Git

90 minutes Normal

DataScientest • com

Machine status

Ubuntu Server 18.04 LTS SSD Volume Type 64-bit x86

Online

34.245.135.23

Connect

Reset Stop

Git

Introduction

Git est le système de gestion de versions (*Version Control System* ou VCS) le plus populaire. Il a été développé par Linus Torvalds, créateur du noyau Linux. Il offre un type de gestion de versions très différent, car il s'agit d'un système de gestion de versions distribué. Avec un système de gestion de versions distribué, il n'y a pas une base de code centralisée d'où tirer le code. Vous observerez cela plus en détail avec les dépôts locaux et distants (grâce à GitHub).

D'autres systèmes de gestion de versions, tels que SVN et CVS, reposent sur un système centralisé, ce qui signifie qu'une seule copie maîtresse du logiciel est utilisée. Ces systèmes nécessitent donc une connectivité réseau pour toute action au contraire de git qui va utiliser le réseau seulement lors d'une mise à jour entre le dépôt local et le dépôt distant.

C'est un gestionnaire de version qui est utilisé aussi bien pour des projets publics que des projets privés. Initialement, git permet de versionner des fichiers de code (python, c, c++...), mais il peut également servir pour tout type de fichier (txt, md, csv...).

des participants du projet.

Premier pas

Installation

Git est installé par défaut dans de nombreuses distributions Linux. Il est notamment installé sur votre machine virtuelle DataScientest.

Si vous souhaitez l'installer sur votre ordinateur personnel, vous pouvez suivre les instructions ci-dessous.

Linux (Debian / Ubuntu)

Pour installer git il suffit d'utiliser le gestionnaire de paquet apt:

```
sudo apt install git-all
```

macOS

Il existe plusieurs manières d'installer Git sur macOS.

Vous pouvez choisir le mode d'installation qui vous correspond parmi ceux listés sur le site officiel de git

Windows

Téléchargez l'assistant d'installation depuis le site officiel de git, exécutez le et suivez les instructions.

Initialisation

Nous allons à présent voir les commandes principales de git et nous rentrerons au fil du cours dans les détails de celles-ci.

Il faut avant tout créer un répertoire de travail. Une fois cette étape faite, il suffit de rentrer dans ce répertoire et d'utiliser la commande `git init` pour l'initialisation du dépôt en local.

Show / Hide solution

A la suite de cette commande, git crée un répertoire caché nommé `.git` . Pour rappel vous pouvez lister tous les éléments (y compris les éléments cachés) avec la commande `ls -a`. Vous pouvez rentrer dans le répertoire et faire un `ls`.

Show / Hide solution

ce qu'il y a dans ce répertoire, comprenez qu'il permet le bon fonctionnement de git.

Ajouter des documents

Lorsque l'on crée un fichier, pour que git prenne en compte l'ajout de ce fichier dans le dépôt il faut utiliser la commande `git add nom_du_fichier`

La commande d'ajout peut permettre d'ajouter plusieurs documents en même temps, voire de prendre en compte toutes les modifications qui ont été faites dans le répertoire.
C'est possible en utilisant "*git add .*".

Il faut néanmoins savoir que c'est une pratique à éviter.
En effet le but va être d'historiser au mieux vos ajouts/modifications. Pour qu'en cas de besoin il soit plus simple dans le futur de comprendre ce qui a été fait dans votre répertoire.
Plus tôt vous prendrez de bons réflexes avec git, mieux ce sera pour vos futures missions pour lesquelles vous utiliserez git.

Faire un commit

On a mentionné plus haut le fait d'historiser vos ajouts/modifications de documents. Pour accomplir cette tâche, à la suite d'un ajout (git add), on doit faire un *commit* c'est-à-dire enregistrer l'ajout ou la modification du document dans l'historique des modifications. On peut ajouter un message à nos *commits* pour garder une trace commentée de toutes les modifications. On va donc faire un commit avec un commentaire intéressant et pertinent pour les utilisateurs grâce à la commande git `commit -m "le commentaire"`.

Exemple de commentaires courts et explicites

```
1
2 git commit -m "ajout du fichier nom_
3 git commit -m "modification de la f
4 git commit -m "ajout de documentati
5
```

Supprimer des documents

l'ajout ou la modification d'un document avec `git add`, il faudra également enregistrer la suppression avec un *commit*.

```
git rm  "mon_fichier.txt"
git commit -m "suppression du fichi
```

Connaître le statut de son dépôt local

A tout moment, il est possible de connaître l'état du dépôt: grâce à la commande `git status`, git nous guide également sur les actions que l'on doit faire. Cette commande est extrêmement importante, il est conseillé pour bien prendre en main l'outil de toujours faire un `git status` avant et après chaque commande pour voir l'évolution de nos documents.

Lors de l'exécution de `git status`, si rien est à faire, on devrait avoir l'affichage suivant.

```
On branch master

No commits yet

nothing to commit (create/copy fi
```

On comprend donc les informations qui vont nous être fournies à savoir: Sur quelle branche sommes nous (notion qui sera vue plus tard); si il y a des fichiers à ajouter; si il y a des fichiers à commiter.

Exemple

Nous allons à présent faire un exemple pour mettre en application ces commandes de bases avant de passer aux fonctionnalités avancées de git.

> Créez un fichier doc.txt dans votre répertoire
>
> Ajoutez du texte dans ce fichier (ex:"Mon premier fichier dans mon répertoire git")
>
> Observer le statut du dépôt
>
> Ajoutez doc.txt dans le git
>
> Faite le commit pour l'ajout avec le message "ajout du fichier de documentation"

Show / Hide solution

Configuration

Il est possible d'observer les variables de configuration de git à l'aide de la commande suivante.

```
git config --list
```

C'est ici que les informations concernant le fait qu'il y ait un dépôt distant ou non sont stockées. On peut également rajouter des informations concernant l'utilisateur. Par exemple, lors d'un commit il est important d'avoir les coordonnées (nom, email) de la personne qui a fait les modifications. Vous pouvez changer ces informations grâce à la commande suivante.

```
git config --global user.name "Danie
git config --global user.email "dan
```

Log

Pouvoir observer les logs est essentiel pour un outil de gestion de versions. C'est une fonctionnalité très utilisée, notamment pour savoir quel utilisateur a fait quelle modification (et à quel moment). Cela permet de gérer les versions du répertoire de travail, d'avoir un vrai suivi des branches et de l'évolution des fichiers.

> Faites une modification dans le fichier doc.txt puis faite l'ajout et le commit.

Show / Hide solution

On va donc maintenant aller voir les logs ! Exécuter la commande `git log` et observer la sortie. Tout d'abord on peut remarquer entre nos deux commits que l'auteur a changé car l'on a modifié la configuration. Il existe des options de git log pour avoir un affichage avec une sortie plus élégante ou des informations en plus:

- `--oneline`: Chaque commit est représenté sur une ligne.
- `--graph`: Représente la sortie sous la forme d'un graphe (utile lorsqu'il y a plusieurs branches).
- `--name-status`: Rajoute comme information le type d'action réalisée (A pour un ajout, M pour une modification)

Voici des exemples avec les options combinées.

```
git log
git log --oneline
git log --oneline --graph
git log --oneline --graph --name-st
```

Différence entre deux commit

Il se peut lors du développement d'une application qu'il y ait un incident en production ou dans un fichier que l'on a réalisé. Dans ce genre de situation, on veut rapidement voir les différences entre la version d'avant et d'après afin de voir les modifications, pour cela on utilise la commande `git diff` ou l'on passe en paramètre les numéros des deux commits a inspecter.

Show / Hide solution

> Observer la différence entre vos deux commits. (des `-` et des `+` indique les ajouts et suppressions de ligne)

Git Blame

Une fonctionnalité intéressante de git est de pouvoir afficher les informations d'un document ligne par ligne, ainsi que l'auteur, la date du commit lié à au changement d'état du document. On peut pour cela utiliser `git blame nom_du_fichier`.

Retour en arrière

Après avoir observé les différences entre les commits, on peut en cas d'incident vouloir revenir à une version précédente du code. Pour cela il existe plusieurs possibilités, les plus utilisées étant `git revert` et `git reset`.

git revert

La commande `git revert` permet de créer un nouveau commit qui annule un commit précédent, en appliquant la transformation inverse.

Par exemple, si le commit qui porte le numéro `123` crée un nouveau fichier `fichier_1.txt`, la commit `git revert 123` créera un nouveau commit qui supprime le fichier `fichier_1.txt`.

```
git revert numero_du_dernier_commit
```

On peut aussi revenir en arrière de plusieurs commit en utilisant `git revert` pour annuler une plage de commit :

```
git revert HEAD~2..HEAD # revenir 2
```

Cette commande créera deux nouveaux commits, qui annulent respectivement le dernier et l'avant dernier commit.

git reset

A la différence de `git revert`, qui crée des nouveaux commits pour annuler des commits passés, `git reset` supprime purement et simplement des commits de l'historique.

Ainsi, si l'historique des commits est le suivant : `commit1 -- commit2 -- commit3 -- commit4`, pour revenir à l'état du deuxième commit, on pourra utiliser les commandes suivantes (équivalentes) :

```
git reset commit2 # revenir à l'état
git reset commit2 # supprimer les 2
```

Ces commandes auront pour effet de supprimer les commit `commit3` et `commit4`. L'historique après cette opération sera donc `commit1 -- commit2`.

En comparaison, si l'on avait utilisé la commande `git revert HEAD~2..HEAD`, le nouvel historique serait `commit1 -- commit2 -- commit3 -- commit4 -- commit5 -- commit6`, les commits `commit5` et `commit6` ayant respectivement pour effet d'annuler les `commit4` et `commit3`.

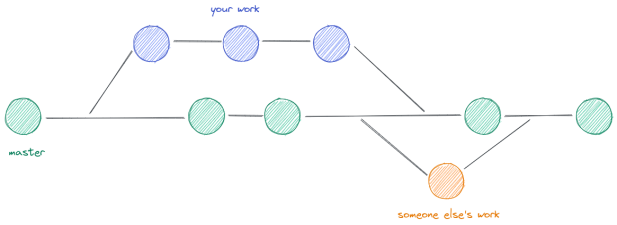
Utiliser `git revert` est considéré comme plus "propre", car on garde une trace de tous les commits, et de leurs suppressions.

> Créez plus de commit et appliquer ces différentes fonctions.

Branche

application, il est toujours souhaitable d'avoir une version stable de celle-ci, mais que se passe-t-il si vous souhaitez développer une nouvelle fonctionnalité ? Car vous ne voulez pas écraser la version stable, ou vous ne voulez pas avoir a faire des `git revert` sans arrêt en cas d'erreur. C'est là que l'intérêt des branches entre en jeu.

Théorie



Par défaut une seule branche existe lors de l'initialisation du dépôt, la branche `master` (celle-ci a été renommée en branche `main` sur GitHub). La bonne pratique est de laisser la version de votre projet la plus stable sur celle-ci.

Lors du développement d'une nouvelle fonctionnalité vous serez amenés à créer une nouvelle branche. Le fait de créer une nouvelle branche va nous permettre de travailler avec les fichiers du répertoire courant dans la nouvelle branche. Ainsi vous pourrez développer votre nouvelle fonctionnalité sur la nouvelle branche, sans oublier les bonnes pratiques apprises jusqu'à présent (git add, status, commit...). Quand vous êtes satisfaits du travail réalisé sur la branche, il est possible de fusionner la nouvelle branche avec la branche `master`.

Pratique

Création de branche

Pour créer une nouvelle branche, on exécute la commande suivante depuis la nouvelle courante:

```
1  git branch nom-branche
2
```

Pour observer les branches qui sont à notre disposition on peut simplement faire un `git branch`.

Changement de branche

Pour se déplacer de branche en branche on exécute la commande suivante:

```
1  git checkout nom-branche
2
```

Enfin pour fusionner deux branches, par exemple si on veut apporter les changements de la branche `branche-source` à la branche `master`, il faudra d'abord se situer dans la branche `master` avec un `git checkout` puis utiliser la commande `git merge` :

```
git checkout master
git merge branche-source

```

Exercice

> Créez une nouvelle branche `modif-doc`
>
> Déplacez vous dans celle-ci
>
> Faites des modifications dans un document (par exemple via `nano`)
>
> Enregistrez ces modifications (add + commit)
>
> Assurez vous que les modifications sont bien prises en comptes dans la branche (status + log)
>
> Retournez dans la branche master et observez que les modifications de sont pas présentes
>
> Fusionnez les deux branches et observez maintenant que les modifications se trouvent bien dans master

Show / Hide solution

En résumé, les branches permettent une meilleure gestion des versions des fichiers. Elles permettent également de travailler plus facilement à plusieurs (chaque personne peut travailler sur une branche distincte).

Gestion des conflits

Dans chaque situation où plusieurs personnes travaillent sur le même fichier, le travail finit par se chevaucher. Il arrive également que deux développeurs modifient la même ligne de code de deux manières différentes. Dans ces cas, Git ne peut pas dire quelle version est correcte, seulement un des développeurs peut décider.

```
Auto-merging [filename1]
CONFLICT (content): Merge conflict :
Automatic merge failed; fix conflict
```

La résolution des conflits de fusion peut prendre une minute ou plusieurs jours (s'il y a beaucoup de fichiers à corriger). Il est recommandé, et c'est une bonne pratique de codage, de synchroniser votre code plusieurs fois par jour en faisant des commits, des pushs, et des merges régulièrement.

Exercice

Nous allons à présent voir un exemple de conflit et sa résolution:

> Créez un nouveau dossier *conflits*
>
> Entrez dans le dossier *conflits*
>
> Initialisez le dépôt git
>
> Créez un fichier *merge.txt* et insérer le texte `première ligne`
>
> Faites l'ajout et le commit

Show / Hide solution

Nous disposons à présent d'un nouveau dépôt avec une branche master et un fichier merge.txt avec du contenu. Nous allons maintenant créer une branche ou nous allons créer un conflit.

> Créez une nouvelle branche `update-merge-file`
>
> Modifiez le contenu du fichier `merge.txt`
>
> Faites l'ajout et le commit

Show / Hide solution

> Retournez maintenant sur la branche *master*
>
> Ajoutez une deuxième ligne dans le fichier *merge.txt*
>
> Faite l'ajout et le commit

Show / Hide solution

> Fusionnez `update-merge-file` dans `master`

Show / Hide solution

Le message suivant devrait apparaître.

> Auto-merging merge.txt CONFLICT (content): Merge conflict in merge.txt Automatic merge failed; fix conflicts and then commit the result.

Le message nous indique donc qu'il y a un conflit avec le fichier *merge.txt*. Un `git status` permet de comprendre que le fichier *merge.txt* a donc été modifié afin de régler le conflit. Nous allons voir à présent le contenu dans le fichier:

Show / Hide solution

Nous pouvons voir certains ajouts dans le fichier:

- `<<<<<<< HEAD`
- `=======`
- `>>>>>>> update-merge-file`

Ces lignes permettent de montrer le conflit et comprendre la différence entre les deux branche: Tout le contenu entre la ligne `<<<<<<< HEAD` et `=======` provient de la branche actuelle vers laquelle pointe la réf HEAD (en l'occurrence `master`). Et donc, tout le contenu entre `=======` et `>>>>>>> update-merge-file` provient de la branche de merge (`update-merge-file`).

Afin de régler le problème, seul le développeur sait quelle version garder. Ainsi, la meilleur méthode pour résoudre celui-ci est de modifier modifier manuellement les fichiers en questions pour garder les versions souhaitées.

> Modifiez le fichier Faites l'ajout et le commit

Show / Hide solution

Tag

Une dernière notion importante à connaître lorsque l'on utilise git est le concept de ***tag***. Les tags sont des références vers un point particulier de l'historique des modifications.

Un tag représente donc un *commit* en particulier, alors qu'une branche représente une succession de *commits*. Les tags sont notamment utilisés pour indiquer un point important dans

Pour créer un tag, il suffit d'exécuter la commande `git tag` suivi du nom du tag :

> Créez un tag 1.0.0

Show / Hide solution

Conclusion

Dans ce module nous avons vu les commandes principales pour pouvoir réaliser un projet avec git. Sachez que git est un outil complexe à prendre en main. C'est qu'en pratiquant que vous allez assimiler au mieux les notions vues ici. Nous ne sommes pas rentrés dans les détails du travail en groupe avec git. Cette notion va être mise en avant dans le module GitHub, qui va permettre d'avoir un dépôt distant où plusieurs développeurs du même projet vont pouvoir travailler de manière collaborative.

Validate