



M2i Formation

Git

Gérer le versioning

Présentation de Git

Git, un logiciel de gestion de versions

<https://git-scm.com>

- Permet de gérer l'évolution du contenu d'une arborescence via une architecture client/serveur. Le serveur gère le « dépôt » et les clients ont une « copie locale » du dépôt à un instant T
- Sous licence GNU (libre et open-source)

Pourquoi utiliser un logiciel de gestion de version ?

- Suivre les changements d'un projet (*et identifier plus facilement d'éventuels bugs*)
- Gérer les conflits d'édition (*et permettre l'édition collaborative sur des fichiers communs*)
- Réaliser des sauvegardes régulières (*et pouvoir revenir en arrière facilement*)

Comparaison entre Git et Subversion (SVN)

GIT

Logiciel de gestion de versions décentralisé

Chaque « copie locale » est un dépôt à part entière et permet à ce titre de faire des « commits » locaux

- git init / git clone
- git pull
- git add / git commit / git push
- git branch / git merge
- git diff / git log
- git reset
- git checkout
- git status

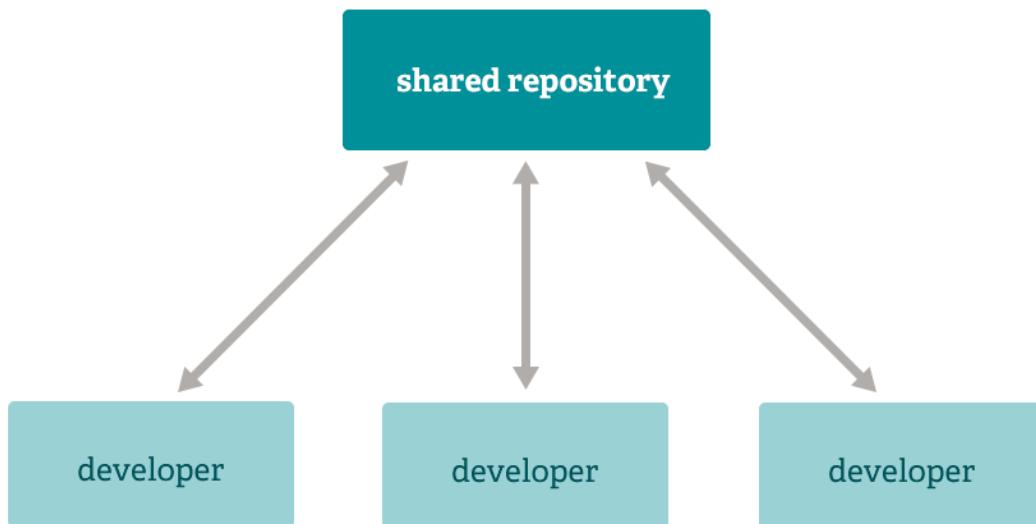
SVN

Logiciel de gestion de versions centralisé

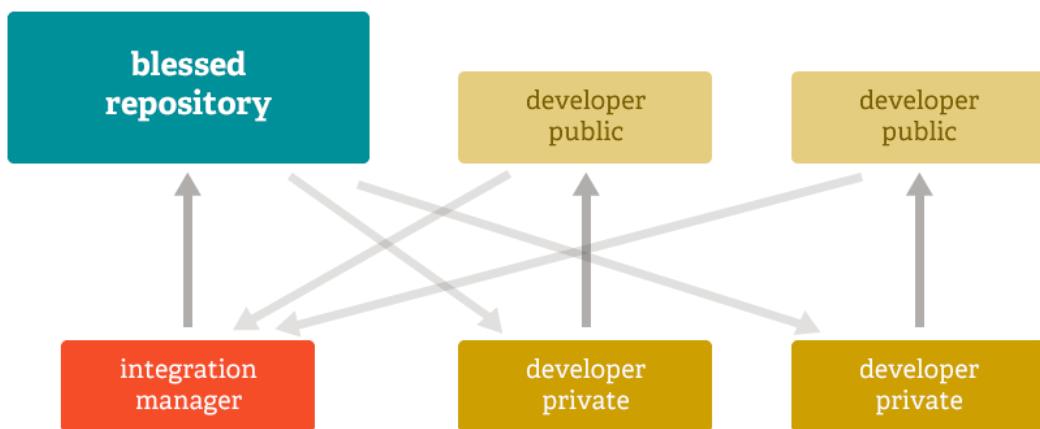
La « copie locale » est uniquement une copie en lecture du dépôt et les commits sont directement envoyés au serveur

- svn checkout
- svn update
- svn add / svn commit
- svn diff / svn log
- svn revert

Les différents workflows



Flux de travail « centralisé »



Flux de travail comprenant un « responsable » ayant autorité

Prise en main / Comprendre les principes de Git

Installation sous windows

<https://git-scm.com>

Configuration de GIT

```
git config --global user.name "votre_pseudo"  
git config --global user.email moi@email.com  
git config --list
```

Commandes pour démarrer

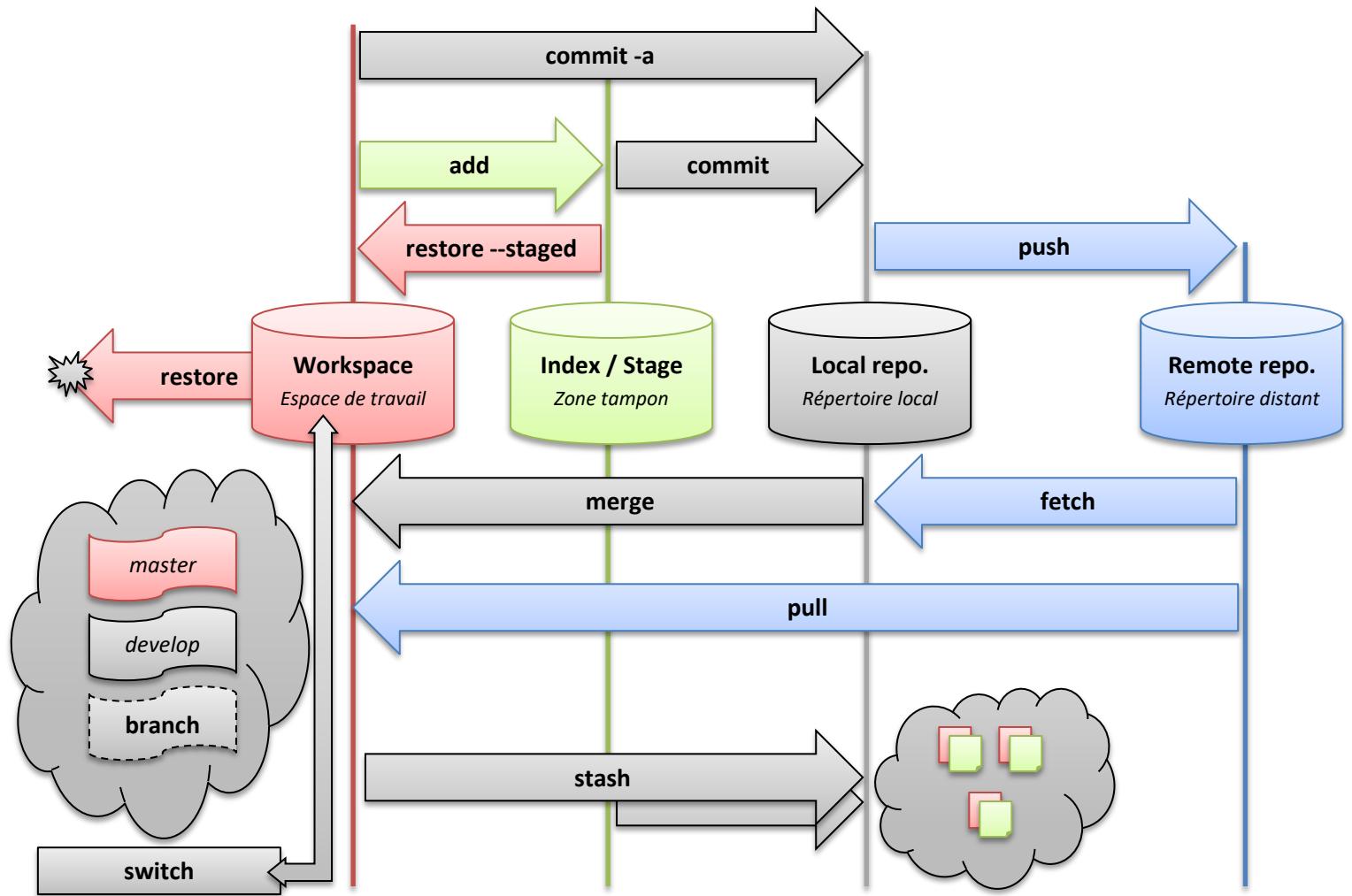
- git init *permet de créer un dépôt dans le répertoire courant*
 - git init *mon-depot* *permet de créer un dépôt dans le répertoire mon-depot/*
 - git clone *permet de cloner un dépôt depuis un serveur distant*
 - git status *permet d'obtenir l'état du dépôt*
 - git remote *permet d'obtenir la liste des dépôts distants*

Commandes pour envoyer des modifications

Commandes pour recevoir des modifications

- `git pull` *permet de récupérer des modifications depuis le serveur et les applique sur le répertoire de travail*
 - `git fetch` *permet de récupérer des modifications depuis le serveur*
 - `git merge` *permet d'appliquer les modifications sur le répertoire de travail*

Schéma des principaux échanges client/serveur



Sur ce schéma, on peut observer 4 éléments différents :

- « **Remote repository** » représente le projet dans son ensemble (commits, branches, etc.) sur le serveur. C'est la version « commune » du projet.
- « **Local repository** » représente le projet dans son ensemble (commits, branches, etc.) sur le poste local du développeur. Cette version peut être « en avance » ou « en retard » par rapport à la version « remote » .
- « **Index/Stage** » représente un ensemble de changements en attente d'être « compactés » au sein d'un « commit ».
- « **Workspace** » représente l'espace de travail du développeur.

Travailler en équipe au jour le jour

Voir les différences en local

- Lorsque l'on modifie plusieurs fichiers, il peut être utile de réafficher les modifications effectuées
 - git diff *Affiche les modifications des fichiers modifiés, non indexés*
 - git diff --cached *Affiche les modifications des fichiers modifiés ET indexés*
 - git diff HEAD *Affiche les modifications par rapport au dépôt local*

Voir l'historique des changements

- Après de nombreux « commits », il peut être intéressant d'afficher l'historique des modifications effectuées sur le dépôt
 - git log *Affiche les derniers commits effectués sur le dépôt*
 - git log -p *Affiche le détail des derniers commits effectués sur le dépôt*
 - git show hash *Affiche le détail d'un commit spécifique grâce à son « hash »*

Etiqueter des versions

- De temps en temps, il peut être utile de « taguer » un état du projet (ex: v1, v2, etc.)
 - git tag *Affiche les étiquettes existantes*
 - git tag v1 -m « ... » *Crée une étiquette v1 avec le message indiqué (« ... »)*
 - git show v1 *Affiche le détail de l'étiquette*
 - git push origin v1 *Envoie l'étiquette sur le serveur*
 - git push origin --tags *Envoie toutes les étiquettes sur le serveur*

Gestion des conflits

- Un conflit survient dès lors qu'un même fichier a été modifié par des « commits » différents sur des lignes communes. Selon les modifications, Git pourra corriger de lui-même les conflits ou bien devra vous laisser la main afin de les corriger.

```
F2000@F2000-PC MINGW64 /d/www/formation (master)
$ git merge
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

F2000@F2000-PC MINGW64 /d/www/formation (master|MERGING)
$ cat README.md
<<<<< HEAD
# Projet Git
Ceci est une formation M2i.
=====
# Foobar
Foobar is a Python library for dealing with word pluralization.
>>>>> refs/remotes/origin/master
```

Actions nécessaires :

- Modifier le fichier en conflit (README.md) afin de définir sa « version finale »
- Ajouter le fichier en conflit dans l'index (`git add README.md`)
- « Commiter » le conflit corrigé (`git commit`)

Créer et appliquer des patchs

- Un « patch » est un fichier texte qui contient une succession d'instructions/modifications à appliquer.

Méthode 1 : utiliser « git diff » et « git apply »

Comme la sortie de « `git diff` » correspond à la liste des modifications appliquées (ligne par ligne) on peut facilement en faire un patch : « `git diff > hotfix.patch` ».

Un patch réalisé de cette manière peut être appliqué via la commande « `git apply hotfix.patch` ».

Méthode 2 : utiliser « git format-patch » et « git am »

Pour utiliser « `git format-patch` », il faut créer une branche sur laquelle on va réaliser les modifications voulues (`git commit`, etc.). Puis lancer la commande « `git format-patch base_branch` » où `base_branch` est la branche de référence.

Un patch réalisé de cette manière peut être appliqué via la commande « `git am mon_patch.patch` ».

Annuler des actions

Modifier le dernier commit non propagé

La commande « **git commit --amend** » permet de modifier un commit local (sur le « local repository »).

Annuler le dernier commit non propagé

La commande « **git reset HEAD~n** » permet d'annuler N commits locaux et remet les modifications dans le « workspace ». L'option « **--hard** » efface définitivement les modifications.

Désindexer un fichier

La commande « **git reset HEAD [...]** » ou « **git restore --staged [...]** » permet de désindexer tout l'index courant (ou un fichier spécifié).

Réinitialiser un fichier modifié

La commande « **git checkout** » ou « **git restore** » permet de réinitialiser toutes les modifications locales d'un fichier.

Annuler le dernier commit propagé sur le serveur

Mauvaise méthode : utiliser « **git reset --hard HEAD~n** » pour revenir en arrière et tenter de « push ». Git rejettéra le « push » à juste titre car cela réécrira l'historique du serveur et quiconque aurait récupérer les commits « effacés » se retrouverai avec des références obsolètes.

- On pourra néanmoins utiliser « **git push --force** » pour forcer l'envoi sur le serveur. Mais à n'utiliser que si l'on est sûr que personne n'a utilisé ces commits

Bonne méthode : utiliser « **git revert HEAD~n..HEAD** » qui va créer des commits « inverses » à chacun des commits indiqués. On peut également annuler un commit spécifique via son hash (voir « **git log** ») grâce à « **git revert [HASH]** ».

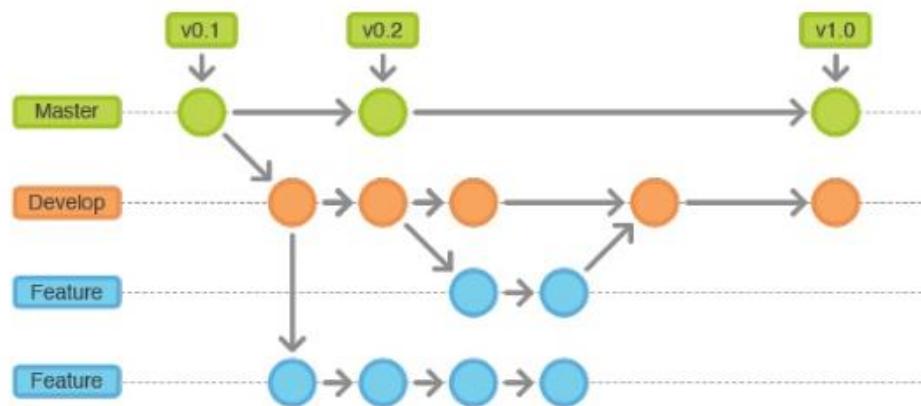
Gestion des branches

Les branches

<https://git-scm.com/.../Les-branches-avec-Git>

- Permet de créer des sous-espaces de travail et de pouvoir facilement travailler sur de nouvelles fonctionnalités

Exemple d'utilisation fréquente



Sur ce schéma, on peut observer 4 branches différentes :

- « **Master** » représente généralement la branche de production et n'est jamais modifiée directement ». Elle sert de référentiel pour savoir ce qui est en production à un instant T.
 - « **Develop** » représente généralement la branche de développement et n'est jamais modifiée directement ». Elle représente ce qui passera en production lors de la prochaine mise à jour.
 - « **Feature** » représente les différentes évolutions du projet qui seront intégrées à la branche « *develop* » lorsqu'elles seront terminées et testées.

Commandes pour gérer les branches

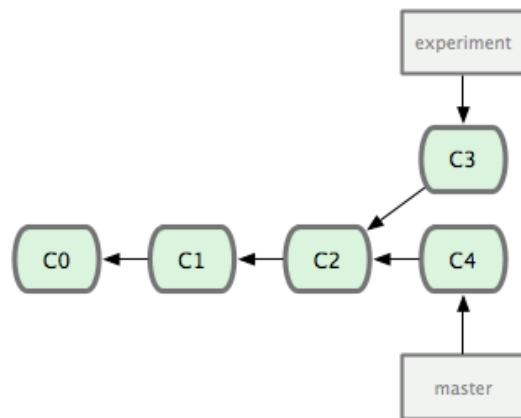
- `git branch` *permet d'afficher les branches existantes*
 - `git branch f01` *permet de créer la branche « f01 »*
 - `git switch f01` *permet de basculer sur la branche « f01 » (eq. git checkout f01)*

 - `git branch -d f01` *permet d'effacer localement la branche « f01 »*

 - `git push origin f01` *permet d'envoyer la branche « f01 » sur le dépôt distant*

Imaginons une branche « experiment » créée à partir du commit C2 depuis « master » (voir schéma à droite) et ayant évolué grâce à au commit C3, pendant que « master » a également évolué de son côté grâce au commit C4.

Si l'on veut récupérer les modifications de la branche « experiment » dans master (donc le commit C3), deux solutions s'offrent à nous :

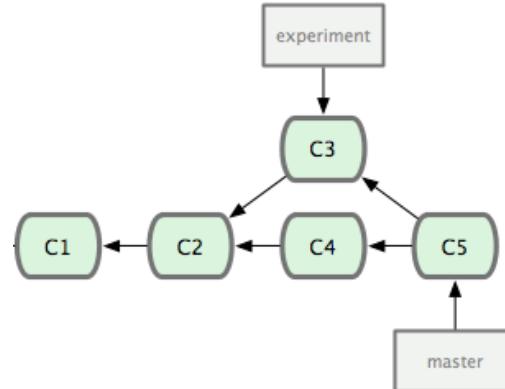


Fusionner une branche via « git merge »

Première solution : se positionner sur la branche « master » et récupérer les commits provenant de la branche « experiment » via la commande :

« git merge experiment »

Cela va créer un « commit de fusion » (C5) pointant sur C4 et C3. Ces 3 commits apparaîtront dans l'historique (« git log »).

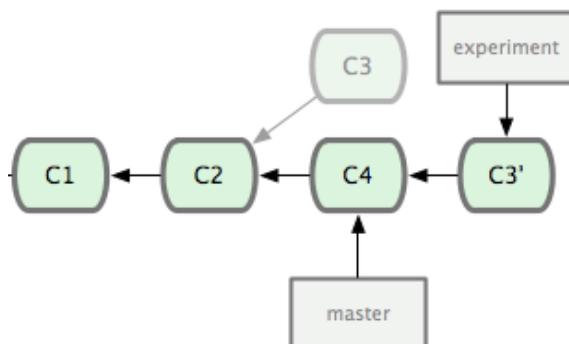


Fusionner une branche via « git rebase »

Seconde solution : se positionner sur la branche « experiment » et « rebaser » par rapport à la branche « master » via la commande :

« git rebase master »

Cela va réappliquer le commit C3 après les modifications provenant de la branche « master » (C4) via un commit équivalent (noté C3').



On pourra ensuite se positionner sur la branche « master » et fusionner la branche via la commande « git merge experiment », ce coup-ci sans commit de fusion. Seuls les commits C4 et C3' apparaîtront dans l'historique.

Note : le résultat final (C5 et C3') est strictement équivalent au niveau de la fusion. Seul l'historique diffère entre les deux méthodes.

Compléments

Github / Les « pull requests »

- Permet de rendre un « merge » collaboratif via des outils intégrés
 - **Espace de discussion** permettant d'évoquer des problématiques particulières sur la résolution d'un problème
 - **Espace de relecture** permettant de demander des changements spécifiques sur une section de code (algorithme à améliorer, conventions à respecter, etc.)
 - **Intégration de hooks** possible via des « web-hooks » (tests unitaires, etc.)

Les relectures

Lors d'une relecture, il conviendra d'identifier les changements à effectuer en isolant les sections de code à corriger. Un commentaire peut en effet être attaché à une ligne de code particulière ou à une section de code.

Une fois la relecture terminée, vous devez indiquer l'impact de votre relecture :

- « **Comment** » *Relecture purement informative sans acceptation/refus explicite*
- « **Approve** » *Relecture approuvant la PR*
- « **Request changes** » *Relecture demandant des correctifs avant le « merge »*

Les **hooks**, tout comme les relectures, permettent d'autoriser ou non le « merge » de la PR.

Les correctifs

En cas de demande de changements (via la relecture d'un collègue ou l'échec des tests unitaires par exemple), les correctifs devront être effectués via de nouveaux commits sur la branche concernée. Tant que la pull-request (PR) est ouverte, les nouveaux commits sont automatiquement visibles sur la PR concernée.

Une nouvelle relecture sera alors nécessaire afin de valider les correctifs effectués.

Lors de l'acceptation du « merge », différents modes de fusion sont proposés par Github :

- « **Create a merge commit** » *Implique un commit de fusion en plus des commits indiqués*
- « **Squash and merge** » *Implique la fusion des commits de la PR en un seul commit*
- « **Rebase and merge** » *Implique le « rebasage » des commits de la PR avec la fusion*

Ignorer des fichiers (le fichier .gitignore)

- Permet d'ignorer certains répertoires et/ou fichiers dans Git. Ces fichiers seront invisibles pour Git et n'apparaitront plus en « untracked files ».
- Exemples de fichiers/dossiers à ignorer :
 - config/parameters.yaml
 - logs/
 - vendors/
 - ...

Ce fichier doit se trouver à la racine pour être lu par Git. Il peut être commité afin d'être partagé aux autres utilisateurs du dépôt.

Le fichier .gitkeep

De temps en temps, il peut être utile de conserver l'arborescence des dossiers mais sans les fichiers à l'intérieur. Par exemple, on veut garder le répertoire de logs mais sans les fichiers de logs à proprement parler.

Pour cela, on va créer un fichier « .gitkeep » (nom communément utilisé par la communauté) afin que Git détecte un fichier dans le répertoire et nous propose d'ajouter ce répertoire dans Git (en effet, si le dossier est vide celui-ci est ignoré par Git).

Exemple de fichier .gitignore

```
# Ignore le fichier
config.yaml

# Ignore le répertoire "logs" et son contenu
/logs/*
# Sauf le fichier .gitkeep
# > Attention, cette ligne doit se trouver après la précédente (/logs)
!/logs/.gitkeep
```

Le remisage (git stash)

- Permet de mettre de côté des modifications en cours afin de pouvoir changer de branche plus facilement ou tout simplement nettoyer son espace de travail (« workspace »).

Commandes liées au remisage

- `git stash` *permet de remiser tout le travail en cours*
- `git stash push -m [...]` *permet de remiser tout le travail en cours en nommant la remise*
- `git stash list` *permet d'afficher les différentes informations remisées*

- `git stash show -p` *permet d'afficher le « diff » de la remise la plus récente*
- `git stash apply` *permet de réappliquer la remise la plus récente*
- `git stash stash@{2}` *permet de réappliquer la remise spécifiée*

- `git stash pop` *permet de supprimer (en appliquant) la remise spécifiée*
- `git stash drop` *permet de supprimer (sans appliquer) la remise spécifiée*

- `git stash branch b01` *permet de réappliquer la remise la plus récente au sein d'une nouvelle branche*

L'option « --index » permet d'indiquer à Git que vous souhaitez retrouver l'état de l'index au même état que lors du remisage.

Note

Un travail « remisé » peut être réappliqué sur une branche différente sans problème. En cas de conflit, Git vous permettra de les résoudre.

Récupérer un commit spécifique (git cherry-pick)

- La commande « `git cherry-pick hash` » permet de récupérer et d'appliquer un commit spécifique très facilement.

- Le commit indiqué doit être connu de Git : on ne peut pas récupérer un commit non partagé d'un développeur. Si le développeur ne souhaite pas publier son commit, il faudra passer par un patch (par exemple) pour échanger les modifications concernées.

Réécrire l'historique (git rebase -i)

- La commande « git rebase -i HEAD~N » permet de réécrire l'historique des N-1 derniers commits

Exemple

```
F2000@F2000-PC MINGW64 /d/www/formation (master)
$ git rebase --interactive HEAD~6

pick 31624ff xxxxxxxxxxxxxxxx      # Commit n-5
pick f8a6bf1 xxxxxxxxxxxxxxxx      # Commit n-4
pick 9bee379 xxxxxxxxxxxxxxxx      # Commit n-3
pick 489a458 xxxxxxxxxxxxxxxx      # Commit n-2
pick 748923f xxxxxxxxxxxxxxxx      # Commit n-1
pick 5aac718 xxxxxxxxxxxxxxxx      # Dernier commit

# Rebase 2d83cdb..5aac718 onto 2d83cdb (6 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```

Comme l'indique les commentaires, on peut effectuer plusieurs actions sur chaque commit, dont:

- « pick » utiliser le commit sans changement particulier
- « reword » utiliser le commit mais éditer le message de commit
- « edit » utiliser le commit mais s'arrêter pour permettre de l'amender
- « squash » utiliser le commit mais le fusionner avec le commit précédent
- « drop » retirer le commit

Vous pouvez également réordonner les lignes comme bon vous ensemble. Les lignes sont exécutées du haut vers le bas.

Débogage (git blame / git bisect)

Annoter un fichier

- La commande « git blame mon_fichier » permet d'afficher, pour chaque ligne, par qui et quand cela a été modifié

```
F2000@F2000-PC MINGW64 /d/www/formation (master)
$ git blame README.md
44e12efa (F2000.FR 2019-05-05 21:54:15 +0200 1) # Projet Git
44e12efa (F2000.FR 2019-05-05 21:54:15 +0200 2)
44e12efa (F2000.FR 2019-05-05 21:54:15 +0200 3) Ceci est une formation M2i.
```

Identifier un commit « buggé » par dichotomie

- « git bisect start » démarre la recherche
- « git bisect bad hash » indique que le commit spécifié contient le bug à identifier
- « git bisect good hash » indique que le commit spécifié NE contient PAS le bug à identifier

La recherche par dichotomie démarre ensuite

- « git bisect reset » restaure l'état initial du dépôt

```
F2000@F2000-PC MINGW64 /d/www/formation (master)
$ git bisect start

F2000@F2000-PC MINGW64 /d/www/formation (master|BISECTING)
$ git bisect bad HEAD

F2000@F2000-PC MINGW64 /d/www/formation (master|BISECTING)
$ git bisect good c44a2b0a0e1fa2a66c94ae0369caf1cf13fef9e7
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[44e12efaa8db038571b4f0b7b66103f5344fccd] tr

F2000@F2000-PC MINGW64 /d/www/formation ((44e12ef...)|BISECTING)
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[43d9deb2b1e584ce5ca9bbe8efc4c5a802192379] Update README.md

F2000@F2000-PC MINGW64 /d/www/formation ((43d9deb...)|BISECTING)
$ git bisect good
849d0922bd3a0cc3dfbc4eb4bdf726093de33f3 is the first bad commit
commit 849d0922bd3a0cc3dfbc4eb4bdf726093de33f3
Author: F2000.FR <contact@f2000.fr>
[...]

F2000@F2000-PC MINGW64 /d/www/formation ((43d9deb...)|BISECTING)
$ git bisect reset
Previous HEAD position was 43d9deb... Update README.md
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Les hooks (dossier .git/hooks)

- Permet de lancer des scripts personnalisés à certaines étapes de Git
- Ces scripts doivent se trouver dans le dossier .git/hooks qui contient déjà des exemples

Hooks côté « client »

- « **pre-commit** » : lancé avant le message de commit. Utile pour exécuter des tests ou vérifier des conventions de code.
- « **prepare-commit-msg** » : lancé après le message de commit par défaut. Permet de personnaliser le message de commit.
- « **commit-msg** » : lancé après que l'utilisateur ai saisi son message. Permet de valider le message de commit.
- « **post-commit** » : lancé après le commit. Permet d'effectuer des notifications.
- « **pre-rebase** » : lancé avant un rebase. Permet d'empêcher un rebase selon des conditions.

Et aussi : « pre-push », « post-rewrite », « post-merge », « post-checkout ».

Hooks côté « serveur »

- « **pre-receive** » : lancé avant d'accepter un « push ». Permet de valider ou non le « push ».
- « **post-receive** » : lancé après l'acceptation d'un « push ». Permet d'effectuer des notifications.