Introduction à Git

Ibrahim ALAME

14/02/2023

1 Qu'est ce qu'un système de contrôle de version ?

1.1 Le problème du contrôle des versions

Imaginez que vous avez un projet et que vous voulez créer une sauvegarde avant de faire des modifications qui sont susceptibles de rendre le projet non fonctionnel. Avant les logiciels de contrôle de version, la seule méthode était de créer un dossier pour votre sauvegarde, par exemple v1.2, puis de copier tous les fichiers dedans.

Mais les difficultés avec ce système sont très nombreuses : comment retrouver les modifications qui ont abouti à telle erreur ? Que faire si on a supprimé par inadvertance un fichier avant de sauvegarder ? Comment collaborer à plusieurs sur un projet ?

1.2 Les systèmes de contrôle de version

Les systèmes de contrôle de version (Version Control Systems (VCS)) permettent à une équipe de gérer les changements apportés à un système de fichiers, qui est le plus souvent du code. Chaque version comporte l'ensemble des modifications depuis la version précédente.

Toutes les modifications des fichiers (ajout, suppression, modifications) sont enregistrés dans une base de données spécifique appelée dépôt (ou repository en anglais). Ce dépôt est géré par un logiciel de contrôle de versions.

Le système de contrôle de version enregistre également des informations sur les versions : qui en est l'auteur, un message entré par l'utilisateur résumant les changements, un horodatage de la sauvegarde etc.

Si une erreur est commise, l'équipe peut facilement revenir à une version antérieure et comparer les versions précédentes pour trouver plus rapidement le problème. La gestion des versions permet également d'empêcher les catastrophes comme par exemple une suppression de code non intentionnelle.

Enfin ces systèmes permettent une meilleure collaboration : plusieurs développeurs peuvent modifier le projet en même temps sans craindre de le casser pour les autres. Il permet également de résoudre facilement les conflits de modifications lorsque plusieurs développeurs ont travaillé sur les mêmes fichiers.

1.3 La gestion de versions centralisée

Dans un système de gestion de versions centralisée, il n'y a qu'un seul dépôt sur un serveur qui fait référence. L'ensemble des développeurs doivent donc s'y connecter pour effectuer des modifica-

tions. Ce système n'est plus utilisé aujourd'hui sauf très rares exceptions.

Les problèmes étaient en effet nombreux : les développeurs ne pouvaient pas modifier en même temps les mêmes fichiers et ne pouvaient pas faire des essais rendant le projet non fonctionnel car cela aurait pour conséquence de le rendre non fonctionnel pour tous les développeurs. Les premiers VCS utilisaient ce système : on peut citer par exemple CVS (Concurrent Versions System créé en 1990) ou Apache Subversion (créé en 2000).

1.4 La gestion de version décentralisée

La gestion de version décentralisée permet à chaque développeur d'avoir en local sa copie du projet ainsi que son logiciel de gestion de versions.

Les avantages sont nombreux,

— le premier étant que les développeurs n'ont pas à être connecté au gestionnaire de versions sur le dépôt en ligne. Ils peuvent ainsi travailler indépendamment de leur côté.

— Le second est qu'il n'y a pas un seul serveur avec tout le code qui crée un seul point de défaillance. Ici chaque participant au projet a une copie entière de tout le code.

— Le troisième avantage est de pouvoir expérimenter seul sur un projet sans risquer de gêner les autres développeurs avec ses essais.

Les gestionnaires de version décentralisés les plus connus sont Git et Mercurial. Nous verrons Git en détails dans la prochaine leçon.

2 Présentation de Git

2.1 Un peu d'histoire

Git a été créé en 2005 par Linus Torvald auquel on doit également le noyau Linux (oui, Linux vient de son prénom Linus).

Le noyau Linux est un programme très bas niveau écrit en C et utilisé par tous les systèmes d'exploitation GNU/Linux (par exemple : Ubuntu, la version la plus utilisée de Debian, Fedora etc), par le système d'exploitation mobile Android, et par de très nombreux autres systèmes (notamment pour les objets embarqués, les appareils éléctroniques -machines à laver, smart TV etc-).

L'équipe de développement du noyau utilisait en fait un logiciel propriétaire pour la gestion des versions, BitKeeper qui avait au début une version gratuite pour les projets libres. Lorsque la licence a été révoquée, Linus a donc créé Git pour continuer à pouvoir développer le noyau avec un système de gestion de version libre de droits.

Git est aujourd'hui le système de gestion de version décentralisée le plus populaire, et de très loin, des dizaines de millions de développeurs l'utilisent Git a donc été prévu dès le départ pour gérer le développement de projets très complexes (noyau Linux) avec des centaines de contributeurs actifs.

Tous les projets majeurs récents utilisent Git pour leur contrôle de versions. Nous pouvons citer par exemple : Linux, Android, tous les frameworks (Angular, Vue.js, React etc), Node.js, Flutter etc.

2.2 Fonctionnement

Git permet de travailler sur un même projet de manière décentralisée. Chaque développeur a sa propre version complète du projet et de l'ensemble des modifications ayant eu lieu depuis l'origine.

Il permet à des centaines de développeurs de travailler sur un même projet en parallèle et de gérer toutes les versions, les fusions de modifications et tout ce qui est nécessaire pour travailler collaborativement sur un grand nombre de fichiers. Nous verrons que Git permet également d'avoir un dépôt distant servant de référence. Nous étudierons son rôle en détails plus tard dans le cours.

En une phrase, Git est un système de fichiers contenant l'histoire d'une collection de fichiers d'un dossier particulier.

2.2.1 L'utilisation de hash

Git permet d'identifier de manière unique les fichiers, les répertoires et les sauvegardes en utilisant leur hash. Le hachage permet de transformer de manière irréversible une valeur (par exemple le contenu d'un fichier) en une chaîne de caractères appelée hash. Cela signifie que depuis le hash il est impossible de retrouver le contenu original et le même contenu donnera toujours le même hash.

Les applications principales du hachage cryptographique sont les suivantes : signature numérique (pour s'assurer de l'intégrité d'un fichier ou d'un message), vérification des mots de passes (pour stocker seulement le hash et pas le mot de passe en clair) et identification de fichiers ou de données. C'est cette dernière utilisation qui nous intéresse ici.

Le hash d'un fichier, ou comme nous le verrons des objets Git en général, permet d'identifier la version unique d'un fichier. Git utilise la fonction de hachage cryptographique SHA-1 (Secure Hash Algorithm) créée par la NSA. Un hash est composé de 160 bits (0 et 1) que l'on représente le plus souvent en utilisant 40 caractères hexadécimaux, voici un exemple :

24b9da6112252987gu493b52f4296cd6d3b00373.

Si un fichier n'est pas modifié, son hash, appelé somme de contrôle dans ce cas d'utilisation, reste inchangé et le fichier n'est pas restocké. Lorsqu'un fichier comporte des modifications, son hash change et une copie est alors crée sur le disque.

En résumé, Git, lors d'une sauvegarde d'une version, ne va enregistrer que les fichiers qui ont changé et va seulement enregistrer une référence pour les fichiers qui n'ont pas changé. Avec ce système il est absolument impossible de modifier un fichier sans que Git le détecte.

Nous verrons que ces hashs sont utilisés pour à peu près tout avec Git et détaillerons très en détail leur utilisation.

2.2.2 Le stockage local

Comme nous l'avons vu, Git étant un système de gestion de versions décentralisé, tout l'historique est disponible localement.

Vous pouvez voir Git comme une base de données locale de paires clés / valeurs. Les clés étant les sommes de contrôles SHA-1 et les valeurs le contenu des objets Git (notamment les fichiers).

La puissance de Git réside dans le fait qu'après une sauvegarde il est quasiment impossible de perdre des données (il faut vraiment faire des commandes spécifiques) et qu'il il est donc toujours possible de revenir en arrière à une version précédente.

3 Environnement

3.1 Installation de Visual Studio Code

Visual Studio Code est un éditeur de code extensible développé par Microsoft. Il a été créé en 2015 et est open source depuis sa création.

Nous allons l'utiliser dans toutes les formations et le recommandons fortement car il est très performant et gratuit. Vous pouvez le télécharger ici.

3.2 Installation des extensions VS Code

Dans la colonne de gauche de l'éditeur VS Code allez sur l'onglet Extensions.

— Recherchez Git lens puis cliquez sur Installer

— Faites de même pour Git Graph.

Nous verrons leur utilisation en détails dans la formation.

3.3 Installation de Git

Nous allons maintenant voir comment installer Git.

Sur Linux

Pour Debian ou Ubuntu, il suffit de faire dans un terminal

```
apt-get install git-all
```

Sur Mac OS

Sur Mac OS, Git est déjà installé, vous pouvez le vérifier en faisant :

```
git --version
```

Si jamais Git était désinstallé ou si vous souhaitez la dernière version allez ici.

Sur Windows

Sur Windows, téléchargez et installer Git en utilisant l'exécutable officiel que vous trouverez ici.

3.4 Utilisation de Git Bash sur Windows

Par défaut, sur Windows, le terminal est Powershell. Dans tous les cours nous utilisons bash, qui est le terminal le plus utilisé et que vous retrouverez sur les serveurs et sur la plupart des environnements de développement.

Sur Windows uniquement, ouvez VS Code.

1. Faites Ctrl + Shift + p ou View puis Command Palette.
2. Entrez select default shell puis faites entrée.
3. Ensuite sélectionnez Git Bash.

Vous pouvez ensuite faire Terminal puis New Terminal et vous aurez un terminal Bash !

4 Présentation de bash et commandes Linux

4.1 Les shells

Un shell est un interpréteur de commandes destinés à accéder à des fonctionnalités du système d'exploitation. Un shell est accessible depuis une console ou un terminal.

Il existe deux grandes catégories de shell : les shell Unix (présents sur MacOS, les systèmes GNU/Linux et de nombreux autres) et les shells Windows (comme par exemple Windows PowerShell sur les versions récentes).

Le terminal par défaut sur la plupart des systèmes Unix est bash. Vous pouvez vérifier le shell courant en tapant dans votre terminal :

```
echo $0
```

Si vous avez Bash, vous aurez –bash.

Sur les versions récentes de MacOS vous aurez le shell zsh.

Dans tous nos cours nous utilisons bash qui est le shell le plus utilisé aujourd'hui par les développeurs. Sur Windows, vous avez installé Git Bash qui est un ensemble de logiciels permettant d'émuler un environnement bash et les principales commandes POSIX.

4.2 Les commandes systèmes

Vous pouvez vous amuser à voir les versions des commandes systèmes et d'où elles proviennent. Sur un système GNU/Linux, par exemple Ubuntu, tapez :

```
mkdir --version
```

Vous obtiendrez :

```
mkdir (GNU coreutils) 8.28
  Copyright (C) 2017 Free Software Foundation, Inc.
```

Vous pouvez voir que vous êtes bien sur un système GNU et que la commande make directories se situe dans la librairie système GNU coreutils.

Sur MacOS, vous pouvez faire :

```
info mkdir
```

Vous verrez :

```
MKDIR - BSD General Commands Manual.
```

Cela signifie en fait que la commande mkdir sur MacOS utilise le système BSD (Berkeley Software Distribution qui est une version libre dérivée également d'Unix). MacOS utilise donc BSD, mais aussi plein d'autres logiciels propriétaires.

Nous ne faisons pas un cours sur les systèmes d'exploitation ou les shells, mais ce qu'il est important de retenir est que lorsque vous ouvrez un terminal quel que soit le système vous avez accès à un shell qui attend que vous tapiez des commandes systèmes. Le shell va ensuite exécuter ces commandes en appelant les librairies systèmes nécessaires.

4.3 La commande ls

ls est une commande signifiant list directory contents. Autrement dit, cette commande permet de lister les contenus d'un dossier.

Comme vu précédemment cette commande, et donc ses options, ne dépendent pas du tout du shell utilisé mais si votre système est de type Unix ou non. D'ailleurs les options varieront légèrement entre les systèmes, sur un système GNU vous pourrez faire :

```
ls --all
```

Mais sur MacOS cela ne fonctionnera pas.

Heureusement, il y a un standard, appelé POSIX (pour Portable Operating System Interface et le X pour Unix) qui standardise un grand nombre d'options sur les commandes systèmes pour les systèmes Unix.

Un ensemble d'options fonctionnera donc sur tous les systèmes respectants POSIX Par exemple, l'option -a permet d'afficher les fichiers et les dossiers cachés (précédés par un .) sur tous les systèmes respectant POSIX.

Vous pouvez donc faire sur Ubuntu, Debian, MacOS etc :

```
ls -a
```

- Une autre option POSIX utile est -t qui permet de trier les dossiers et fichiers du plus récent au plus ancien.
- -l permet d'afficher en plus du nom, le type de fichier, les permissions, les noms du propriétaire et du groupe, la taille en octets et l'horodatage de la dernière modification.
- Le type de fichier est le premier caractère : - pour un fichier ordinaire, d pour un dossier, ou encore l pour un lien symbolique.

4.4 La commande echo

echo permet d'afficher sur la sortie standard ce qui lui est passé en argument suivi par une nouvelle ligne. Par exemple :

```
echo 1
```

Affichera dans le terminal 1 suivi d'une nouvelle ligne.

4.5 La commande rm

La commande rm pour remove permet d'effacer un ou plusieurs fichiers ou dossiers.

Les options POSIX sont les suivantes :

- -r permet de supprimer récursivement les sous-répertoires. Cela signifie qu'elle va supprimer le dossier et tout son contenu :

  ```
  rm -r dossier
  ```

- -f pour force permet de ne pas demander de confirmation et de ne pas afficher de message d'erreur.
- -i permet de demander à l'utilisateur de confirmer l'effacement pour chaque fichier. Il faudra taper y ou Y puis entrée pour confirmer la suppression fichier par fichier.

4.6 La commande mkdir

La commande mkdir pour make directories permet de créer des dossiers, également appelés répertoires.

L'option -p permet de créer les répertoires parents s'ils n'existent pas. Par exemple :

```
mkdir -p a/b/c
```

Va créer les répertoires parents du dossier c : a et b s'ils n'existent pas

4.7 La commande cd

La commande cd pour change directory permet de naviguer dans un terminal entre les dossiers. . signifie dossier courant et ../ remonter d'un cran. Par exemple :

```
cd ../dossier2
```

Signifie remonte dans le dossier parent puis va dans le dossier2 situé dans celui-ci.

5 Configuration initiale de Git

5.1 Créer un dépôt Git localement

Nous avons Git installé, et nous souhaitons maintenant commencer à l'utiliser. Nous allons créer un dépôt local Git dans un dossier vide par exemple test-git :

```
cd test-git && git init
```

La commande git init permet d'initialiser un nouveau dépôt Git. En tapant cette commande dans un dossier, vous aurez le message suivant :

```
Dépôt Git vide initialisé dans /chemin/absolu/vers/le/dossier/caché/git
```

Git va créer un dossier caché dans le dossier dans lequel vous avez initialisé le dépôt. Vous pouvez maintenant voir ce dossier en tapant :

```
ls -a
```

ls, abréviation de list, permet de lister les fichiers et les dossiers dans le répertoire courant. L'option -a permet d'afficher tous les fichiers et dossiers, y compris les fichiers et dossiers cachés.

Dans ce dossier caché nous trouvons les dossiers branches, hooks, info, objects, refs et les fichiers config, description et HEAD.

Nous les verrons également au fur et à mesure, nous avons simplement vu le fichier config qui est le fichier de configuration locale du dépôt.

5.2 Configuration basique de Git

Git possède une commande permettant de le configurer :

```
git config
```

Les niveaux de la configuration Git

Il existe trois manière de configurer Git :

1. sans option : si vous lancez la commande dans un répertoire Git, alors les configurations ne seront valables que pour le dépôt en cours d'utilisation. La configuration se trouvera alors dans le dossier du dépôt dans .git/config.
2. avec l'option –global : si vous ajoutez cette option à la commande, alors la configuration se fera pour votre utilisateur sur votre machine : par exemple Paul ou root. Le fichier de configuration se trouvera alors ici : ~/.gitconfig (ou C :\Documents and Settings\utilisateur ou C :\Users\utilisateur sur Windows).
3. avec l'option –system : si vous utilisez cette option, alors la configuration sera valable pour tous les utilisateurs du système. Le fichier de configuration se trouvera alors ici : /etc/gitconfig. (ou sur Windows dans C :\ProgramData\Git\config).

Git va lire les configuration dans cet ordre système puis globale puis locale au dépôt. Git va cumuler les configurations trouvées, mais si deux propriétés ont deux valeurs différentes, alors Git prendra la plus spécifique. Par exemple, si vous spécifiez une adresse email au niveau global et une au niveau local dans la configuration de Git, il prendra l'adresse email au niveau local lorsque vous utiliserez Git dans le dépôt.

Configuration de l'identité

Nous allons commencer par voir comment configurer son nom et prénom et son adresse email.

Ces informations seront liées à chaque sauvegarde que vous effectuerez avec Git. Par exemple, pour définir ses nom, prénom et adresse email pour l'utilisateur courant, il faut faire :

```
git config --global user.name "Jean Dupont"
git config --global user.email "jean.dupont@protonmail.com"
```

5.3 Lister les configurations

Vous pouvez lister à tout moment la configuration de Git en faisant :

```
git config --list
```

Si vous ne vous trouvez pas dans un dépôt Git, seules les configurations systèmes et globales s'afficheront, par exemple :

```
user.name=Jean Dupont
user.email=jean.dupont@protonmail.com
core.autocrlf=input
```

Si vous vous trouvez dans un dépôt, toutes les configurations système, globale et locale fusionnées suivant la règle spécifié ci-dessus apparaîtront :

```
user.name=Jean Dupont
user.email=jean.dupont@protonmail.com
core.autocrlf=input
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=git@gitlab.com:jdupont/megaprojet.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
branch.develop.remote=origin
branch.develop.merge=refs/heads/develop
gui.wmstate=normal
gui.geometry=1822x800+2605+53 612 212
branch.ng9.remote=origin
branch.ng9.merge=refs/heads/ng9
branch.ng9-2.remote=origin
branch.ng9-2.merge=refs/heads/ng9-2
```

Nous verrons au fur à mesure ce que signifie cette configuration. Mais il faut simplement que vous reteniez le principe des niveaux de configuration Git pour le moment.

A noter qu'il est complètement inutile de s'attarder sur les options de configuration avancées de Git, il en existe plusieurs centaines mais vous n'en aurez jamais besoin.

5.4 Lister la configuration pour une propriété

Vous pouvez également vérifier uniquement un paramètre en tapant dans un terminal git config paramètre, par exemple :

```
git config user.email
```

Qui donnerait dans notre exemple :

```
jean.dupont@protonmail.com
```