

GIT

LES FONDAMENTAUX



2024

By Boris SAUVAGE

Git - Introduction

- Git est un système de gestion de version décentralisée
 - aussi appelé des DVCS (Distributed Version Control System)
 - Distribuer sous licence GNU espace V2
- Créée en 2005 par les équipes en charge du noyau Linux, en particulier Linus Torvalds, suite à une rupture des relations avec l'éditeur de BitKeeper
- Git va permettre de sauvegarder les versions d'un même fichier et suivre l'historique d'un projet
 - Il semble enregistrer les différences
 - Spécialisé pour les fichiers texte (code source, XML, JSON, YAML,...) mais il sait aussi gérer les fichiers binaires
- Système décentralisé qui va faciliter la collaboration entre plusieurs développeurs au sein d'un même projet.
- Quelques autres outils de gestion de version : BitKeeper, CVS, Subversion, ClearCase, Mercurial

Git - Vocabulaire

Quelques mots importants :

- **Repository (Dépôt)**
 - le repository est l'ensemble de l'historique d'un projet
- **Commit (Validation) :**
 - dans le contexte de Git, il s'agit de l'ensemble des fichiers d'un projet au moment où le commit a été effectué
 - Un commit est identifié par une empreinte (HASH) SHA-1 de 168 bits (ou 20 octets)
 - Exemple d'empreinte : 2ab13ea5406c1fee0267a9ba3cb61fe5d6557213
 - Dans l'utilisation, nous utiliserons les premiers digits pour mentionner un commit
- **Revision (révision):**
 - Une révision en Git est une notion très générale
 - Il s'agit de toute modification apportée au dépôt
 - Un commit est une révision
 - Un fichier modifié lors d'un git constitue aussi une révision de ce fichier
 - Toute révision dans Git est identifiée par une empreinte SHA-1

Git – les 3 zones

- Lorsqu'on utilise Git en local, on va principalement manipuler trois zones :

- **Working directory**

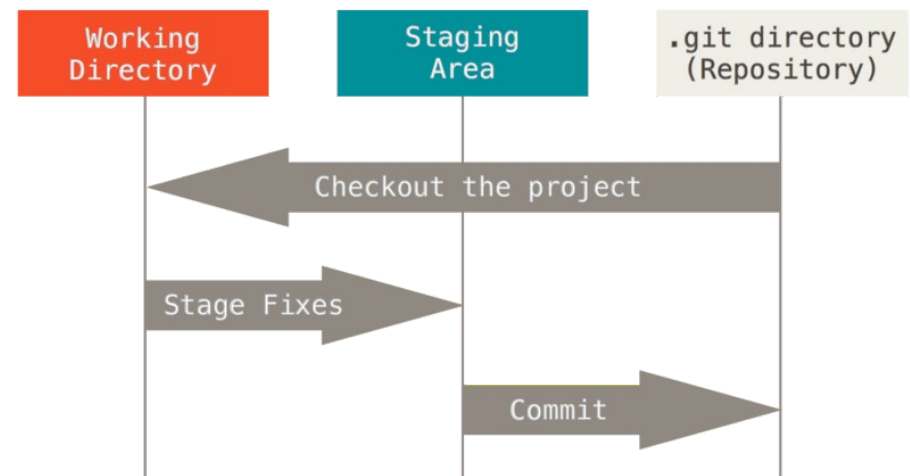
- Répertoire de travail
- Ce sont les fichiers que l'on modifiera présents sur le disque (code source)

- **Staging Area**

- Zone d'index
- Elle contient le contenu du prochain commit

- **Git repository**

- Dépôt Git
- Il contient tout l'historique des commit du projet



Git – Utilisation

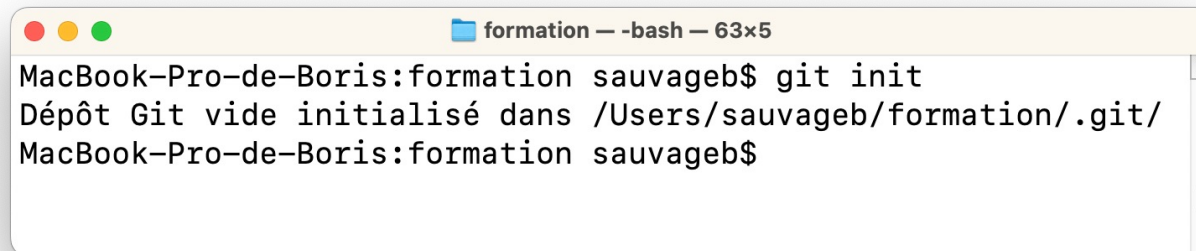
- Git fonctionne en ligne de commande
- Des outils graphiques existent (voir dernière partie)
 - ils ne font que lancer, dans les coulisses, des commandes Git
- La commande à utiliser est la commande **git**
- Elle se présente toujours sous la forme `git <commande> <paramètres>`
- on utilise **git** depuis un terminal ou une console
 - sous windows, on utilisera de préférence Git Bash (installé avec Git)

Git – La création d'un dépôt

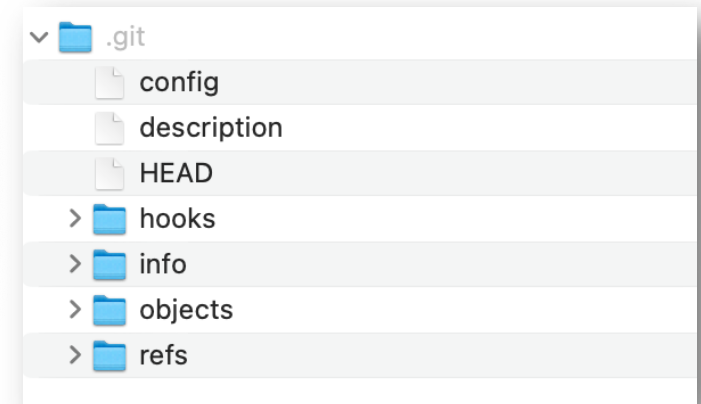
Pour créer un dépôt, on utilise la commande **git init**

```
git init
```

- Crée un dépôt Git dans le répertoire en cours



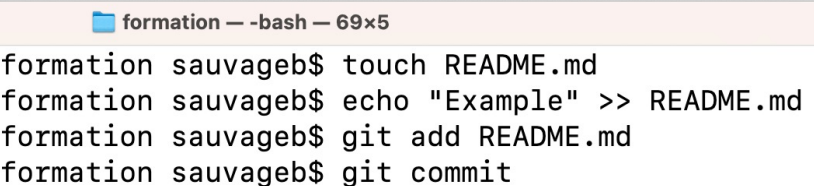
```
MacBook-Pro-de-Boris:formation sauvageb$ git init
Dépôt Git vide initialisé dans /Users/sauvageb/formation/.git/
MacBook-Pro-de-Boris:formation sauvageb$
```



Git – L'ajout de fichiers

- Lorsque l'on crée un fichier dans le répertoire de travail, il est à l'état **untracked** (non suivi)
 - Git ne tiendra pas compte de ce fichier, donc il est dans cet état non suivi
- Pour intégrer ce fichier au suivi de version, on va le basculer dans la zone d'index pour qu'il puisse faire partie du prochain commit
 - Le fichier sera alors suivi par Git et perdra son statut **untracked**
 - Le fichier sera indexé
- Pour ajouter un fichier à la zone d'index, on utilise la commande **git add**

```
git add <nom_fichier>
```



```
formation — -bash — 69x5
formation sauvageb$ touch README.md
formation sauvageb$ echo "Example" >> README.md
formation sauvageb$ git add README.md
formation sauvageb$ git commit
```

Git – Le commit 1/4

- Une fois que les fichiers ont été mis dans la zone d'index, on peut créer un commit avec la commande **git commit**
- L'éditeur de texte s'ouvrira, il sera nécessaire de saisir le **titre du commit** dans la première ligne du fichier
 - Message précisant la finalité du commit (Résolution anomalie, ajout d'une fonctionnalité)
 - Dans l'idéal, limité à 50 caractères
 - Dans les lignes suivantes, on pourra ajouter une description
- Une fois le texte saisi et l'éditeur de texte quitté, le commit écrit

Git – Le commit 2/4

- Démonstration lorsque l'on fait un commit :

```
formation — -bash — 69x5
formation sauvageb$ touch README.md
formation sauvageb$ echo "Example" >> README.md
formation sauvageb$ git add README.md
formation sauvageb$ git commit
```

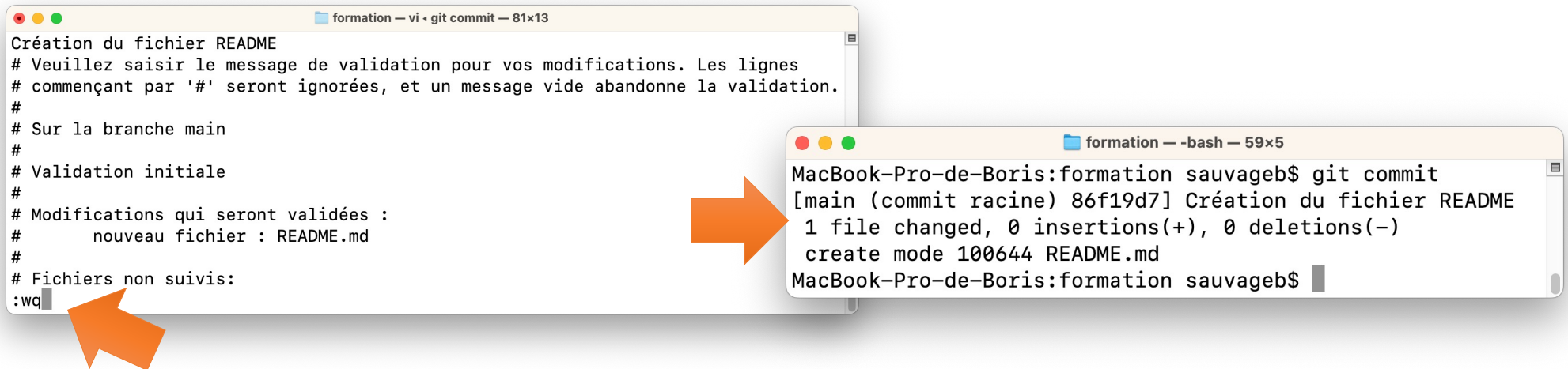


```
formation — vi - git commit — 81x13
Création du fichier README
# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Sur la branche main
#
# Validation initiale
#
# Modifications qui seront validées :
#     nouveau fichier : README.md
#
# Fichiers non suivis:
-- INSERT --
```

- L'éditeur de texte de base de Git est **VI**
 - Vous devrez donc vous mettre en mode INSERT pour saisir les informations
 - Ici, j'ai uniquement précisé le titre

Git – Le commit 3/4

- Démonstration lorsque l'on fait un commit :



```
formation — vi • git commit — 81x13
Création du fichier README
# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Sur la branche main
#
# Validation initiale
#
# Modifications qui seront validées :
#   nouveau fichier : README.md
#
# Fichiers non suivis:
:wq
```

```
formation — -bash — 59x5
MacBook-Pro-de-Boris:formation sauvage$ git commit
[main (commit racine) 86f19d7] Création du fichier README
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 README.md
MacBook-Pro-de-Boris:formation sauvage$
```

- Afin de sauvegarder le commit, il faut indiquer **:wq** (write & quit)
 - Le message est ensuite créé

Git – Le commit 4/4

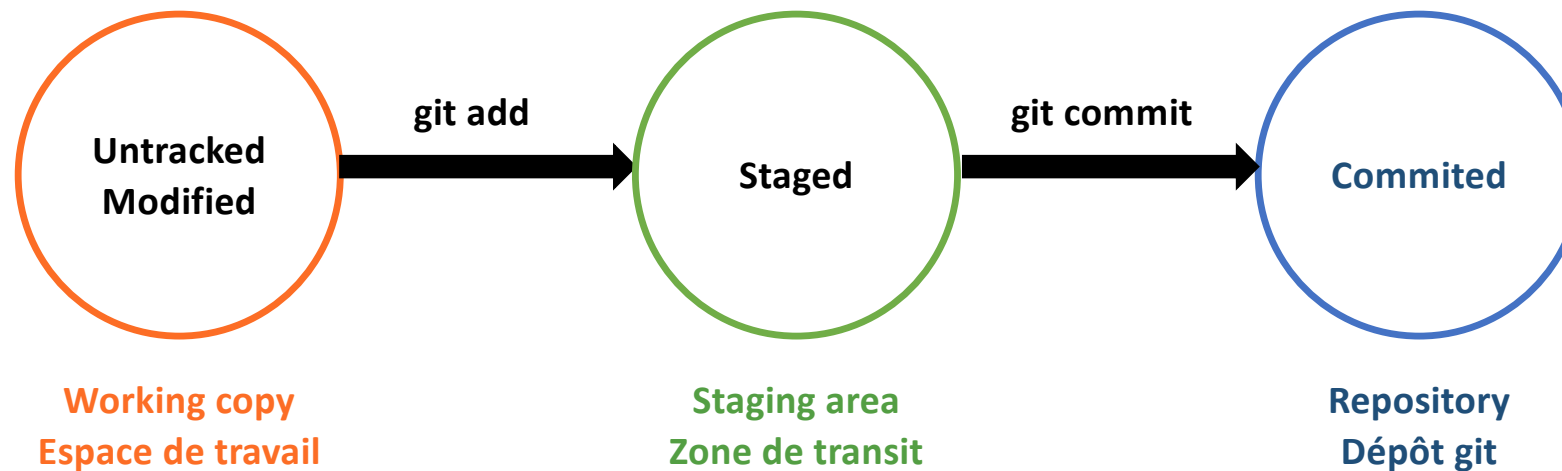
- La création d'un commit créer donc, une nouvelle version du projet au sein du dépôt
- Au final, après 4 commit, on aurait la représentation suivante :

git log

```
formation — -bash — 64x25
* commit 34b432de0506ce13b7d9b287ec65e1731234363b (HEAD -> main)
| Author: sauvageb <sauvageboris.pro@gmail.com>
| Date:   Sun Mar 10 13:25:52 :   +0100
|
|     Ajout d'un titre 3 dans README.md
|
* commit dda57a0ca45c0104fb2dbcf0363110283d60c6d2
| Author: sauvageb <sauvageboris.pro@gmail.com>
| Date:   Sun Mar 10 13:25:35   +0100
|
|     Ajout d'un titre 2 dans README.md
|
* commit d45c13fae36a6129a85b95e180c94ed10194f2a3
| Author: sauvageb <sauvageboris.pro@gmail.com>
| Date:   Sun Mar 10 13:23:50   +0100
|
|     Ajout d'un titre dans README.md
|
* commit 86f19d703027418f40450ef5385ffa6de34e595e
| Author: sauvageb <sauvageboris.pro@gmail.com>
| Date:   Sun Mar 10 13:04:56   +0100
|
|     Création du fichier README
|
MacBook-Pro-de-Boris:formation sauvageb$
```

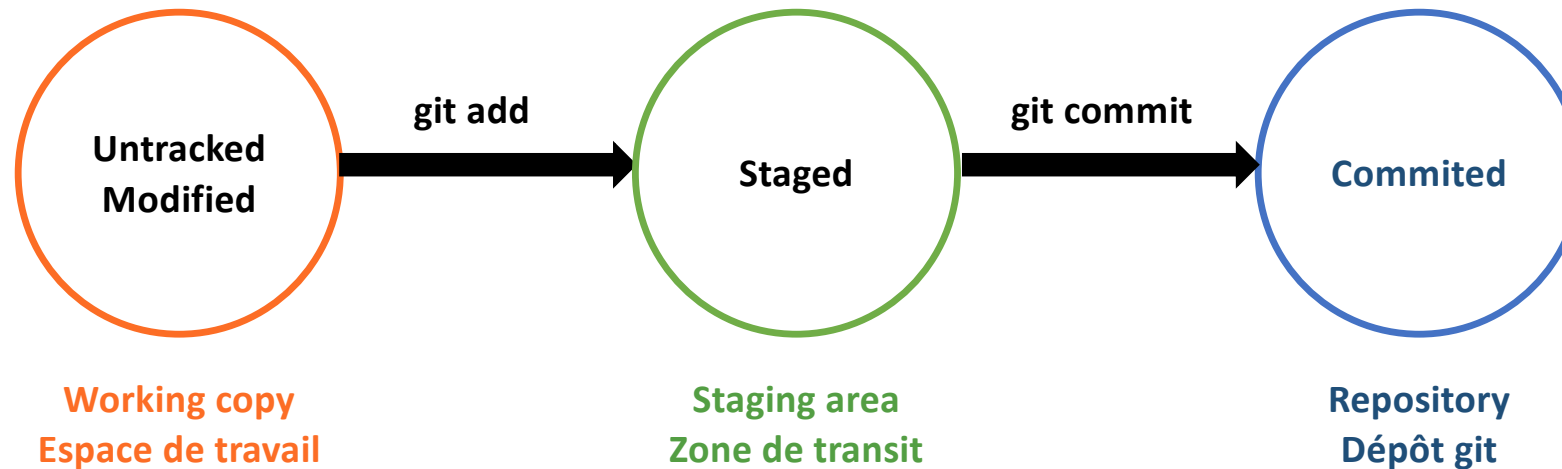
Git - États des fichiers 1/3

- Un fichier doit être explicitement ajouté au dépôt Git



- **Commit** : ensemble cohérent de modifications
- **Working copy** : contient les modifications en cours (c'est le répertoire courant)
- **Staging area** : Zone de transit qui contient la liste des modifications effectuées dans la working copy qu'on veut inclure dans le prochain commit
- **Repository** : ensemble des commits du projet (et les branches, les tags ou libellés)

Git - États des fichiers 2/3



- **Untracked (Fichiers non suivis) /Modified**

- Nouveaux fichiers ou fichiers modifiés
 - Pas pris en compte pour le prochain commit

- **Unmodified/Committed**

- Aucune modification pour le prochain commit

- **Staged (Fichiers Indexés)**

- Fichiers ajoutés, modifiés, supprimés ou déplacés
 - Pris en compte pour le prochain commit

Git - États des fichiers 3/3

- Vérifions le contenu de notre dépôt

```
git status
```

- Commençons par créer un fichier file.txt

```
echo "code source 1" > file.txt
```

- Vérifions le contenu de notre dépôt

```
git status
```

- Ajout du fichier dans la zone de transit

```
git add file.txt
```

```
git add .
```

```
git add --all
```

- Vérifions le contenu de notre dépôt

```
git status
```

- Création du commit

```
git commit -m "mon premier commit"
```

**Untracked
Modified**

Staged

Committed

Créer un Repository git

- Configuration du compte git

```
git config --global user.name "monNom"  
git config --global user.email "mon@email"
```

- Désactivation de la coloration dans la console

```
git config --global color.ui false
```

- Pour consulter la liste de configurations (et vérifier les modifications)

```
git config --list
```

- Pour vérifier la valeur d'une propriété de configuration

```
git config user.name
```



Annuler un Commit

- Création d'un commit (que l'on annulera)

```
git checkout test  
echo "creating file2">>file2.txt  
git add file2.txt  
git commit -m " creating file2.txt "  
echo "updating file2">>file2.txt  
git commit -am " updating file2.txt "
```

- Annuler le commit ayant comme message « updating file2 »

```
git revert idDuCommit
```

- Modifier le message du commit d'annulation si besoin. Cliquer sur Echap, saisir :wq et cliquer sur Entrée
- Vérifier l'annulation avec :

```
git log --oneline  
cat file2.txt
```


Notions sur HEAD

- Référence symbolique au dernier commit sur la branche courante dans votre dépôt local
 - c'est le commit sur lequel vous travaillez actuellement
- HEAD change dynamiquement lorsque vous changez de branche ou que vous faites un commit
 - Il pointe toujours vers le dernier commit de la branche courante
- **git log** permet de voir où HEAD est actuellement pointé
 - Le commit sur lequel HEAD pointe est généralement marqué en haut de la liste des commits

Revenir en arrière - HEAD

- Pour aller dans le passé dans son projet, on dispose de deux commandes
 - **git checkout**
 - **git reset**
- On va utiliser l'une ou l'autre, selon la finalité du retour en arrière
- En Git, HEAD fait référence au commit qui sera le parent du prochain commit
 - C'est l'emplacement où l'on se trouve actuellement
- HEAD pointe souvent vers une branche (main par exemple), mais peut aussi pointer vers un commit particulier

Revenir en arrière - checkout

- La commande **git checkout** permet de déplacer **HEAD** vers un autre emplacement
 - Cela réinitialisera le **Working directory** et la **zone d'index**
 - On ne pourra pas faire **git checkout** si des modifications sont en cours
- On peut donc utiliser **git checkout** pour consulter l'état d'un projet dans une ancienne version, sans avoir à modifier la structure même de l'historique
- Exemples :
 - **git checkout HEAD~2**
 - **git checkout HEAD^^**
 - **git checkout 569gf42**

Revenir en arrière – reset 1/2

- La commande **git reset**, quant à elle, n'agira pas sur HEAD mais sur la branche sur laquelle on se trouve
 - Elle déplacera la branche sur un autre emplacement, et donc HEAD, vue que HEAD pointe sur la branche
 - Elle permet donc de réinitialiser son projet sur une version antérieure
- Il y a trois modes dans le **reset** qui iront plus ou moins loin dans la réinitialisation
 - **Mode soft** avec l'option **--soft**
 - **Mode mixed** avec l'option **--mixed (par défaut)**
 - **Mode hard** avec l'option **--hard (ATTENTION : option destructrice)**

Revenir en arrière – reset 2/2

Voci la commande des 3 possibilités :

- **mode mixed (par défaut)** : annule le commit et garde les modifications dans le working directory

```
git reset --mixed
```

- **mode soft** : annule le commit et garde les modifications dans la staging area

```
git reset --soft idDuCommit
```

- **mode hard** : annule le commit et **ne garde pas** les modifications

```
git reset --hard idDuCommit
```

En profondeur avec rebase

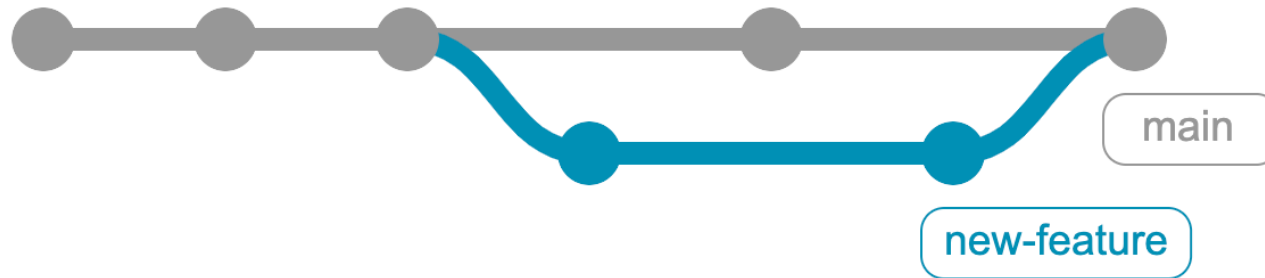
- La commande **git rebase** permet de réécrire complètement l'historique des commits
- Exemple : retravailler l'historique en faisant intervenir les 4 derniers commits
 - **git rebase -i main~4**
- Plusieurs possibilités seront proposés. Pour chaque commit, il conviendra d'indiquer l'action voulue :
 - **pick** : pas de changement
 - **reword** : changement du message
 - **edit** : changement complet du commit
 - **squash** : fusion du commit avec le précédent
 - **fixup** : idem que squash en conservant le message précédent
 - **exec** : exécuter une commande sur le commit précédent (penser commande de test qui retournerai un code 0 ou un code d'erreur)
 - **drop** : supprimer ce commit
- Il s'agit d'une commande très puissante

Branches locales 1/2

- D'un point de vue purement technique, en Git, une branche est un pointeur vers un commit
- Par défaut, on a une branche principale nommée **main** ou **master**
- On dispose aussi d'une branche courante nommé **HEAD**
 - C'est la branche sur laquelle on travail
- La branche principale ne doit jamais faire l'objet de modifications directe
 - C'est la branche utilisée pour les livraisons
- La création d'une branche par fonctionnalité ou modification est une bonne pratique

Branches locales 2/2

- Les branches permettent de pouvoir travailler sur une copie de l'historique, sans risquer d'altérer un code déjà testé et validé

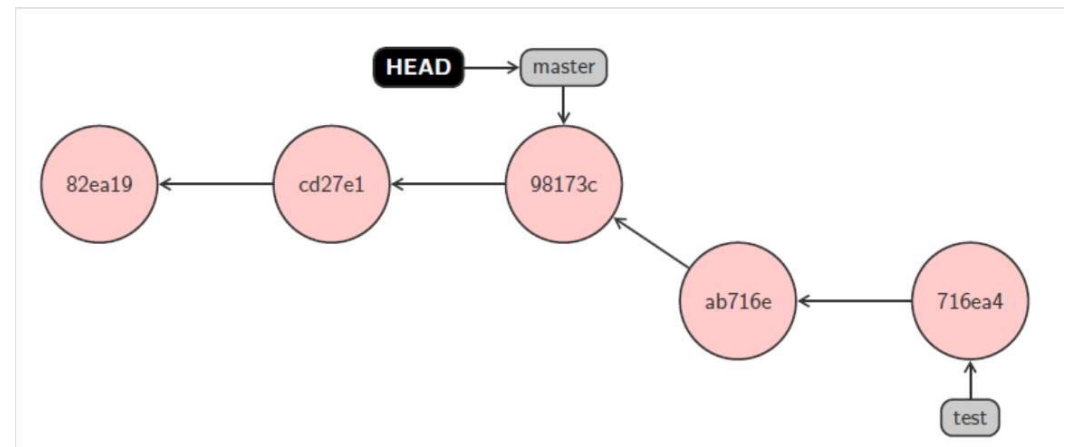


- On a ici deux branches :
 - **main**
 - **new-feature**
- On a aussi fait un merge de **new-feature** dans **main**

Commit sur une branche

- Un commit va toujours se faire sur la branche courante

```
git checkout test  
  
echo "code source 5">>file.txt  
  
git commit -am " mon cinquième commit"  
  
echo "code source 6">> file.txt  
  
git commit -am "mon sixième commit"  
  
git log --oneline  
  
git checkout master  
  
git log --oneline
```



- En créant une branche, cette dernière pointe sur le commit à partir duquel elle a été créée
- En faisant un Commit à partir de la branche créée, cette dernière dévie de la branche principale

Opérations de base sur une branche

- On peut créer une branche et basculer dessus

```
git checkout -b develop
```

- On peut renommer une branche avec l'option `-m` (**devBranch en develop**)

```
git branch -m devBranch develop
```

- On peut supprimer une branche vide (ou fusionnée) avec l'option `-d`

```
git branch -d branchToDelete
```

- Pour forcer la suppression d'une branche, on peut utiliser l'option `-D`

```
git branch -D branchToDelete
```

Fusion de branches

Problématique

- Lors de l'élaboration d'un projet, plusieurs branches seront créées, chacune pour une tâche bien particulière

Solution

- Fusionner les branches et rapatrier les modifications d'une branche dans une autre
- On peut fusionner deux branches pour en combiner les modifications
- La fusion se fait vers la branche courante
- Deux cas possibles de fusion :
 - **Sans conflit :**
 - fast forward : sans commit de merge
 - non fast forward (avec l'option --no-ff) : avec un commit de merge
 - **Avec conflit :**
 - avec un commit de merge

Fusion : Fast Forward

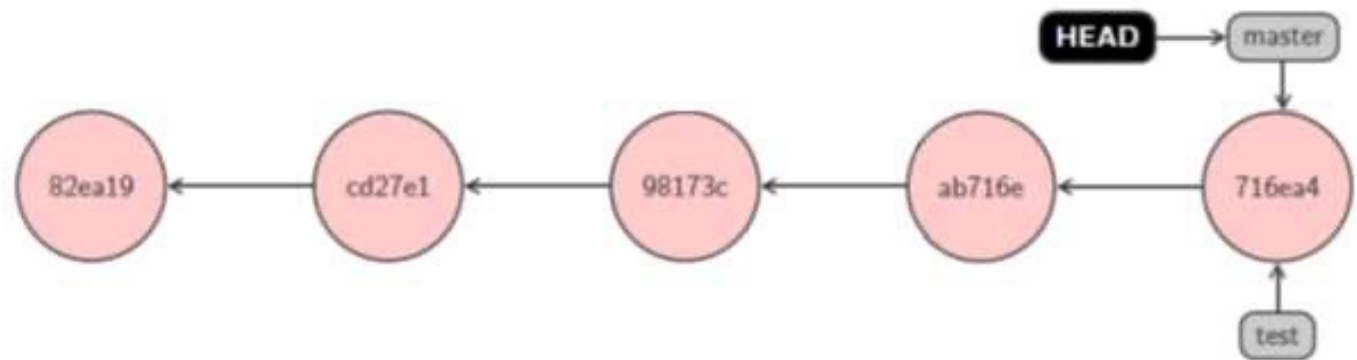
- Cas simple / automatique
- Quand il n'y a pas d'ambiguïté sur l'historique

```
git checkout master
```

```
git log --oneline
```

```
git merge test
```

```
git log --oneline
```



Remarque

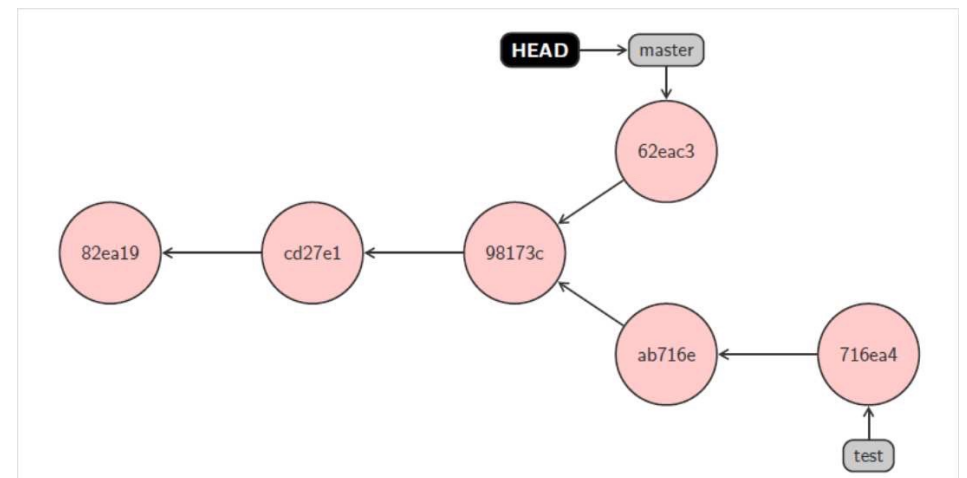
- La branche courante ne change pas après une fusion
- La branche fusionnée continue à exister

Fusion : Non Fast Forward

- Quand il y a ambiguïté sur l'historique ou un conflit : lorsque deux branches à fusionner contiennent des modifications sur le même fichier
- Il faut résoudre le conflit en modifiant le(s) fichier(s) de conflit
- Faire un Commit de merge

```
git checkout test
echo "code source 7" >> file.txt
git commit -am " mon septième commit "
echo "code source 8" >> file.txt
git commit -am " mon huitième commit "
git log --oneline
git checkout master
echo "code source 9" >> file.txt
git commit -am " mon neuvième commit "
git log --oneline
git merge test
```

Conflit détecté lors d'une demande de merge !

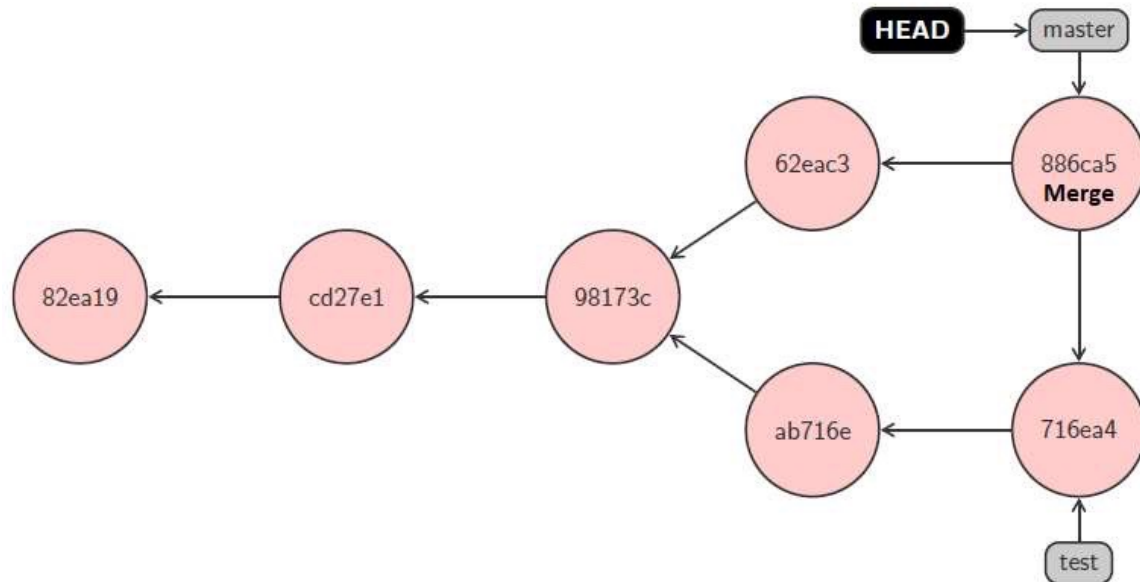


Fusion : Non Fast Forward

- Résolution manuelle du conflit

```
code source 1
code source 2
code source 3
code source 4
code source 5
code source 6
<<<<<< HEAD
code source 9
=====
code source 7
code source 8
>>>>>> test
```

nano file.txt



- Faire un Commit de merge `git commit -am "Résolution du conflit"`

La branche test ne bougera pas

- On peut annuler l'opération sans faire le commit de merge (après la détection d'un conflit)

`git merge --abort`

`git reset --merge`

`git reset --hard HEAD`

Les tags 1/2

Problématique

- Pour accéder à un commit qui présente une version importante de notre projet
- Il faut chercher le commit en question dans tous les Commits, et ensuite faire git checkout

Solution

- Utiliser les étiquettes (tags)

- Une étiquette permet de marquer un Commit/une version de notre application

```
git tag -a v1 -m "première version du projet"
```

- Création d'un tag sur un commit spécifique

```
git tag -a v1 943d727 -m "deuxième version du projet"
```

Les tags 2/2

- Rechercher un tag

```
git tag --list
```

- Se positionner sur un tag

```
git checkout v1  
git log --oneline
```

- Supprimer un tag

```
git tag v1 --delete
```


Le fichier .gitignore

- Git peut ignorer des fichiers du répertoire de travail en utilisant le fichier .gitignore
- Création du fichier .gitignore pour contenir les fichiers à ignorer

```
echo informatique.txt >> .gitignore  
echo *.html >> .gitignore  
echo view/* >> .gitignore  
echo java >> informatique.txt
```

- Désormais, git status affiche uniquement les fichiers non précisé dans le .gitignore
 - Tous les fichiers avec l'extension html sont ignorés
 - Tous les fichiers du répertoire view

Dépôt distant 1/2

- Les dépôts distants sont souvent des site hébergeur : GitHub, Bitbucket et GitLab
- Définir un dépôt distant

```
git remote add origin ../firstGitBare
```

- Afficher la liste des dépôts distants

```
git remote
```

- Afficher les branches distantes

```
git branch -r
```

Dépôt distant 2/2

- Envoyer (publier) la branche master sur le dépôt distant

```
git push origin main
```

- Pour supprimer un remote

```
git remote remove nomRemote
```

- Pour renommer un remote

```
git remote rename oldName newName
```

Cloner un dépôt distant

- Se placer dans le parent du dépôt courant

```
cd ..
```

- Cloner le dépôt firstGitBare dans firstGitClone

```
git clone firstGitBare firstGitClone
```

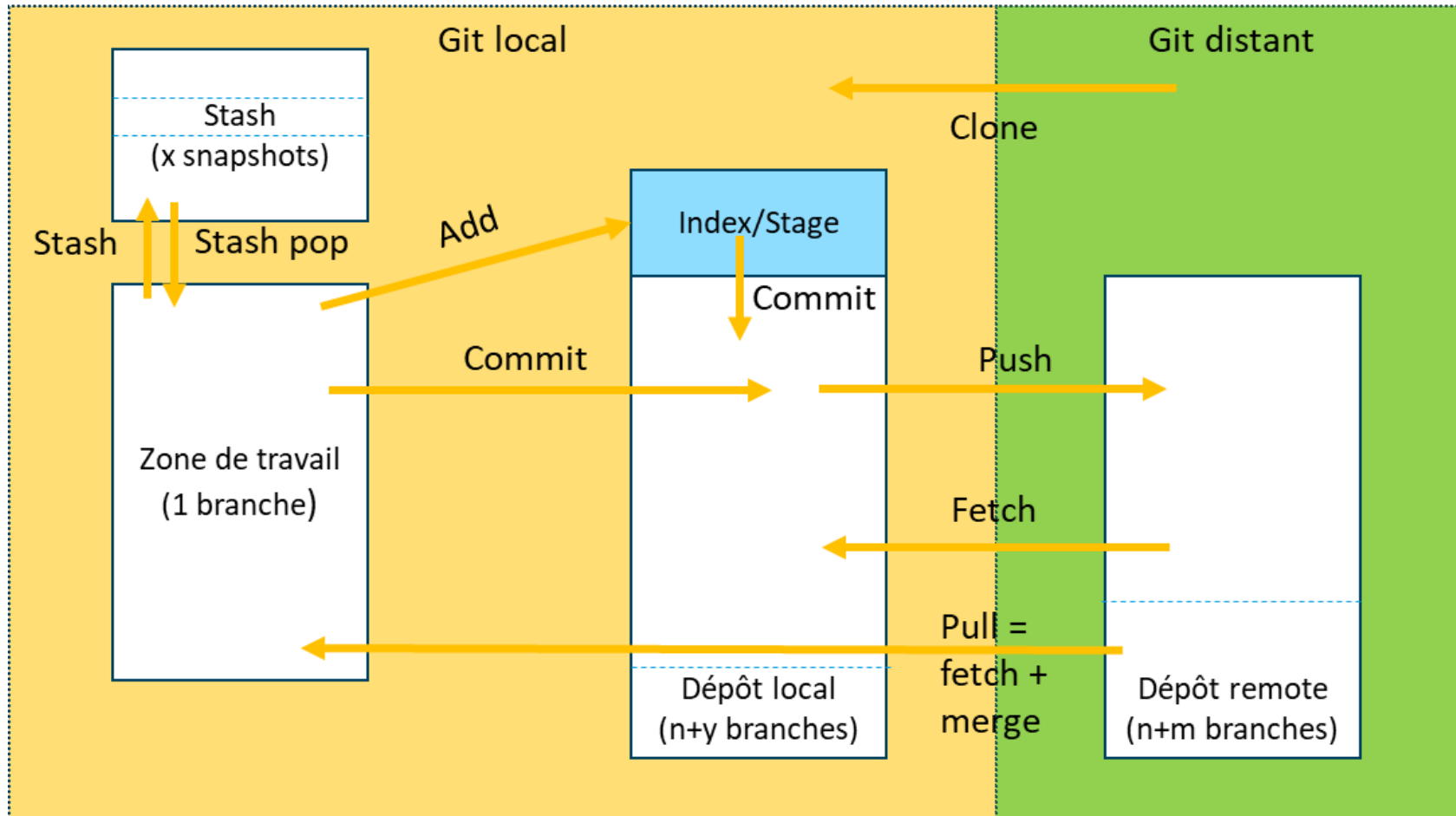
- Se placer dans le répertoire cloné

```
cd firstGitClone
```

- Vérifier le dépôt distant et les commits

```
git remote -v  
git log --oneline
```

Résumé



Aide mémoire 1/3

CONFIGURATION DES OUTILS

Configurer les informations de l'utilisateur pour tous les dépôts locaux

```
$ git config --global user.name "[nom]"
```

Définit le nom que vous voulez associer à toutes vos opérations de commit

```
$ git config --global user.email "[adresse email]"
```

Définit l'email que vous voulez associer à toutes vos opérations de commit

```
$ git config --global color.ui auto
```

Active la colorisation de la sortie en ligne de commande

```
git remote add origin [url] => ajout d'une URL en dépôt  
git remote rm origin => supprimer le chemin du dépôt  
git remote -v => afficher le dépôt actif
```

CRÉER DES DÉPÔTS

Démarrer un nouveau dépôt ou en obtenir un depuis une URL existante

```
$ git init [nom-du-projet]
```

Crée un dépôt local à partir du nom spécifié

```
$ git clone [url]
```

Télécharge un projet et tout son historique de versions

EXCLURE DU SUIVI DE VERSION

Exclure des fichiers et chemins temporaires

```
*.log  
build/  
temp-*
```

Un fichier texte nommé `.gitignore` permet d'éviter le suivi de version accidentel pour les fichiers et chemins correspondant aux patterns spécifiés

```
$ git ls-files --other --ignored --exclude-standard
```

Liste tous les fichiers exclus du suivi de version dans ce projet

Aide mémoire 2/3

EFFECTUER DES CHANGEMENTS

Consulter les modifications et effectuer une opération de commit

\$ git status

Liste tous les nouveaux fichiers et les fichiers modifiés à commiter

\$ git diff

Montre les modifications de fichier qui ne sont pas encore indexées

\$ git add [fichier]

Ajoute un instantané du fichier, en préparation pour le suivi de version

\$ git diff --staged

Montre les différences de fichier entre la version indexée et la dernière version

\$ git reset [fichier]

Enleve le fichier de l'index, mais conserve son contenu

\$ git commit -m "[message descriptif]"

Enregistre des instantanés de fichiers de façon permanente dans l'historique des versions

REFAIRE DES COMMITS

Corriger des erreurs et gérer l'historique des corrections

\$ git reset [commit]

Annule tous les commits après '[commit]', en conservant les modifications localement

\$ git reset --hard [commit]

Supprime tout l'historique et les modifications effectuées après le commit spécifié

SYNCHRONISER LES CHANGEMENTS

Référencer un dépôt distant et synchroniser l'historique de versions

\$ git fetch [nom-de-depot]

Récupère tout l'historique du dépôt nommé

\$ git merge [nom-de-depot]/[branche]

Fusionne la branche du dépôt dans la branche locale courante

\$ git push [alias] [branche]

Envoie tous les commits de la branche locale vers GitHub

\$ git pull

Récupère tout l'historique du dépôt nommé et incorpore les modifications

Aide mémoire 3/3

ENREGISTRER DES FRAGMENTS

Mettre en suspens des modifications non finies pour y revenir plus tard

\$ git stash

Enregistre de manière temporaire tous les fichiers sous suivi de version qui ont été modifiés ("remiser son travail")

\$ git stash pop

Applique une remise et la supprime immédiatement

\$ git stash list

Liste toutes les remises

\$ git stash drop

Supprime la remise la plus récente

VÉRIFIER L'HISTORIQUE DES VERSIONS

Suivre et inspecter l'évolution des fichiers du projet

\$ git log

Montre l'historique des versions pour la branche courante

\$ git log --follow [fichier]

Montre l'historique des versions, y compris les actions de renommage, pour le fichier spécifié

\$ git diff [premiere-branche]...[deuxieme-branche]

Montre les différences de contenu entre deux branches

\$ git show [commit]

Montre les modifications de métadonnées et de contenu incluses dans le commit spécifié

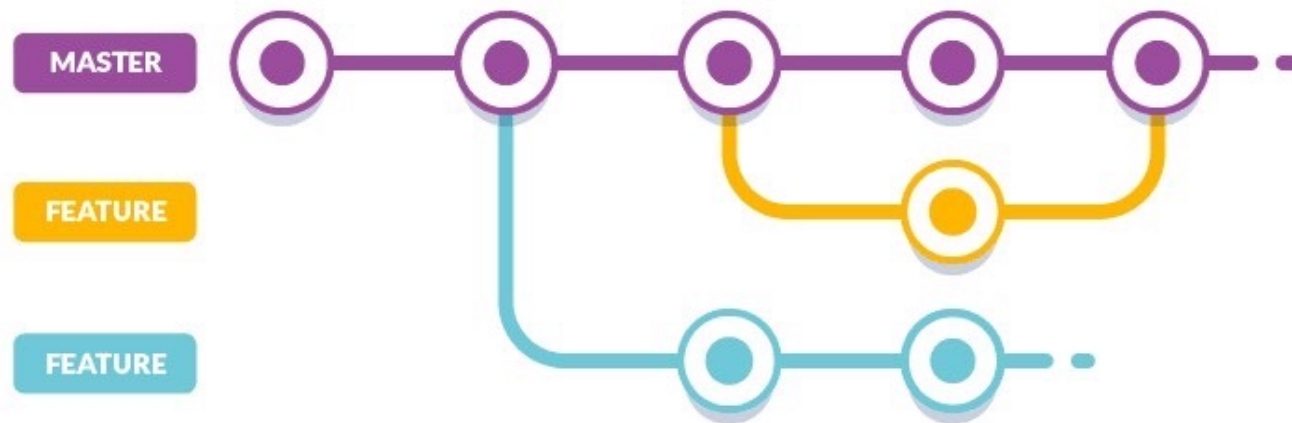
Les workflows

- Un dépôt Git peut vite devenir compliqué à maintenir sans règles imposées
- **Solution : Suivre** un workflow de développement
 - Pour unifier les pratiques au sein d'une même équipe
 - Pour simplifier la gestion du dépôt
 - Pour connaître en temps réel l'état de son dépôt (les fonctionnalités en cours de développement, les branches pouvant être supprimées, ...)
- **Exemples de workflow de développement**
 - **Feature Branch Workflow** : la simplicité, penser petit et synchronisation régulière
 - **GitFlow** : une solution **robuste** s'adaptant à tous les contextes
 - **Fork & Merge** : chacun son dépôt distant, utilisé pour l'**open-source**

Feature Branch Workflow 1/2

- Workflow idéal lorsqu'il n'y a pas besoin de gérer des releases ou des versions
 - Fonctionne surtout en équipe réduite
- **Concept**
 - Une seule branche principale : main
 - Chaque **fonctionnalité** fait l'objet d'une branche **feature** tirée de la main
- **Principes**
 - Tout ce qui est sur main est **déployable**
 - Chaque branche doit avoir un nom significatif
 - Commit et push réguliers
 - **Merge request (Pull request pour Github)** à la fin d'une feature
 - Déployer directement après **merge**

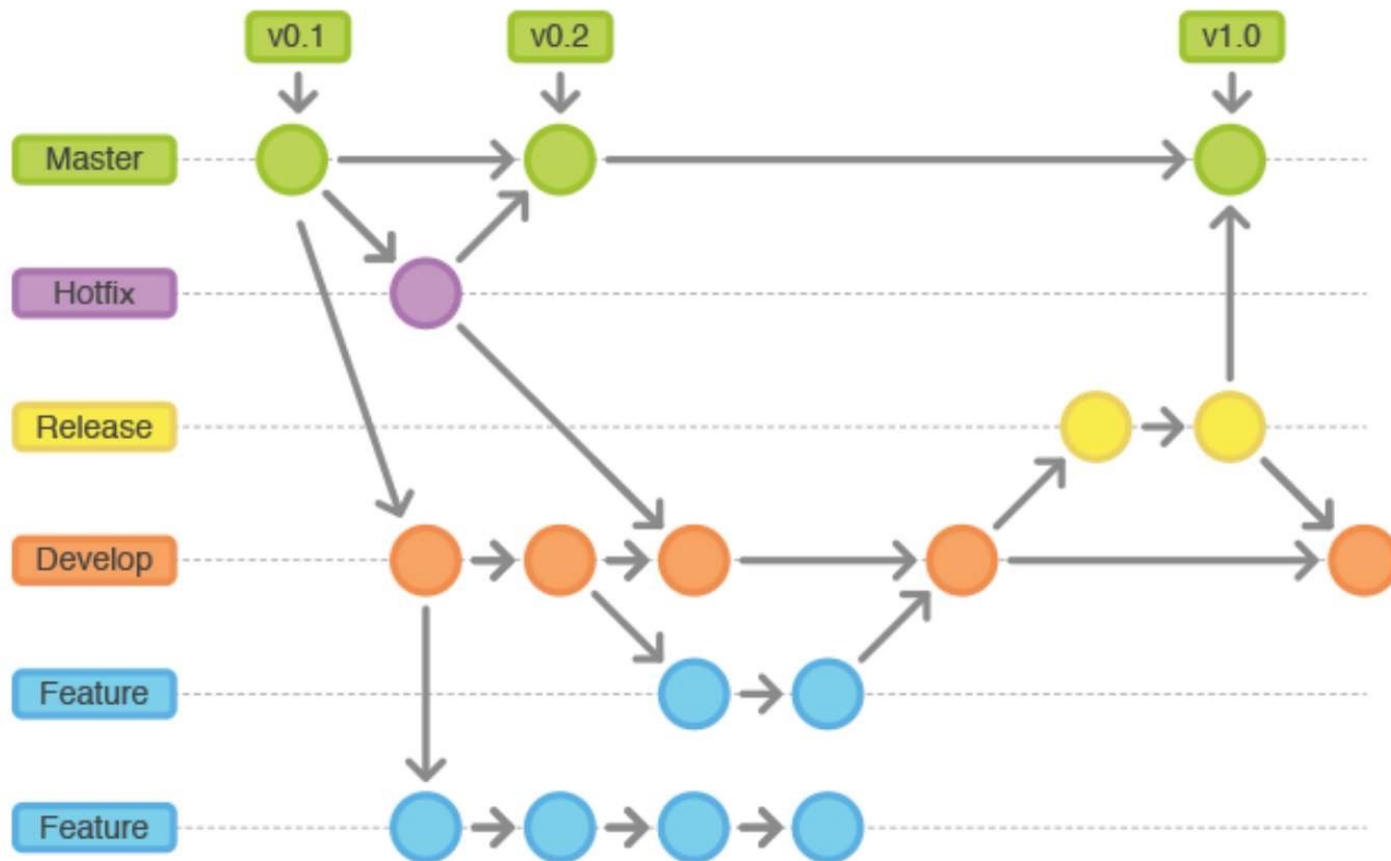
Feature Branch Workflow 2/2



GitFlow 1/2

- Pour les contextes d'équipes grandissantes, de méthode agile et de livraison continue
- **Concept**
 - Une branche pour la production : **main**
 - Une branche pour le développement : **develop**
 - Des branches pour la préparation des mises en production et la gestion des versions : **release**
 - Des branches pour le développement des fonctionnalités : **feature**
 - Des branches pour les corrections en production : **hotfix**
- **Principes**
- Tout ce qui est sur main est **déployable**
- Qualification continue sur la **develop** ou les **features**
- Qualification totale/recette sur les branches de **release**

GitFlow 2/2



Fork & Merge 1/2

- Idéal pour l'open source afin que n'importe qui puisse contribuer à un projet sans le corrompre
 - Plus complexe à mettre en place et à maintenir
- **Concept**
 - Aucun travail sur le dépôt principal
 - Fork du dépôt distant (git clone vers un autre dépôt distant)
 - Travail sur le fork qui peut régulièrement récupérer les modifications du dépôt principal
 - **Merge/Pull request** du fork à la fin avec discussions et merge par les maintainers du dépôt principal
- **Principe**
 - Tout ce qui est sur le dépôt principal est déployable et revu de tous

Fork & Merge 2/2



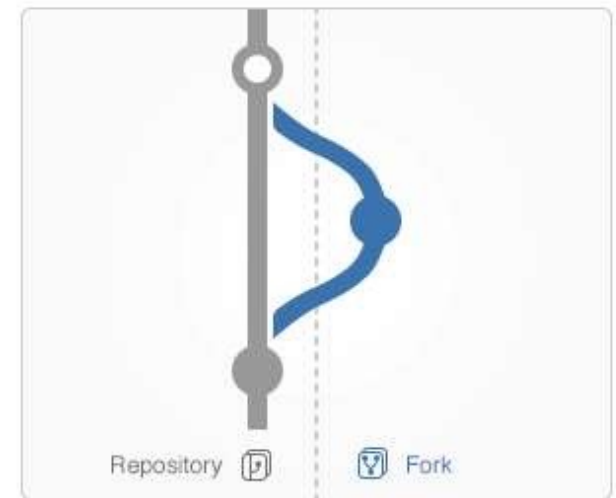
Fork

Develop features on a branch and create a pull request to get changes reviewed.



Discuss

Discuss and approve code changes related to the pull request.



Merge

Merge the branch with the click of a button.

GIT

Atelier : Poney Club

