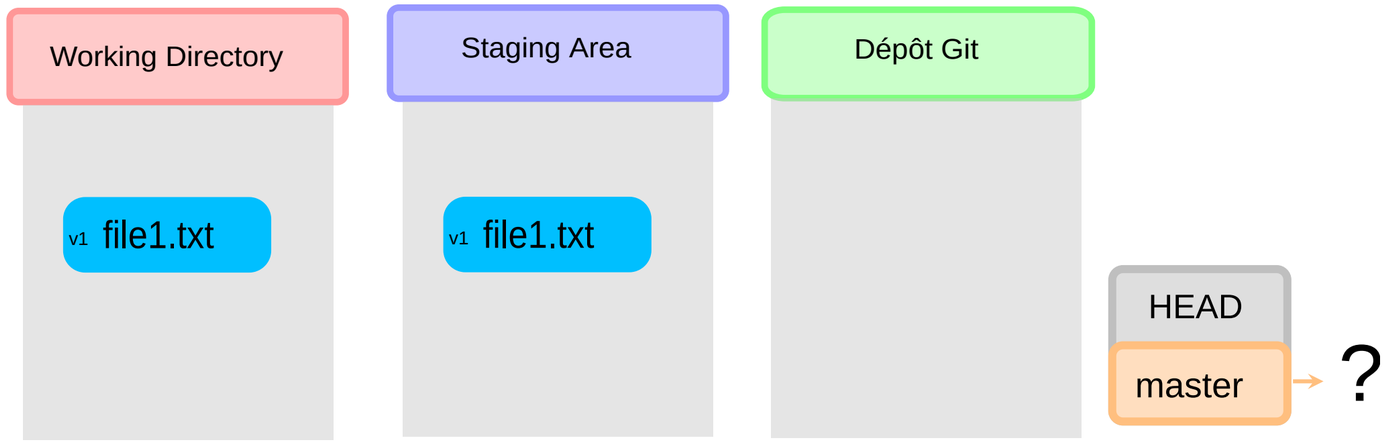



```
$ git init
```

Création d'un dépôt local

Génère le dossier `.git`

Ajout de fichiers



Ajout de tous les fichiers

```
$ git add .
```

Ajout d'un fichier en particulier

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

Ajout de tous les fichiers textes

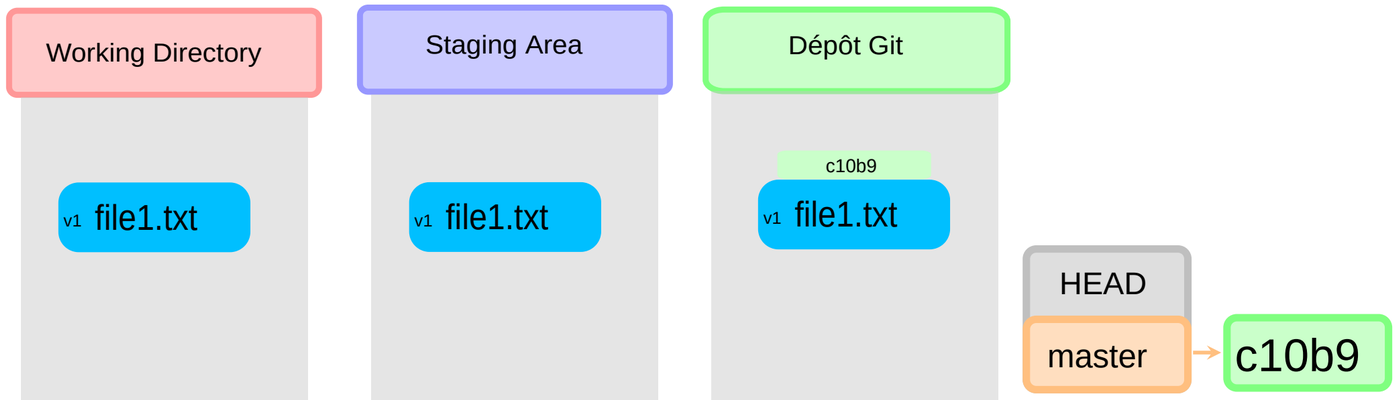
```
$ git add *.txt
```

Déplacement de fichiers

```
$ git mv old new
```

```
$ git rm fichier
```

Commit d'un fichier



Commit des fichiers dans l'index

```
$ git commit –m "Description commit"
```

Quand commiter ?

- 1 commit = 1 tâche
- Commit = code stable

aussi souvent que possible !

Formattage des commits

Mettre en place des règles sur le projet.

1 bonne pratique = les règles Google AngularJS.

https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#commits

Format des commits

<type>(scope):<sujet>

type : type de commit (feat, fix, chore, test, docs)

scope : partie du code concerné

sujet: description succincte

Exemples

```
feat(Module): add info method
fix(input[number]): add min/max
docs(Licence): update definition
chore(travis): fix deploy rules
```

Les commandes usuelles

Consulter l'état d'un dépôt : `git status`

Consulter l'historique des commits du plus récent au plus ancien : `git log`, `git log --oneline`, `git log --graph`, `gitk`

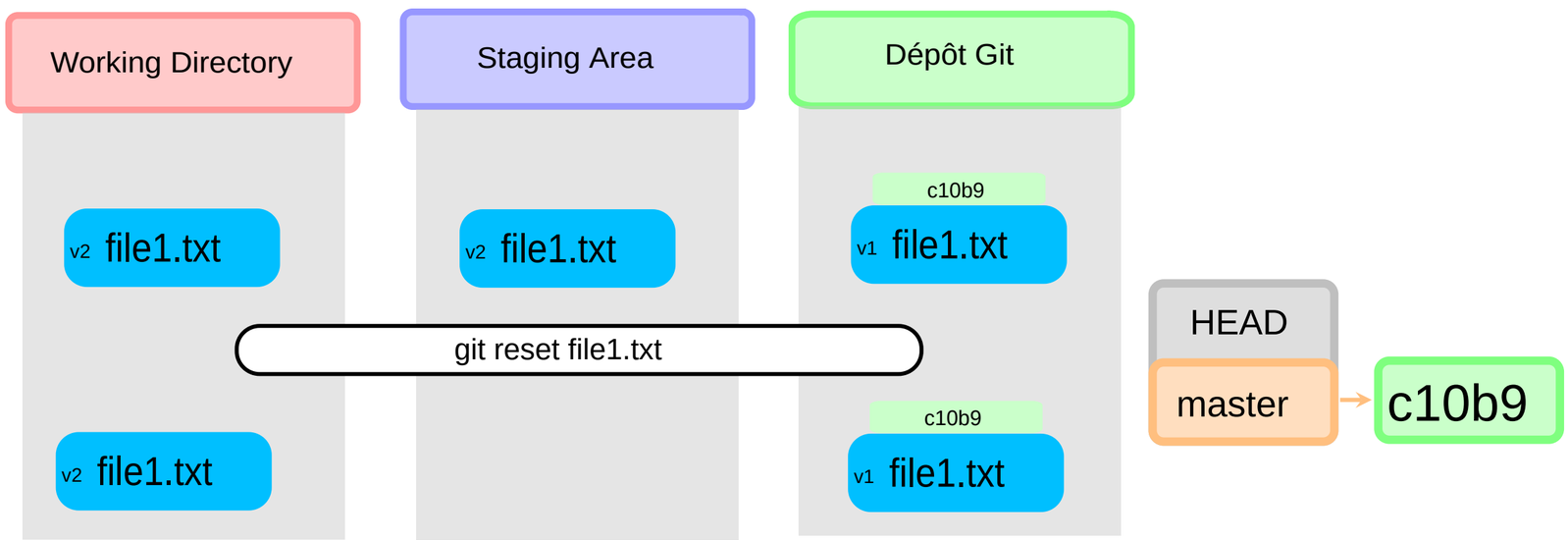
Basculer vers une version antérieure sur un commit particulier : `git checkout <hash>`

Revenir au commit le plus récent :
`git checkout <Nom_Branche>` / `git checkout master`

Récupérer l'état d'un fichier tel qu'il était à lors d'un commit particulier : `git checkout <hash> <fichier>`

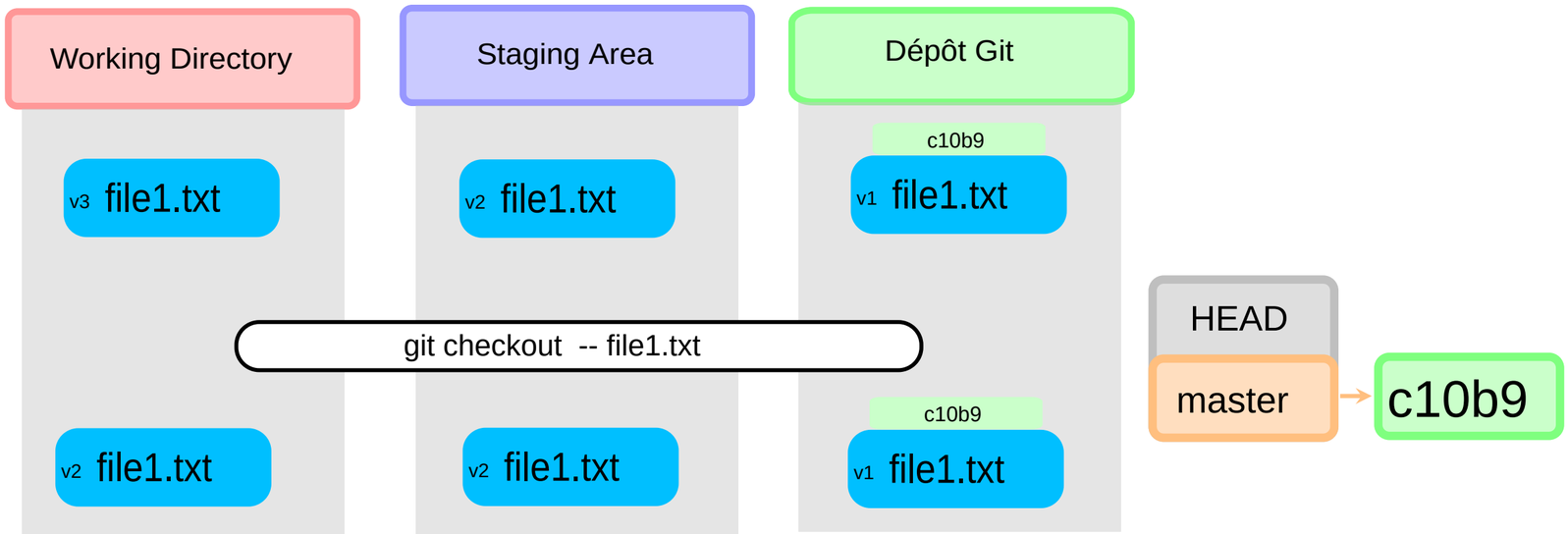
Retour arrière

Retirer un fichier de la " **stagging area**".



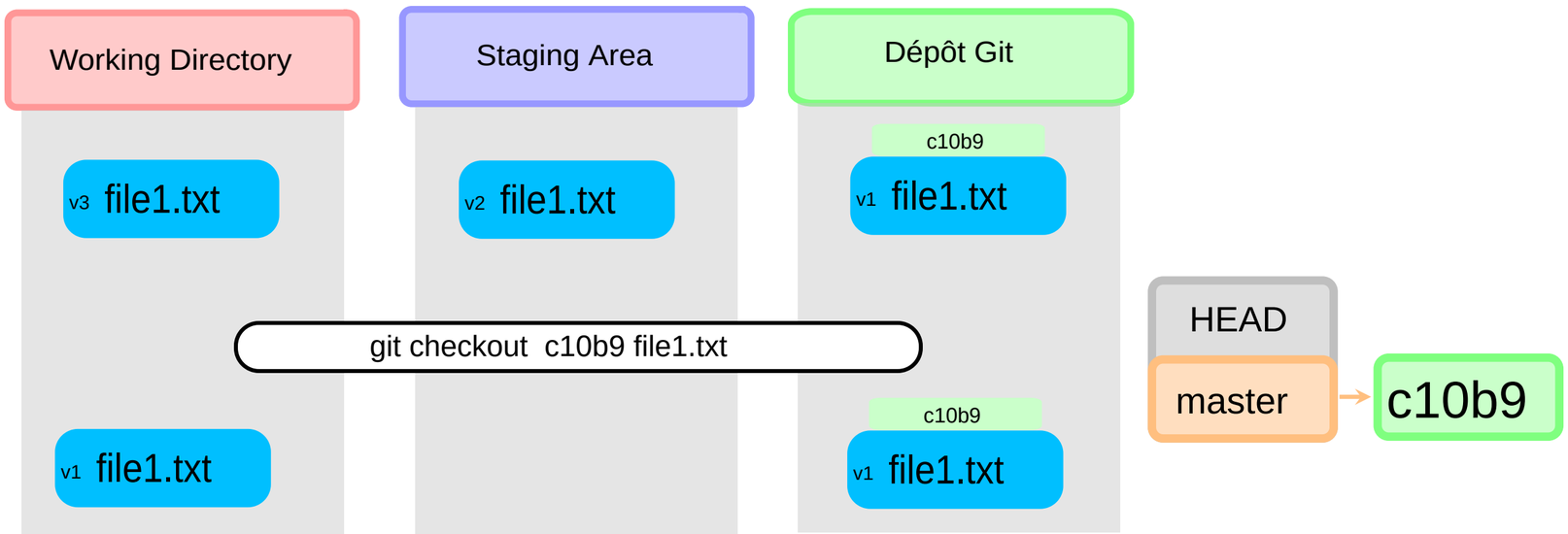
Retour arrière

Restaurer un fichier depuis la **"stagging area"**.



Retour arrière

Restaurer un fichier depuis un **commit**.



3 modes de reset

Hard : Perte des "commits" et des fichiers.

Mixed : mode par défaut, supprime les commit, leur contenu se retrouve dans la " **working area**".

Soft : supprime les "commits", leur contenu se retouve and la " **stagging area**".

Ne jamais supprimer un commit distant

Avoir un "working directory" propre

(pas de modifications en cours)

Reset Hard


```
git reset --hard <no_commit>
```

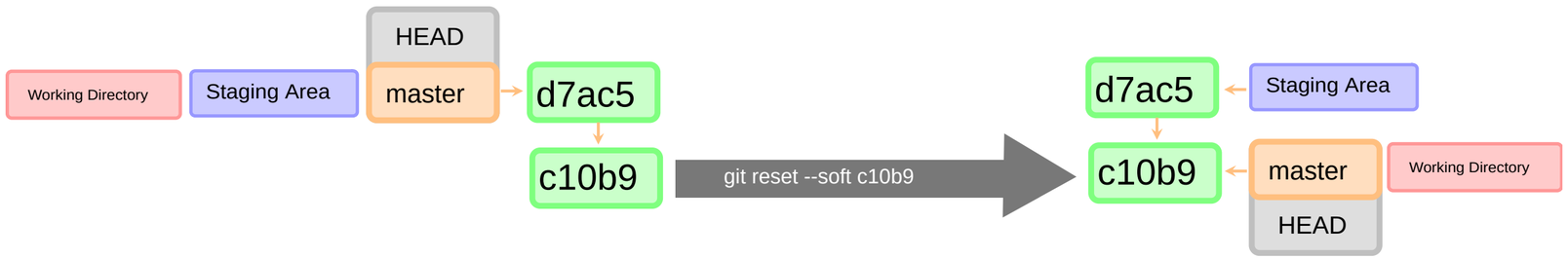
Le numéro de commit correspond à la position sur laquelle il y a une reposition.

Reset --mixed


```
git reset <no_commit>
```

Reset par défaut toutes les modifications sont mélangées dans la **working directory**.

Reset --soft




```
git reset --soft <no_commit>
```

Reset toutes les modifications sont dans la **staging area**.

Modifier le dernier commit

Pour corriger le message d'un dernier commit
et/ou ajouter un fichier oublié.

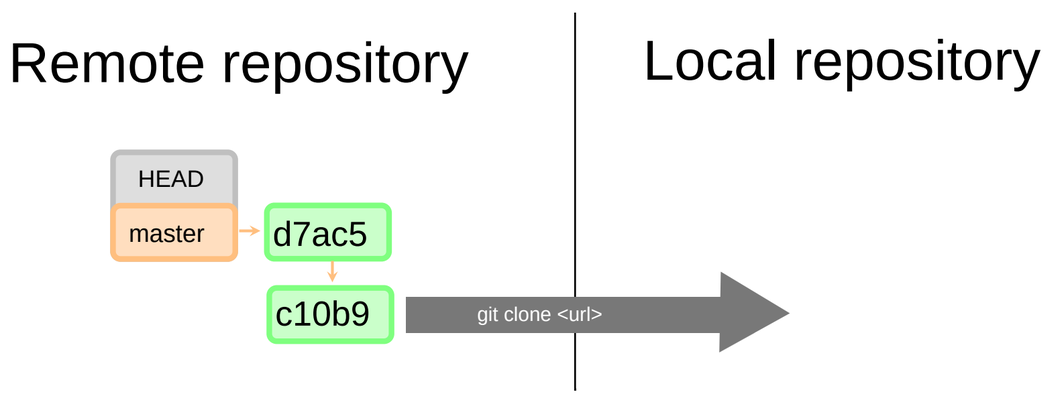
```
git commit --amend <text>
```

Ne fonctionne que si le commit est local !

Cloner un dépôt

Construit un dépôt local à l'image d'un dépôt distant
dans le répertoire courant

```
git clone <remote_url>
```

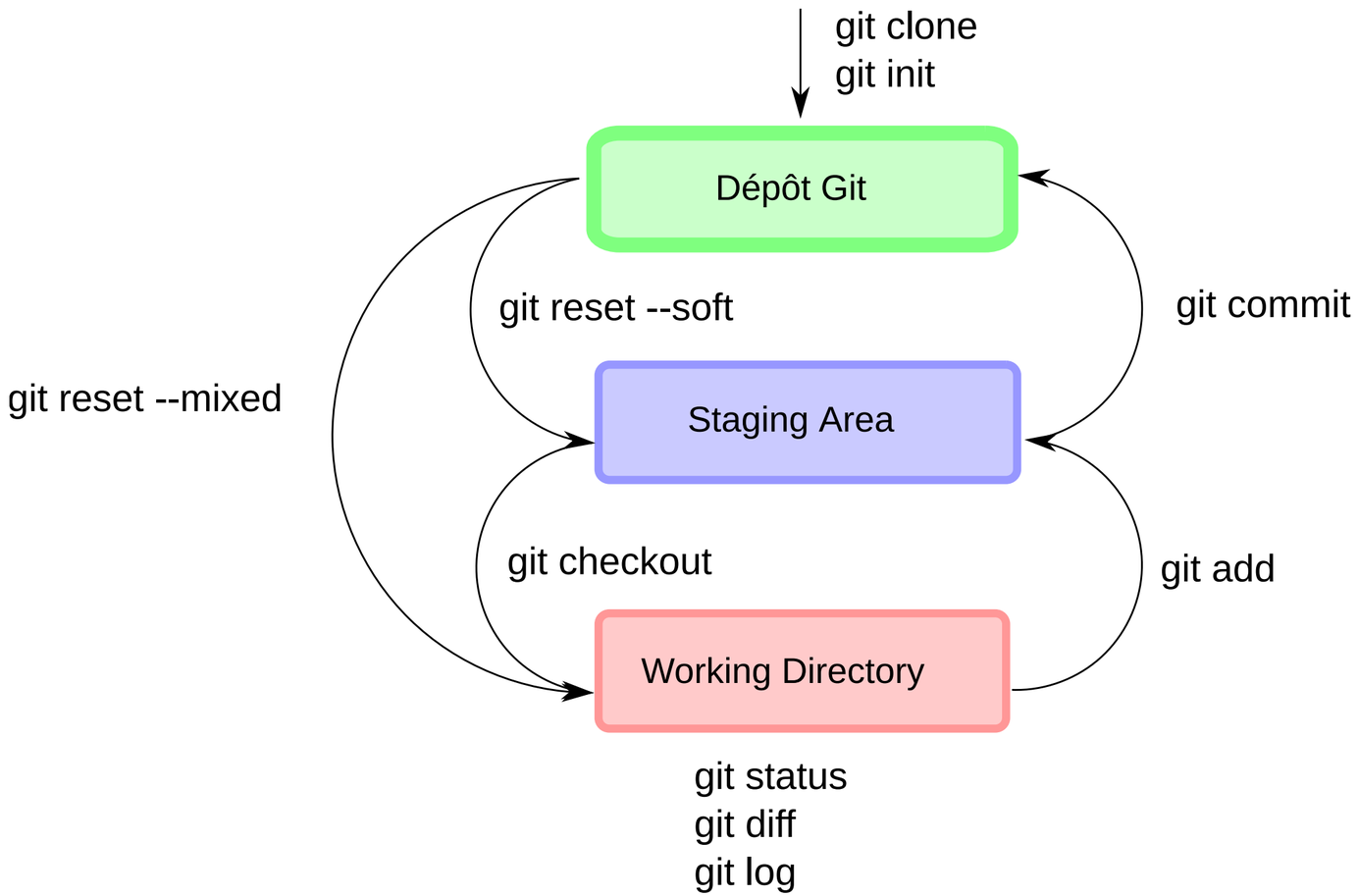


Plusieurs protocoles sont disponibles : Local, git, ssh/git, http(s)

```
git clone git://github.com/jeanbon/project.git
```

```
git clone https://gitlab.com/jeanbon/project.git
```

En résumé


git clone
git init
Dépôt Git
git reset --soft
git commit
git reset --mixed
Staging Area
git checkout
git add
Working Directory
git status
git diff
git log

Gestion des dépôts distants

Remote

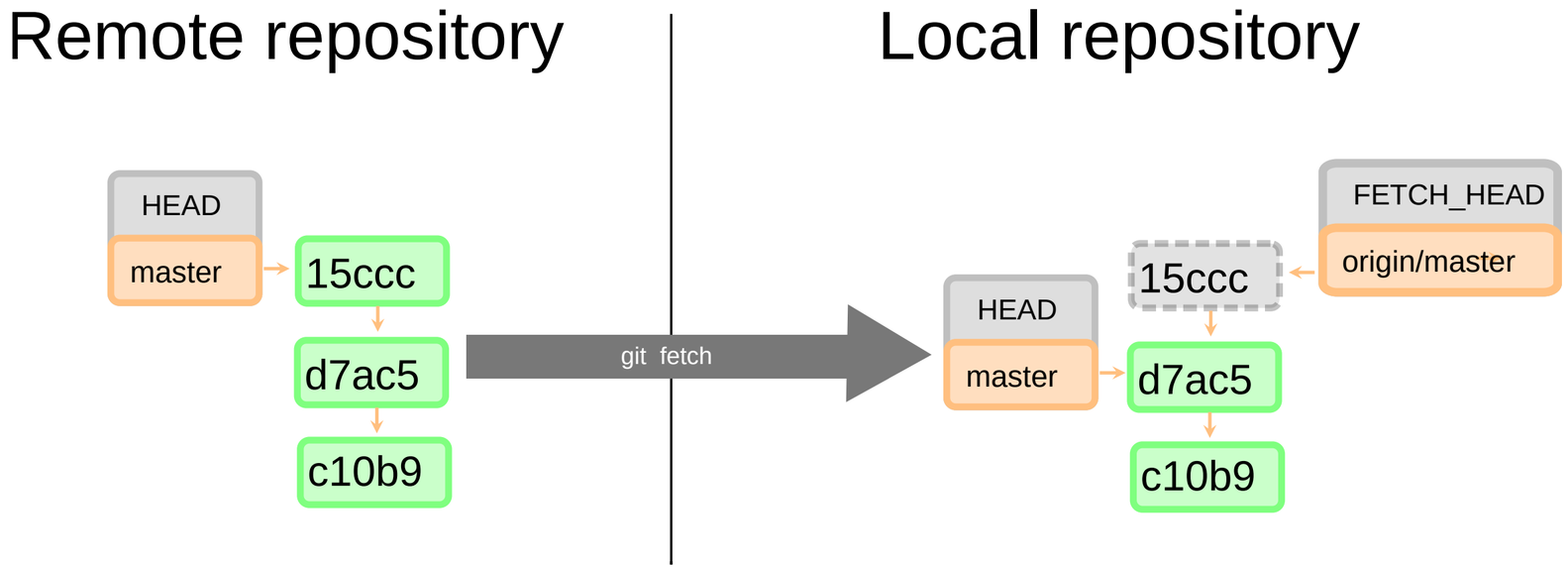
Mot clé pour identifier les repos distants.
`git remote [operation]`

Lister les dépôts : `git remote` / `git remote -v`

Ajouter un nouveau dépôt distant :
`git remote add [nom_court] [url]`
`git remote add github https://github.com/test.git`

Suppression d'un dépôt : `git remote rm github`

Fetch : Consulter les modifications distantes



Récupère les nouveaux commits.

Ne met pas à jour les **branches locales**.

Sans risques !

Pull: récupérer les modifications distantes

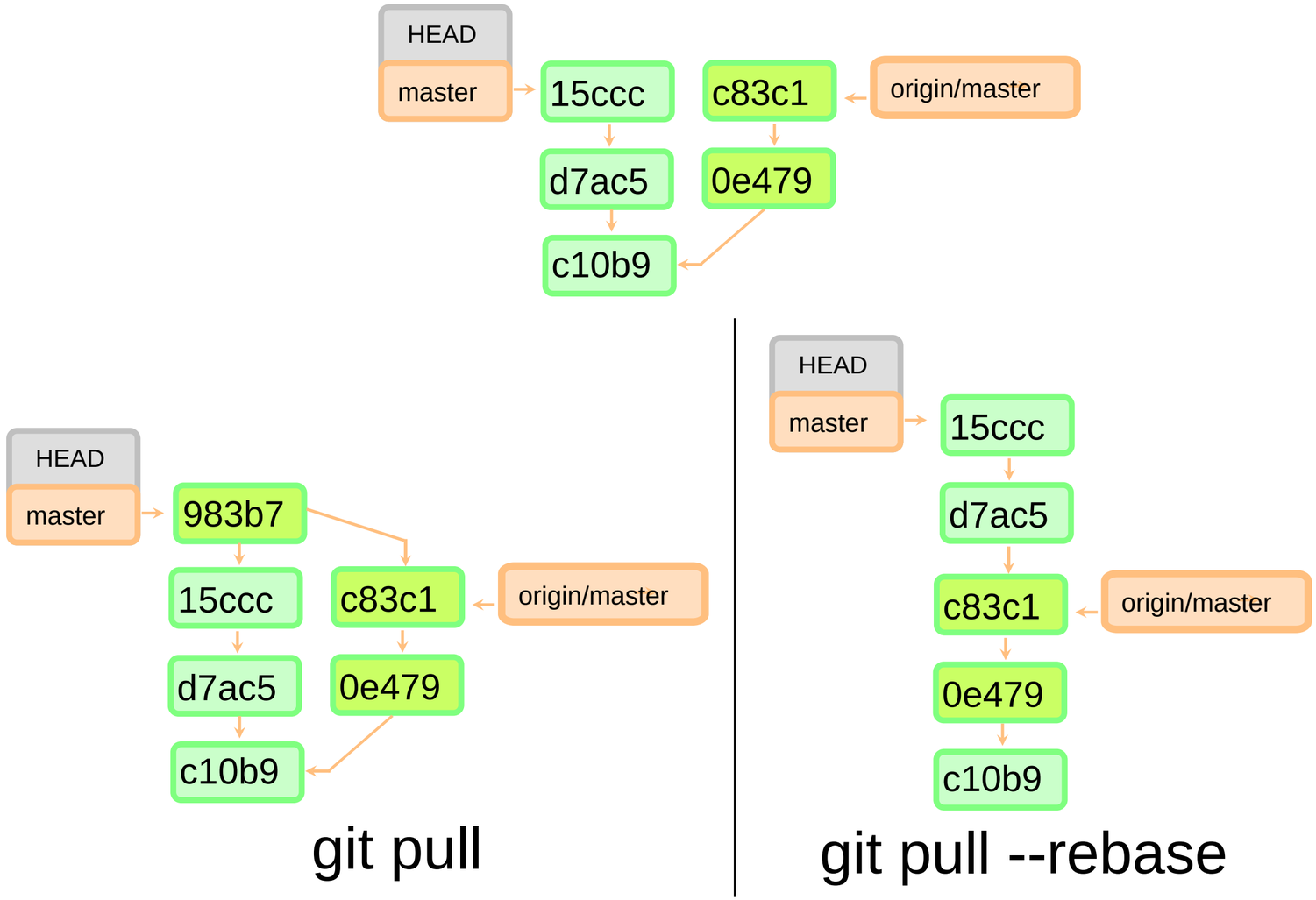
Avoir un workspace propre : pas de modification en cours.

git pull : récupère les modifications et les merge dans le dépôt local immédiatement. Intéressant si aucun commit local.

git pull --rebase : récupère les modifications et les replace dans l'ordre (date) par rapport au commit locaux.
Solution préconisée dans tous les cas.

```
git config --global pull.rebase = true
```

pull vs pull --rebase



Envoyer les modifications

`git push`

- par défaut pousse la modification sur la branche courante.
- rejeté si quelqu'un a fait un push avant.
- la fusion au préalable est obligatoire via `git pull --rebase`.

En résumé

- `git clone <url>` pour récupérer un nouveau dépôt.
- `git pull --rebase` pour récupérer des modifications serveurs (le workspace doit être propre).
- `git push` envoie les nouveaux commits (le workspace doit être à jour avec le serveur).

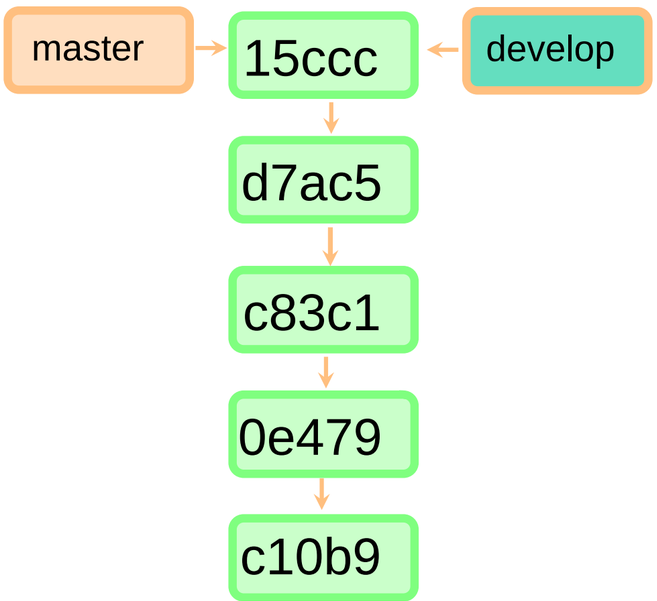
Utilisation des branches

1 branche = 1 référence

Une branche est un pointeur léger sur un commit.

Facile à créer et à détruire de manière instantanée.

Locale puis distante si push.



Création / récupération d'une branche

```
$ git branch <nom_branche>
```

```
$ git checkout <nom_branche>
```

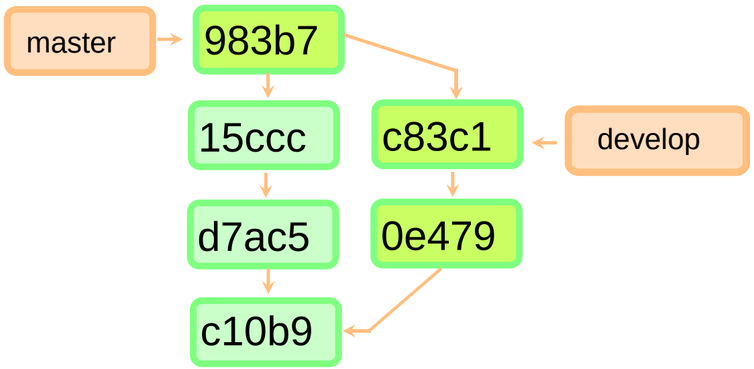
Publication d'une branche

```
$ git push -u origin <nom_branche>
```

Récupérer une branche

La fusion d'une branche se fait toujours par un **merge**.

Merge




```
git checkout master
```

```
git merge develop
```

Un nouveau commit de fusion est créé.

Une fois le travail fusionné la branche n'est plus néccessaire.

Tags

Les tags permmettent de créer une étiquette sur un point spécifique de l'historique de version.
Sert à mémoriser des choses importantes (release).
Immutable par rapport aux branches.

```
git tag <tagname>
```

Les tags sont locaux, il faut les envoyer vers le serveur avec la commande `git push --tags`.

En résumé

Les branches ne sont que des pointeurs.

Il ne faut pas hésiter à en abuser.

La fusion de branche se fait par un **merge**.

Workflow de développement

Les attentes

- Pouvoir développer sereinement.
- Favoriser le travail en équipe.
- Respecter les différentes phases d'un projet.

Le modèle de branche de Vincent Driessen

- Apporte des conventions de nommage.
- Apporte de la lisibilité dans l'arbre git.
- Colle au plus prêt des phases d'un projet.
- Adopté comme standard par les utilisateurs de git.
- Automatisé grâce à **git-flow**.

Les Branches principales

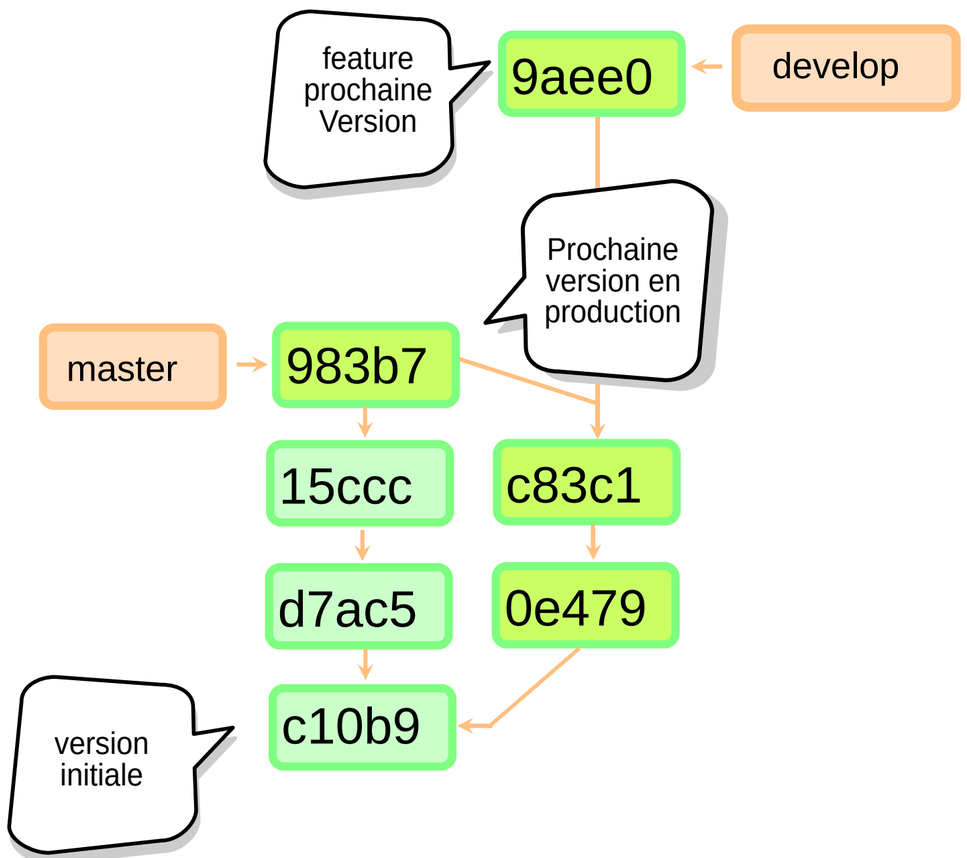
Master

Au plus proche de la production

Develop

Branche d'intégration / de développement

```
git flow init
```



Nouvelles fonctionnalités

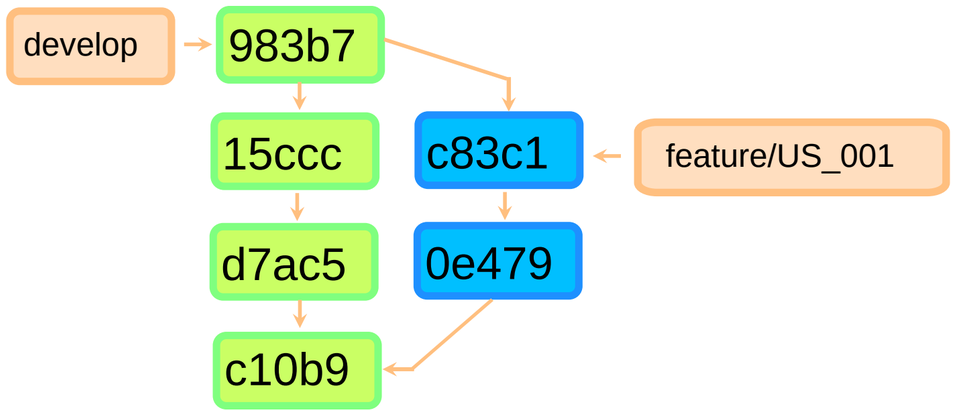
Feature

Couvre un périmètre fonctionnel

chaque feature est reportée par un merge sur la branche **Develop**

```
git flow feature start nom_feature
```

```
git flow feature finish nom_feature
```



Version prête pour recette

Release

Contient un ensemble de version

Le nom correspondra au tag

Lors de la fermeture les commits sont :

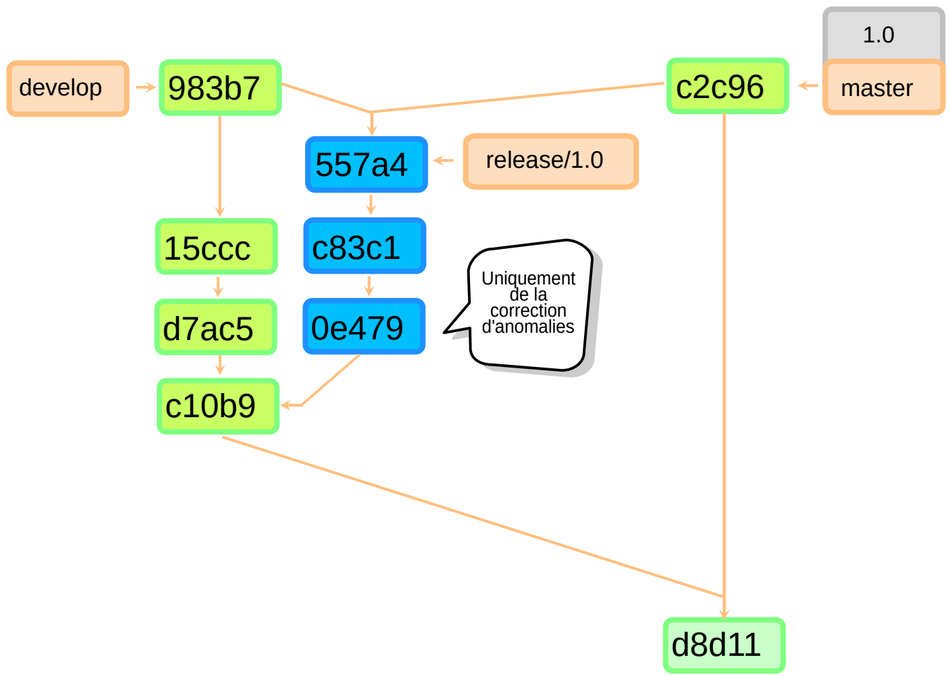
reportés dans **Develop**

reportés dans **Master**

un **Tag** est créé

```
git flow release start nom_release
```

```
git flow release finish nom_release
```



corrections de production

Hotfix

Ne contient que des correctifs

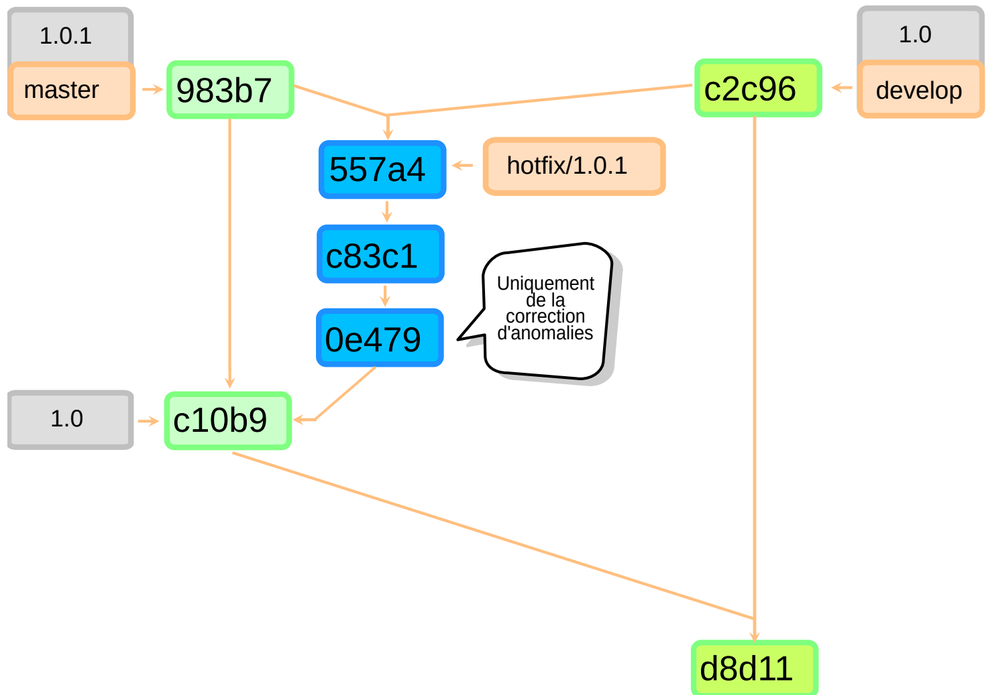
Le nom correspondra au tag

Lors de la fermeture les commits sont :

reportés dans **Develop**

reportés dans **Master**

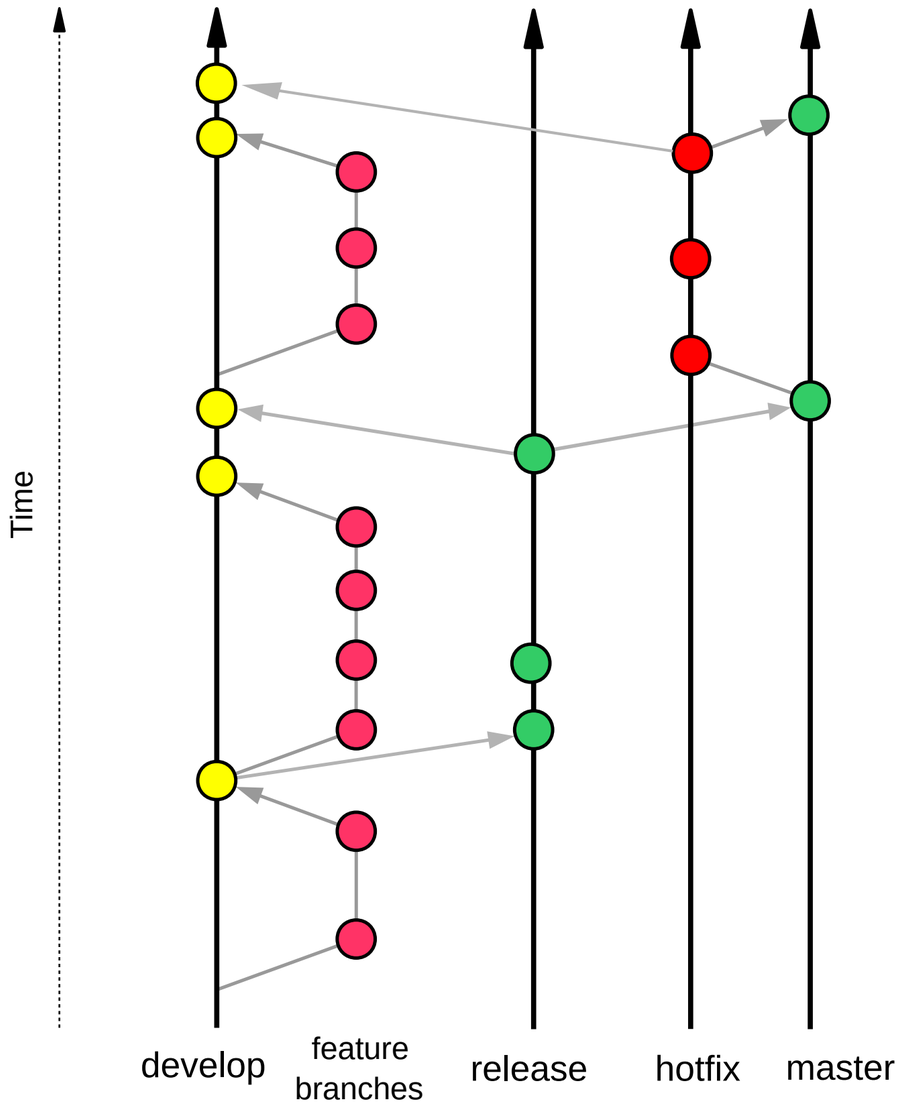
un **Tag** est créé




```
git flow hotfix start nom_hotfix
```

```
git flow hotfix finish nom_hotfix
```

Git Flow : vision globale



Gestion des sous-modules

Principe

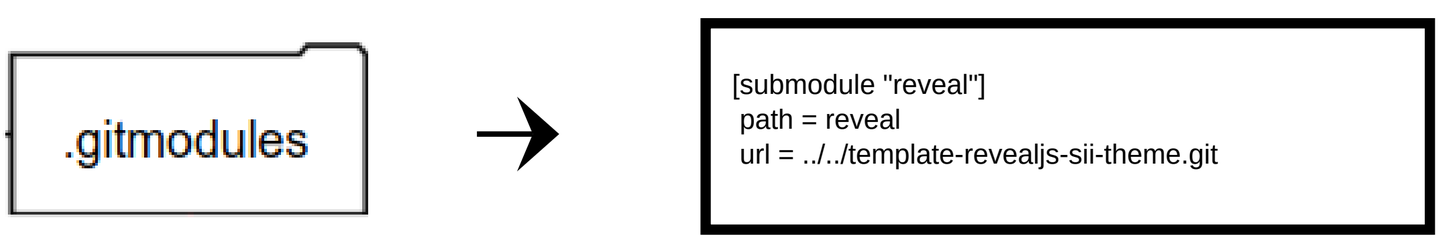
Les sous-modules sont utilisés pour **inclure** un dépôt git dans un autre dépôt.
Utile pour séparer le code dans différents dépôts.
Un **sous module** peut être ajouté à plusieurs dépôts.

Fonctionnement

L'ajout d'un sous module se fait via la commande suivante :
`git submodule add <URL> <local-path>`

Un fichier `.gitmodules` est créé dans le dépôt.

Se comporte comme un `git clone`.



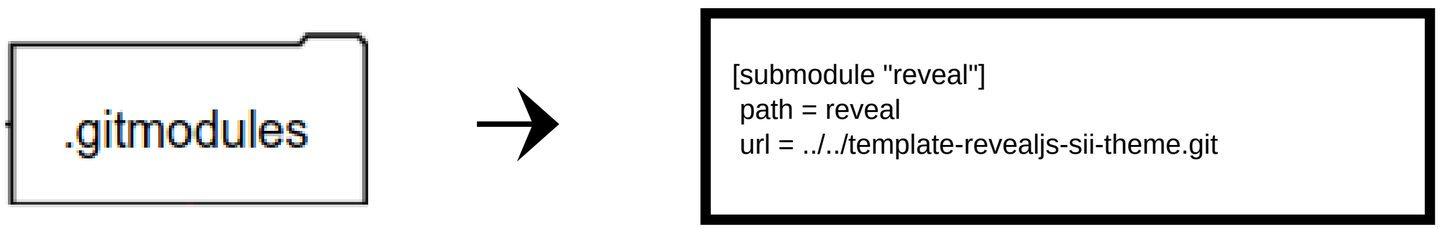
Configuration

Se comporte **comme un dépôt git**.

Par défaut clone la branche principale du projet.

Pour se positionner sur un tag, il faut exécuter la commande **git** dans le dossier du sous module.

Pensez à commiter le fichier `.gitmodules`.




```
$ cd reveal
$ git checkout 1.0.0
```

Mise à jour

La mise à jour s'exécute de cette manière :

```
git submodule update --remote --rebase
```

remote : pour chercher les modifications distantes.

rebase : pour rejouter l'ordre des commits.

Pensez à commiter le fichier .submodule pour figer la nouvelle version.

Clone

Un dépôt contenant un sous-module devra l'initialiser :

```
git submodule init
```

Initialiser et mettre à jour les sous-modules d'un coup :

```
git submodule update --init
```

Les sous-modules peuvent contenir des sous modules eux-même :

```
git submodule update --init --recursive
```

Conseils

Pas de modifications via le dossier sous-module mais directement dans le repository original.
Définir au plus tôt dans le projet les sous-modules.
Penser à suivre les modification distantes.
Utiliser des tags pour les sous module.

Bonnes pratiques

Reflog

Bouée de secours.

Historise tous les changements.

Permet de retrouver des commits supprimés !

```
$ git reflog
```

Stash

Sauvegarde les modifications non indexées sur une étagère

Utile pour récupérer des commits distants sans commit les modifications locales

`$ git stash <nom_stash>` (création)

`$ git stash pop <nom stash>` (supprime et applique)

`$ git stash apply <nom stash>` (applique le stash)

Revert

Annule un commit.

Un nouveau commit est créé pour restaurer l'état des fichiers.

Unique solution de retour arrière lorsque le commit a été envoyé sur le serveur distant.

```
$ git revert <sha>
```

Bisect

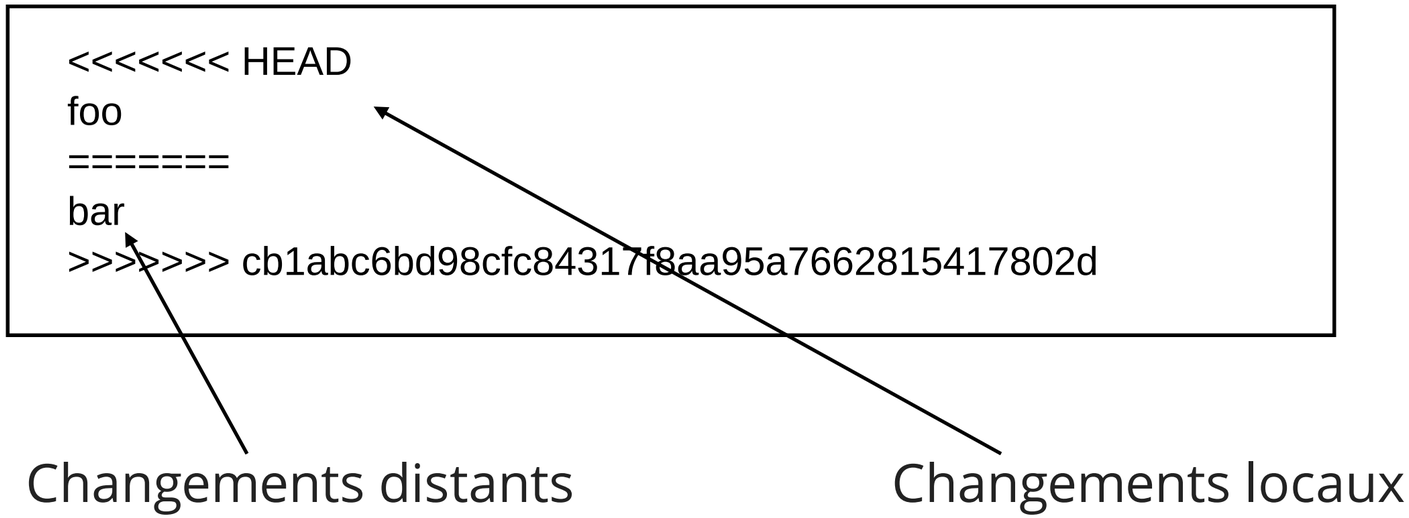
Compare des commits entre eux.

A partir d'une référence.

Très utile pour rechercher une anomalie dans le code.

```
$ git bisect start
$ git bisect bad
$ git bisect good <sha>
```

Gestion des conflits



Il faut faire un choix

Un nouveau commit doit être réalisé pour résoudre le conflit

```
$ git add .
```

```
$ git commit -m "fix(conflict)"
```

Conclusion

Vos avis

Quelques liens utiles

- Le site officiel : https://git-scm.com
- Cheatsheet : https://groupe-sii.github.io/cheat-sheets/git/index.html
- gitflow : https://github.com/nvie/gitflow
- git wiki : https://git.wiki.kernel.org

Apprendre Git En ligne

- Try git : https://git.wiki.kernel.org
- Learn git branching : https://learngitbranching.js.org/