Formation Git

Qu’est-ce que le contrôle de version ?

Le contrôle de version, également appelé contrôle de source, désigne la pratique consistant à suivre et à gérer les changements apportés au code d’un logiciel.

Avant Git, nous avons connu d’autres systèmes de gestion de version : CVS, Subversion (SVN), Mercurial.

Avantages des systèmes de contrôle de version

- Historique complet des changements à long terme de chaque fichier
- Branching et merges, travail en parallèle et indépendant
- Traçabilité, pouvoir retracer les changements (qui ? pourquoi ? quand ?)

Gestion du code source Git

Créé en 2005 par Linux Torvalds (créateur du noyau Linux). https://git-scm.com

Le principale objet de l’utilisation de Git est de ne jamais perdre un changement commité.

Mais Git utilise diverses méthodes pour enregistrer les changements, il est donc possible de :

- Modifier des changements (amender un commit notamment)
- Réécrire l’historique en se dégageant toute responsabilité en cas de perte de contenu

Pourquoi Git ?

C’est avant tout un système de gestion decentralisé, chaque dépot local ou distant comprend l’ensemble de l’historique des changements.

Exemple : la création d’unee branche locale n’est pas visible sur les autres copies du dépôt.

Performance

Les changements sont avant tout locaux, cela veut dire que la gestion des branches et des commits se fait localement donc sans accès réseau.

Sécurité

Chaque changement (commit, version, tag, branche, ...) est tracé et sécurisé par un alogithme de hachage (SHA1), ce qui apporte une protection sauf à réécrire tout l'historique avec la complexité que cela demande.

Flexibilité

Le travail en mode decentralisé permet de découper les projets selon divers mode de gestion de branches (workflows) et d'isoler les responsabilités.

Dépôt GIT ?

Un dépôt Git est un entrepôt virtuel de votre projet. Il vous permet d'enregistrer les versions de votre code et d'y accéder au besoin.

Initialisation d'un dépôt GIT

Pour créer un dépôt GIT utiliser la commande :

```
# Initialise un dépôt GIT dans le répertoire courant (vide ou contenant des fichiers
existants)
# Par défaut la branche master est créée
# Voir git init --initial-branch=<branch-name> | -b <branch-name> | configuration
init.defaultBranch
git init -b main
```

Exécuter une deuxième fois la commande n'écrasera rien.

Cloner un dépôt GIT

Votre projet est déjà configuré dans un dépôt centralisé, la commande clone est la plus courante pour obtenir un clone de développement local.

```
# Clone le dépôt distant dans le répertoire courant
git clone <repo url>

# par exemple
git clone ssh://git@gitlab.lan.bdx.sqli.com:10022/Formation/data/sql-et-plsql-
debutant.git
git clone https://github.com/ghostunnel/ghostunnel.git
```

Enregistrer les changements dans le dépôt GIT

Une fois le dépôt créer ou cloner, vous pouvez faire créer un fichier ou faire des modifications puis les enregistrer.

```
# créer un fichier
echo "Ceci est un nouveau fichier" > nouveau-fichier.txt

# Ajouter ce fichier à la zone de staging
# voir git add --all
git add nouveau-fichier.txt

# Créer un commit avec les éléments dans la zone de staging
git commit -m "Ajout du nouveau fichier au repo"
```

Collaboration avec le dépôt GIT

Une fois votre commit effectué, seul votre copie de travail locale est mise à jour. Votre copie de travail locale est différente de la copie de travail distante. C'est la notion de distribution de dépôt que l'on aborde ici.

Le but est donc de synchroniser notre copie de travail locale avec le dépôt distant.

```
# Synchronisation avec le dépot distant valable si la configuration remote est
configurée
git push
```

Configuration et Installation

Si vous avez créer un dépôt local (avec git init), vous devez configurer l'URL du dépôt distant.

```
git remote add <remote_name> <remote_repo_url>

# pour notre gitlab ce serait par exemple
git remote add origin https://gitlab.bordeaux.sqli.com/rbaron/formation-git.git

# Vous pouvez visualisez votre configuration comme cela (dans le répertoire de votre
projet)
git config -l
```

Synchronisation avec le dépôt distant

Cette commande mappe le répertoire distant qui se trouve dans <url-distant> vers une réf de votre dépôt local disponible sous <branch-name>

```
git push -u <remote_name> <local_branch_name>

# Par exemple
git push -u origin main
```

Votre configuration GIT 1/2

Votre configuration :

```
# Globale
cat $HOME/.gitconfig

# Locale
cat ./.git/config
```

Votre configuration GIT 2/2

Définissez le nom et l'email de l'auteur à utiliser pour tous les commits dans le dépôt actuel.

```
git config --global user.name <name>
git config --global user.email <name>
```

Définissez un alias de commande GIT.

```
git config --global alias.ci commit
```

Personnaliser votre terminal

Personnaliser votre prompt pour vous donner quelques informations vous permettant de vous situer dans votre dépôt local/distant et son statut.

Le script https://github.com/magicmonty/bash-git-prompt vous apporte de nombreuses informations.

```
# Editer votre fichier de configuration ~/.bashrc
# Ajouter la ligne ci-dessous après l'inclusion du script bash-git-prompt.sh
GIT_PROMPT_SHOW_UPSTREAM=1

# Observer la différence
```

Collaborer

git init

Création à partir de template (.git comprenant des hooks, configuration, ...). Ceci ne sert à créer un template de projet de code, mais de configuration GIT.

```
git init --template=<template-name>
```

Création d'un dépôt brut, pour l'utilisation de dépôt centralisé

```
git init --bare
```

Clonage d'un dépôt local ou distant

```
#Clonage dans un dossier spécifique
git clone <repo> <directory>

# Clonage d'un tag spécifique
git clone --branch <tag> <repo>
```

Clonage d'un dépôt brut

```
# Export du dépôt
git clone --bare https://gitlab.bordeaux.sqli.com/rbaron/formation-git.git

# Import du dépôt
cd formation-git.git
git push --mirror https://gitlab.bordeaux.sqli.com/sdief/formation-git.git
```

La zone de staging

La commande **git add** ajoute un changement dans le répertoire de travail à la zone de staging. Elle informe Git que vous voulez inclure les mises à jour dans un fichier particulier du commit suivant.

Les commandes **git add** et **git commit** composent le workflow Git de base.

Le développement d'un projet repose sur le processus de base qui s'articule autour de trois étapes : l'édition, le staging et les commits.

```
# Place dans la zone de staging tous les fichiers modifiés
git add --all
```

Visualiser l'état de vos changements

La commande **git status** affiche l'état du répertoire de travail et de la zone de staging. La sortie de l'état n'affichera pas les informations sur l'historique du projet commité. Pour cela, vous devez utiliser **git log**.

```
# Affiche les fichiers modifiés, les fichiers dans la zone de staging avec leur état
de modification (ajout/modification/suppression)
git status
```

Valider vos changements

Les commits constituent les piliers d'une chronologie de projet Git. Les commits peuvent être considérés comme des instantanés ou des étapes importantes dans la chronologie d'un projet Git.

```
# Commit l'ensemble des fichiers stagés avec un prompt pour vous demander de saisir un
commentaire
git commit
# Commit l'ensemble des fichiers stagés et modifiés ayant déjà été stagés dans
l'historique puis demande un commentaire
git commit -a
# Commit l'ensemble des fichiers stagés avec le commentaire passé en argument
git commit -m "..."
```

Modifier un changement

Les commits, une fois validés, peuvent être modifiés avant d'être envoyés. Pour cela on utilise l'option **--amend**.

```
# Par exemple, vous modifiés des fichiers
git add --all
# Vous ajoutez la zone de staging au commit précédent sans modifier le commentaire
git commit --amend --no-edit
```

Synchroniser vos changements avec le dépôt distant

git push <remote> <branch-name>

Visualisez les changements - git diff

Le diff prend deux ensembles de données et génère une sortie révélant les changements entre eux. Le "diff" peut être appliqué sur le répertoire de travail, entre deux commits, deux branches.

```
echo "ceci est un text" > text.txt
git add text.txt
git commit -a -m "ajout du fichier text.txt"
echo "ceci est un text modifié" > text.txt
git diff
```

Visualiser les différences

```
# Comparaison de deux branches main et new_branch
git diff main..new_branch

# Comparaison du fichier test.txt de deux branches main et new_branch
git diff main new_branch ./test.txt

# Visualiser les commits
git log
```

Faire un stash

GIT permet d'utiliser une zone de remise pour mettre de côté vos changements. Cela est utile lorsque votre copie de travail n'est pas stable et que vous ne souhaitez pas commiter l'ensemble de vos modifications mais que vous devez basculer sur une autre branche. Vous pouvez alors changer de branche, faire de nouveaux commits.

```
# Remiser votre copie de travail, revient à faire un git reset --hard mais en
sauvegardant vos changements pour les retrouver plus tard
git stash
# ou avec un commentaire pour le retrouver
git stash save "..."
# Lister vos stashs en cours
git stash list
stash@{0}: WIP on main: 5002d47 our new homepage
stash@{1}: WIP on main: 5002d47 our new homepage
stash@{2}: WIP on main: 5002d47 our new homepage
```

Appliquer votre stash

Une fois revenu sur votre branche cible, vous pouvez appliquer votre stash

```
# Appliquer votre stash et le supprimer de la remise
git stash pop
# Appliquer votre stash et le garder, permet d'appliquer le stash sur plusieurs
branches
git stash apply
# Nettoyer le stash
git stash clear
```

Visualiser les stashs

```
# Voir le contenu d'un stash (option -p pour complet)
git stash show
 index.html | 1 +
 style.css | 3 +++
 2 files changed, 4 insertions(+)
```

.gitignore

Git considère chaque fichier de votre copie de travail comme appartenant à l'un des trois types suivants :

- tracké : un fichier qui a été stagé ou commité au préalable
- non tracké : un fichier qui n'a pas été stagé ou commité
- ignoré : un fichier que Git a explicitement reçu pour instruction d'ignorer.

Les fichiers ignorés sont généralement des fichiers compilés, logs, cache, …

Il faut créer un fichier à la racine du projet comprenant les chemins des fichiers à ignorer :

```
# Tous les fichiers contenus dans uns sous arborescence du répertoire logs
**/logs
```

Penser à supprimer le fichier déjà présent dans le dépôt avant de l'ignorer.

Visualiser les changements - git log

La commande **git log** affiche des instantanés commités. Cela permet de lister l'historique du projet. Alors que **git status** vous permet d'inspecter le répertoire de travail et la zone de staging, **git log** fonctionne uniquement sur l'historique commité.

```
git log --oneline

# limit to n lines
git log -n <limit>

# filtrer par auteur pour le fichier .gitignore
git log --author="pattern" -p .gitignore
```

Visualiser les changements - git blame

La commande **git blame** permet d'examiner le contenu d'un fichier ligne par ligne, et de savoir à quelle date chaque ligne a été modifiée pour la dernière fois et par qui.

```
# Liste les changements limité aux lignes de 1 à 5
git blame -L 1,5 README.md

# Affiche l'email au lieu du nom de l'auteur
git blame -e README.md
```

Tags - Créer un tag

Les tags sont généralement utilisés pour capturer un point de l'historique utilisé pour une version marquée (c.-à-d., v1.0.1). Un tag est similaire à une branche qui ne change pas.

```
# Créer un tag léger avec la version v1.1 depuis la branche courante
git tag v1.1

# Créer un tag annoté avec la version et le commentaire
git tag -a v1.1 -m "la nouvelle version 1.1"
```

Tags - Lister les tags

Pour répertorier les tags stockés dans un dépôt, exécutez la commande suivante :

```
git tag

# Lister les tags avec un pattern 0.1
git ta -l *0.1*
v1.0.1
v1.0.2
v1.0.3
v0.1.1
v0.1.2
```

Tags - Tagger un ancient commit

Par défaut, **git tag** crée un tag sur le commit référencé par HEAD. Sinon, **git tag** peut être transmise en tant que réf à un commit spécifique.

```
# Lister les anciens commit
git log --pretty=oneline

b1c57c0079390c7b4bff320a51d9fc666e09be79 JIRA 825: Upload des feuilles de présence
passe de 1Mo à 3Mo
d8abf94db57bf29524caae1466818ffc1ba02cb2 DEV: Adaptation des paramètres PHP en
fonction des valeurs de la production
c0654b463114361f58b234132a0dae2c463d5fe1 Merge branch 'feature/synchros-vers-
extranet2021' into 'integration'

# Créer un tag depuis un commit
git tag -a v5.3 b1c57c0079390c7b4bff320a51d9fc666e09be79
```

Tags - Tagger un ancient tag

Si vous essayez de créer un tag portant le même identifiant qu'un tag existant, Git renvoie une erreur comme celle-ci :

```
fatal: tag 'v5.3' already exists
```

Vous pouvez forcer le remplacement d'un tag (option -f), à faire avec la plus grande attention ! imaginez les problèmes si vous forcer le remplacement d'un tag déjà présent en environnement de production.

```
git tag -a -f v5.3 b1c57c0079390c7b4bff320a51d9fc666e09be79
```

Tags - Partager un tag

Le partage de tags est similaire au push de branches. Par défaut, **git push** ne fait pas de push de tags. Les tags doivent être explicitement transmis à **git push**.

```
# Push sur le dépôt distant le tag v5.3
git push origin v5.3

# Push sur le dépôt distant les tags v5.2 v5.3
git push --tags origin v5.2 v5.3
```

Tags - Autres commandes

```
# Checkout un tag particulier
git checkout v5.1

# Supprimer le tag v5.3 sur le dépôt courant
git tag -d v5.3
Deleted tag 'v5.3' (was 808b598)

# Supprimer un tag sur le dépôt distant
git push --delete origin v5.1
```

Annuler des changements

Ce scénario doit préférer la commande **git checkout <branch-name>** ou de faire un **git revert** qui ne modifie pas l'historique contrairement à la commande **git reset**.

```
# Annuler les changements, récupérer la branche main
git checkout main
# Annuler un commit, un commit de revert est généré
git revert b1c57c0079390c7b4bff320a51d9fc666e09be79
# HEAD^ autant de ^ que de commits
# HEAD~n avec n le nombre de commits
# Annuler le dernier commit et mets les modifs en staging
git reset --soft HEAD^
# Annuler le dernier commit et mets les modifs hors staging
git reset --mixed HEAD^
# Annuler le dernier commit et supprime les modifications
git reset --hard HEAD^
# Supprimer le fichier toto.txt de la zone de staging
git reset toto.txt
```

Réécrire l'histoire - git commit

La commande **git commit --amend** permet de modifier facilement le commit le plus récent. Elle vous permet de combiner les changements stagés avec l'ancien commit au lieu de créer un commit totalement nouveau.

```
# Edit hello.py and main.py
git add hello.py
git commit
# Realize you forgot to add the changes from main.py
git add main.py
git commit --amend --no-edit
```

Seulement pour vos commits de branche locale, ne pas faire cela pour des commits publics !

Réécrire l'histoire - git rebase

Le rebase est l'un des deux utilitaires Git spécialisé dans l'intégration des changements d'une branche à une autre. L'autre utilitaire d'intégration des changements est **git merge**.

Seulement pour vos commits de branche locale, ne pas faire cela pour des commits publics !

```
# Votre dépôt est créé est attaché à un dépôt distant, vous êtes sur master
git checkout -b new-branch
echo "new file" > new-file.txt
git add new-file.txt
git commit -a -m "add new file"
git push origin new-branch
# Les commits fait sur new-branch sont réappliqués sur master, contrairement à git
merge, il n'y aura pas de commit de merge
git rebase master
```

Réécrire l'histoire - Ecraser les commits

Un "squash" est un regroupement de plusieurs commits. Le but est de les fusionner en un seul pour avoir un historique Git plus propre. Il est notamment conseillé de le faire dans le cadre des Pull Request pour simplifier la relecture des modifications effectuées.

```
echo "1" > 1.txt
git add 1.txt
git commit -a -m "add 1.txt"
echo "2" > 2.txt
git add 2.txt
git commit -a -m "add 2.txt"
git log
git rebase -i e16cb2f661d9cf5611cd413795f3602075f8892b
git log
```

Réécrire l'histoire - Visualiser les changements

Git garde une trace des mises à jour sur la pointe des branches à l'aide d'un mécanisme appelé logs de référence ou « reflog ».

```
15:35 $ git reflog show new-branch
f21a3f8 (HEAD -> new-branch) new-branch@{0}: rebase (finish): refs/heads/new-branch
onto e16cb2f661d9cf5611cd413795f3602075f8892b
77c59e7 new-branch@{1}: commit: add 2.txt
2514aa1 new-branch@{2}: commit: add 1.txt
e16cb2f (origin/new-branch, origin/master, origin/HEAD, master) new-branch@{3}:
commit: add new file
8527a42 new-branch@{4}: branch: Created from HEAD
```

Dépôt distant - git remote

La commande **git remote** vous permet de créer, d'afficher et de supprimer des connexions avec d'autres dépôts.

```
# Ajouter un dépôt distant
git remote add <name> <url>
# Renommber un dépôt distant
git remote rename <old-name> <new-name>
# Lister les dépôts distants référencés
git remote --verbose
# Visualiser les informations de synchos du remote "origin"
git remote show origin
```

Dépôt distant - git fetch

La commande **git fetch** affiche un comportement similaire à **git pull**. Cependant, **git fetch** peut être considérée comme une variante plus sûre et non destructrice.

git fetch est utilisé conjointement aux commandes **git remote**, **git checkout**, **git branch** et **git merge**.

```
#Récupérer tous les changements depuis le remote "origin"
git fetch origin
# Basculer sur votre référence main
git checkout main
# Visualiser les changements
git log origin/main
# Fusionner le remote origin/main sur votre référence locale main
git merge origin/main
```

Dépôt distant - git push

N'utilisez pas le flag --force si vous n'êtes pas absolument sûr de ce que vous faites.

La commande **git push** fait un push de la branche spécifiée vers le dépôt distant, avec tous les

commits et objets internes nécessaires.

```
# Envoi les modifications locales sur master vers le remote origin
git push origin master
# Envoi les modifications de toutes les branches locales vers le remote origin
git push origin --all
# Envoi tous les tags locaux vers le remote origin
git push origin --tags
# Suppression d'un branche localement et sur le remote origin
git branch -D new-branch
Deleted branch new-branch (was f21a3f8).
git push origin :new-branch
To ssh://gitlab.lan.bdx.sqli.com:10022/rbaron/formation-git.git
 - [deleted]         new-branch
```

Dépôt distant - git pull

La commande **git pull** exécute d'abord **git fetch** qui télécharge le contenu du dépôt distant spécifié. Ensuite, une commande **git merge** est exécutée pour faire un merge des références dans un nouveau commit de merge local.

```
# Récupére la copie de la branche, fait le merge et fait un commit de merge
git pull origin
# Récupére la copie de la branche, fait le merge mais sans commit de merge
git pull --no-commit origin
# Récupére la copie de la branche, fait un rebase au lieu du merge, de manière
verbeuse
git pull --rebase --verbose origin
```

Dépôt distant - Les branches

Lorsque vous souhaitez ajouter une nouvelle fonctionnalité ou correction de bug (quelle que soit sa taille), vous créez une branche pour encapsuler vos changements.

```
# Lister les branches de votre dépôt
git branch --list ou git branch
# Permet de créer une branche depuis la branche courante
git branch <branch-name> + git checkout <branch-name> ou git checkout -b <branch-name>
# Supprimer une branche locale et distante
git branch -D <branch-name>
git push origin --delete <branch-name>
# Renommer une branche
git branch -m <branch-name>
# Lister les branches distantes
git branch -a
```

Dépôt distant - Le merge

Dans Git, le merge permet de reconstituer un historique forké. La commande **git merge** vous permet de sélectionner les lignes de développement indépendantes créées avec **git branch** et de les intégrer à une seule branche.

```
# Start a new feature
git checkout -b new-feature main
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Merge in the new-feature branch
git checkout main
git merge new-feature
git branch -d new-feature
```

Dépôt distant - Les conflits de merge

Un conflit survient lorsque deux branches distinctes ont modifié la même ligne dans un fichier, ou lorsqu'un fichier a été supprimé dans une branche, mais modifié dans l'autre.

L'option **git merge --abort** permet de stopper les merge en cours et de revenir à l'état précédent, sinon la commande **git reset** peut être également utile.

Dépôt distant - Merge requests

- Comment utiliser les merge requests dans gitlab.
- Gérer les branches protégées

⇒ Démo sur gitlab

Normalisation - Commentaires des commits

Vos messages de commit doivent être équivalent à : <type_commit>-<numéro_jira>: <message_commit>

Les <type_commit> possibles (écrit en minuscule) :

feat: une feature ou une évolution fix: correction d'un bug refactor: revue d'un bout code ne touchant pas le fonctionnel test: ajout de tests docs: ajout de documentation chore: upgrade de version, modification de pom other: d'autres cas

<message_commit> contient la description de votre modification. L'intitulé doit être clair et concis. Introduction du numéro de Mantis, RTC, Jira ou autre outil ticketing: vous pouvez l'ajouter juste

après le type de commit (feat et fix la plupart du temps).

Exemple: feat-EMMA-1204: mise en place de la signature electronique

Normalisation : noms des branches

Les branches doivent se nommer de la façon suivante :

- préfixé par le type de branche : release/, develop[/],feature/, fix/
- pour les fix et les features : suffixé par le numero de ticket (JIRA ou autre) de la règle de gestion. Un label court peut accompagner le nom s'il est exploitable par un tiers.
- pour les releases : suffixé par la version

Exemple d'un projet avec une version à livrer et un fix en cours d'implémentation :

- Projet
  - master
  - fix/0156
  - release/02.01.03
  - develop/02.01.03
  - feature/0147
  - feature/0148-Valeur des ventes à 0 par défaut

Les workflows

- Workflow "trunk based" ou "main based"
- Workflow par branches de fonctionnalité
- Workflow Gitflow (master|main, feature, develop, + branches de fonctionnalité)

Liens

Sources

- https://www.atlassian.com/fr/git/tutorials
- https://www.ekino.com/articles/comment-squasher-efficacement-ses-commits-avec-git