Formation Git : Concepts et Architectures

El Hadji Gaye

Auteur	El Hadji Gaye
Pour	Formation
Date	16/09/2024
Objet	Formation Git : Concepts et Architectures

I) Préambule 3
II) La gestion de version 4
1. Définitions 4
2. Les intérêts de la gestion de version 5
3. Histoire de la gestion de version 8
4. Pourquoi choisir Git comme outil gestion de version 12

III) Qu'est-ce Git 14
IV) Architecture générale de Git 15
1. Généralités 15
2. L'utilisation de hash 16
3. Le stockage local 17

V) Qu'est-ce qu'un dépôt Git 18
1. Introduction 18
2. Le contenu du dossier .git 23

I) Préambule

Lorsqu'on parle de système de gestion de versions, l'approche du néophyte revient à le considérer comme un système de sauvegarde incrémentiel : « Grâce à un système de gestion de versions, les développeurs peuvent retrouver toutes leurs anciennes versions. »
Mais Git ne permet pas uniquement cela, il couvre de nombreuses autres possibilités.
Git permet de paralléliser plusieurs versions du même logiciel, par exemple lorsqu'un développeur travaille sur une nouvelle fonctionnalité, mais que celle-ci ne doit pas encore être intégrée au logiciel final.
Git sert également de documentation complète. Chaque nouvelle modification de code est accompagnée d'un message. Au bout de plusieurs années, ces messages peuvent se compter en milliers et devenir des documentations très intéressantes indiquant le contexte dans lequel les modifications ont été effectuées.
Les systèmes de gestion de versions peuvent paraître contraignants au début, mais se révèlent indispensables lorsque le développeur y a pris goût.

II) La gestion de version

1. Définitions

Un système de gestion de versions (VCS en anglais pour Version Control System ou aussi SCM pour Source Content Management), ou encore système de versionning, est un système qui enregistre toutes les modifications apportées à une liste de fichiers. C'est un système qui permet de suivre précisément l'évolution du contenu des fichiers. Ce type de système est très utilisé en développement informatique, mais il n'y est pas limité puisque n'importe quelle activité utilisant des fichiers lisibles peut être suivie par un VCS. Par exemple, les systèmes Wiki (comme Wikipédia) utilisent un VCS. Il est également possible de citer le très connu site d'entraide informatique Stack Overflow qui suit les versions des questions et des réponses.

Le suivi des fichiers est effectué par sauvegarde des modifications. En effet, lorsque vous avez fini de modifier un fichier, vous allez indiquer au VCS que vous avez terminé ce travail et pour quelles raisons vous l'avez fait. Les modifications effectuées sur le fichier seront sauvegardées par le système.

Il ajoutera alors une nouvelle ligne (qu'on appellera révision) dans votre historique. Le terme révision (appelé également commit en anglais) définit un ensemble de modifications qui sont enregistrées par le système.

Les intérêts d'un tel système sont multiples et vont bien au-delà du suivi pur et simple d'un projet.

2. *Les intérêts de la gestion de version*

Une véritable machine à remonter le temps

Imaginez une petite situation habituelle : vous êtes un indépendant et vous commencez votre semaine. Lundi matin à 9h, un de vos clients vous appelle : "Vous deviez modifier mon site pour enregistrer tout ce que faisaient les utilisateurs du site". Vous vous empressez de vous excuser et vous modifiez le code pour ajouter la fonctionnalité le plus rapidement possible. Une fois correctement réveillé - soit une heure et deux cafés plus tard -, vous vous rendez compte que vous n'êtes pas du tout parti dans la bonne direction. Le code que vous avez produit ne servira à rien et complexifie l'application alors qu'il y avait une manière beaucoup plus simple de faire. Votre VCS vous permet de revenir très simplement en arrière comme si ce début de lundi n'avait pas existé. Vous repartez donc avec un répertoire de travail propre.
Mais votre VCS va encore plus loin puisqu'il peut vous permettre de faire un retour en arrière de plusieurs années sans problème pourvu que vous l'ayez utilisé à ce moment-là.
Un système de versionning permet donc de revenir très facilement en arrière et à n'importe quel endroit de l'historique.

Une documentation détaillée et datée

Dans une équipe, lorsqu'on ajoute une fonctionnalité à un logiciel, il arrive régulièrement qu'on soit bloqué par une ou plusieurs lignes qu'on ne comprend pas. Le code est peu documenté et n'est pas du tout clair. Sans versionning, un développeur pourrait passer plusieurs heures à comprendre le code et à le documenter. Avec un VCS, vous pouvez obtenir différentes informations concernant les lignes de code. Les messages que les développeurs indiquent lorsqu'ils effectuent des modifications sont importants pour les prochains développeurs qui travailleront sur le code, c'est la raison pour laquelle ces messages doivent être rédigés avec soin et clarté. Ce message est communiqué par l'auteur du code où il explique les modifications qu'il a effectuées.
Le message permet d'avoir une idée de ce que le développeur a voulu réaliser dans son code. Par exemple, un message tel que "Prise en compte des remises dans le calcul de la TVA des factures" peut aider à comprendre un code.
Si le code reste encore obscur après la lecture du message, il est aussi possible d'aller voir son auteur si celui-ci est toujours dans l'équipe.
Un système de gestion de versions permet donc de garder une documentation à propos de chaque modification effectuée. Cela permet de simplifier la maintenance des applications et l'intégration de nouveaux collaborateurs.

Une pierre de Rosette pour collaborer

Prenons le cas où deux développeurs (Pierre et Paul) d'une équipe travaillent sur le même projet. Ils sont partis de la même version du projet et ont effectué leurs modifications dans leur propre répertoire de travail. Nous avons donc à un instant T deux versions différentes du projet.
Grâce au système de gestion de version, la fusion de ces deux versions sera facile à effectuer, surtout s'ils ont travaillé sur des fichiers différents. La facilité de collaboration entre développeurs grâce au VCS est d'autant plus avérée lorsqu'on utilise des systèmes de gestion de versions décentralisés.

3. Histoire de la gestion de version

Systèmes de gestion de versions locaux

Les systèmes à stockage local sont des systèmes où tous les utilisateurs partagent le même système de fichiers. Le système conserve les fichiers d'origine puis toutes les modifications effectuées par la suite. De cette manière, il est possible de restaurer les fichiers à n'importe quelle date pendant laquelle a été utilisé le VCS. Les systèmes à stockage local sont arrivés en 1972 grâce à SCCS (Source Code Control System) développé par Marc Rochkind travaillant pour les laboratoires Bell.
En 1982, au cours d'un projet universitaire, Walter F. Tichy publie GNU RCS (Revision Control System). Il apportait aussi un gain de performance important comparé à SCCS grâce à une autre manière de stocker les fichiers. En effet, RCS stockait les dernières versions des fichiers et la différence avec l'ancien fichier, ce qui permettait d'avoir la dernière version des fichiers plus rapidement.
Ce type de solution ne permettait pas aux collaborateurs de travailler ensemble s'ils n'utilisaient pas le même système de fichiers. La collaboration n'était pas aisée et ne permettait aucune mobilité.
Avec ces systèmes seuls les fichiers sont versionnés et ils possèdent chacun leur propre historique de version.
Après avoir été très utilisé dans le monde du libre, RCS a peu à peu été remplacé par CVS.

Systèmes de gestion de versions centralisés

Dans ce type de système, toutes les données du suivi de version sont stockées sur un serveur. À partir de ces systèmes, l'administrateur du VCS pouvait restreindre les privilèges de certains utilisateurs. Parmi les logiciels de gestion de version centralisée, on peut retenir CVS (Concurrent Versions System) et Subversion (ou SVN). CVS est apparu en 1990. Il a été le premier à utiliser un système centralisé. Il a été très utilisé dans le monde du libre et a été remplacé par Subversion en 2000. En effet, Subversion est née de la volonté de l'entreprise CollabNet d'améliorer certains aspects de CVS. Les points d'améliorations les plus notables furent :

- Les commits sont devenus atomiques. Un commit correspond à une série de modifications. Cela signifie que l'historique n'était plus autonome sur chaque fichier. En effet, à partir de Subversion les commits ont concerné plusieurs fichiers.
- Les fichiers renommés ou déplacés conservent leur historique. En effet, avec les anciens systèmes un déplacement ou un renommage de fichier revenait à dire "Le fichier n'a plus rien à voir avec ce qu'il était".

Ces systèmes avaient cependant une importante limitation : la dépendance des clients vis-à-vis du serveur. En effet, pendant que le serveur est inaccessible, il est impossible d'enregistrer des modifications ou de récupérer quoi que ce soit. Cette limitation impose aux administrateurs de garantir une haute disponibilité du serveur de versionning pour ne pas immobiliser les développeurs.

Systèmes de gestion de versions décentralisés

Ce type de système est le plus récent. Il répond tout d'abord au principal défaut des systèmes centralisés : la dépendance des clients vis-à-vis du serveur. Concrètement, lorsque nous utilisons un système décentralisé et que nous souhaitons travailler sur un projet contenu sur le serveur, nous allons tout d'abord cloner ce projet sur notre poste. Cette étape de clonage copie intégralement le projet et son historique sur la machine locale. Cela permet au développeur d'être autonome une fois qu'il a cloné le projet. Il n'a pas besoin d'être connecté à un réseau pour enregistrer ses modifications et consulter l'historique. Bien évidemment, il a besoin d'être connecté au serveur pour recevoir les mises à jour du projet ou partager ses modifications.

L'histoire des VCS (ou DVCS pour Decentralized VCS) commença en 1997 lorsque Larry McVoy (CEO de BitMover) amorça le développement de son système de gestion de versions BitKeeper. BitKeeper est le premier système distribué et son histoire est intimement liée à celle du projet Linux. Au début de l'année 1999, Larry McVoy propose à Linus Torvalds (créateur du noyau Linux) d'utiliser son gestionnaire de versions BitKeeper pour lui permettre de mieux diriger le nombre grandissant de contributeurs au noyau Linux. En effet, la quantité de contributions au noyau devient si importante que le projet prend du retard. Ce retard est en partie dû au fait que la gestion des versions de Linux était basée sur des fichiers de patchs générés par Linus lui-même.
Toujours en 1999, lors d'une interview pour Linux Weekly News, Alan Cox (contributeur du noyau Linux) rappelle que Linus Torvalds est seul pour générer les patchs du projet Linux et qu'il est constamment surchargé de travail. Il affirme également que BitKeeper permettrait de mieux gérer le nombre grandissant de contributeurs au noyau Linux.

En janvier 2002, un développeur du noyau, Rob Langley, propose de nommer un lieutenant d'intégration pour soulager le travail de Linus. Cette proposition démontre que la tâche de Linus est trop fastidieuse et ralentit l'évolution du noyau Linux.

En février 2002, Linus Torvalds décide de migrer vers le système BitKeeper pour simplifier la gestion des versions. À ce moment-là, il est possible d'utiliser gratuitement une version limitée de BitKeeper pour les projets libres. Cette décision a été très controversée dans le monde du libre et dans le cercle des contributeurs au noyau. Deux mois après l'adoption de BitKeeper, un développeur du noyau (Daniel Phillips) soumet un patch retirant la documentation de BitKeeper du projet Linux, en précisant que beaucoup de développeurs "ruminent en silence" la décision de Linus.

Une fois que les membres de la communauté Linux se sont habitués à utiliser BitKeeper, le développement du noyau s'est beaucoup accéléré. Les fusions ont été beaucoup plus rapides et le retard s'est résorbé en quelques mois.

En février 2005, soit trois ans après le passage à BitKeeper, Linus informe Larry McVoy sur le fait qu'un développeur du noyau (Andrew Tridgell, qui est également le créateur de Samba) concevait un client libre pour récupérer les métadonnées de BitKeeper. En effet, dans un système de gestion de versions les métadonnées sont très importantes et permettent au système de fonctionner efficacement. Cependant, la version gratuite de BitKeeper ne permettait pas de récupérer ces données ce qui était une source de frustration chez certains développeurs. Pour pallier ce problème, Andrew Tridgell a analysé le fonctionnement de BitKeeper et notamment ses échanges avec le serveur. Il a donc réussi à développer un client qui permettait de s'affranchir de cette limite.

Larry McVoy et Linus ont très mal vu ce nouveau client. Pour Larry McVoy, les développeurs utilisant une version gratuite de BitKeeper doivent "jouer le jeu" et accepter les limitations de cette version. En réponse à ce client libre, BitMover a annoncé qu'il arrêterait de fournir une version gratuite de BitKeeper aux contributeurs. La situation devenait très complexe pour Linus et toute la communauté était sous tension. Il a donc travaillé sur un nouveau système.
Le 20 avril 2005, Linus annonce la sortie d'un nouveau système de gestion de version distribué : Git. Le développement de Git a été très rapide, car la communauté avait besoin d'un nouveau VCS pour le mois de juillet 2005, la licence gratuite de BitKeeper expirant le 1er juillet 2005.
Cette période d'utilisation de BitKeeper par le projet Linux aura tout de même permis au noyau Linux d'accélérer sa croissance et elle aura permis également à BitMover de promouvoir son système de gestion de versions.

Lorsque Linus a commencé à travailler sur Git, il a défini un certain nombre de besoins auxquels le nouveau système devrait répondre :

- Un système libre : après les problèmes internes et externes dus à l'utilisation de BitKeeper, Linus ne voulait plus être dépendant d'un système propriétaire et voulait diminuer les tensions au sein de la communauté du projet Linux.

- Un système performant : parce que le projet Linux nécessitait beaucoup de fusions de branches et de créations de patchs de mise à jour, Linus voulait un système performant afin de ne pas pénaliser le projet à cause de fusions difficiles.

- Un système distribué à l'instar de BitKeeper pour conserver les avantages de ce type de système.

- Un système compatible avec les protocoles existants : la compatibilité d'un nouveau système avec les protocoles HTTP, FTP et SSH accélérerait le développement et la compatibilité du nouveau système. De plus, un serveur émulant le système CVS a été intégré dans Git pour faciliter le travail des développeurs utilisant CVS.

- Un système s'adaptant à des projets importants : Git a été conçu pour pouvoir gérer des projets importants en bénéficiant de très bonnes performances.

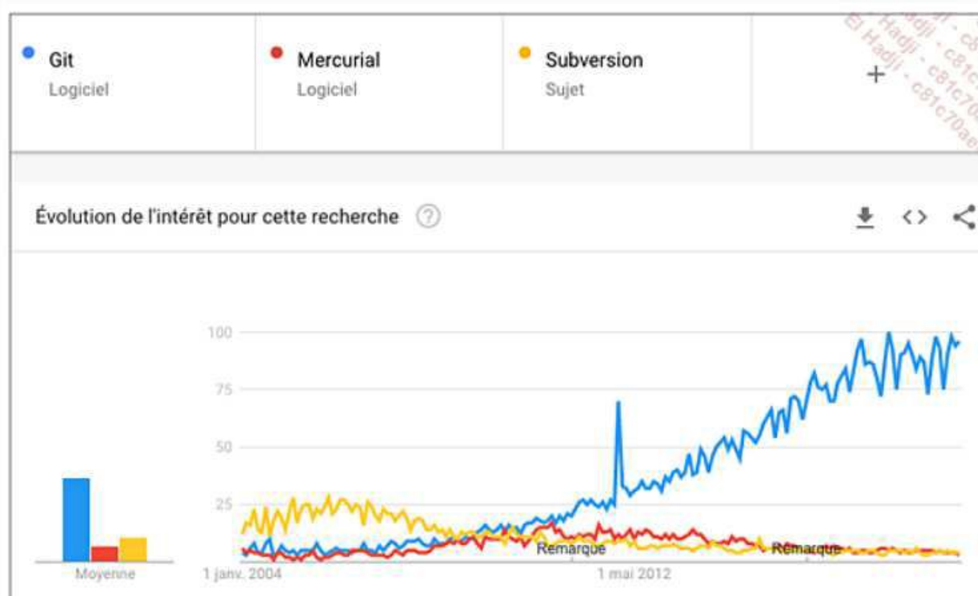
- Un système se basant sur des hashs : un hash (ou une empreinte) est une valeur (souvent représentée sous forme hexadécimale). Elle est calculée à partir d'une autre valeur comme une chaîne de caractères ou un fichier. Deux fichiers ayant un contenu différent auront une empreinte différente (sauf dans certains cas rarissimes). Ce système de fichiers basé sur les hashs permettra à Git de détecter la moindre modification dans un fichier et permettra d'identifier un fichier à partir de son contenu.

À l'origine, le système était conçu comme un système de fichiers sur lequel était placée une surcouche ajoutant les fonctionnalités de suivi de version. C'était cependant un système basé sur le contenu des fichiers, car on définissait un fichier par son contenu grâce au hash généré.
Au début de l'évolution de Git, les nouveaux développements mirent en avant cette surcouche. Jusqu'à aujourd'hui (en 2015), Git a évolué et a continué à gagner en popularité. Cette popularité est également due à un service en ligne né en 2008 : GitHub. C'est une plateforme offrant la possibilité de stocker ses projets sous forme de dépôts distants sur les serveurs de la société. Ce service offre des fonctionnalités pour travailler en équipe et collaborer efficacement entre développeurs.

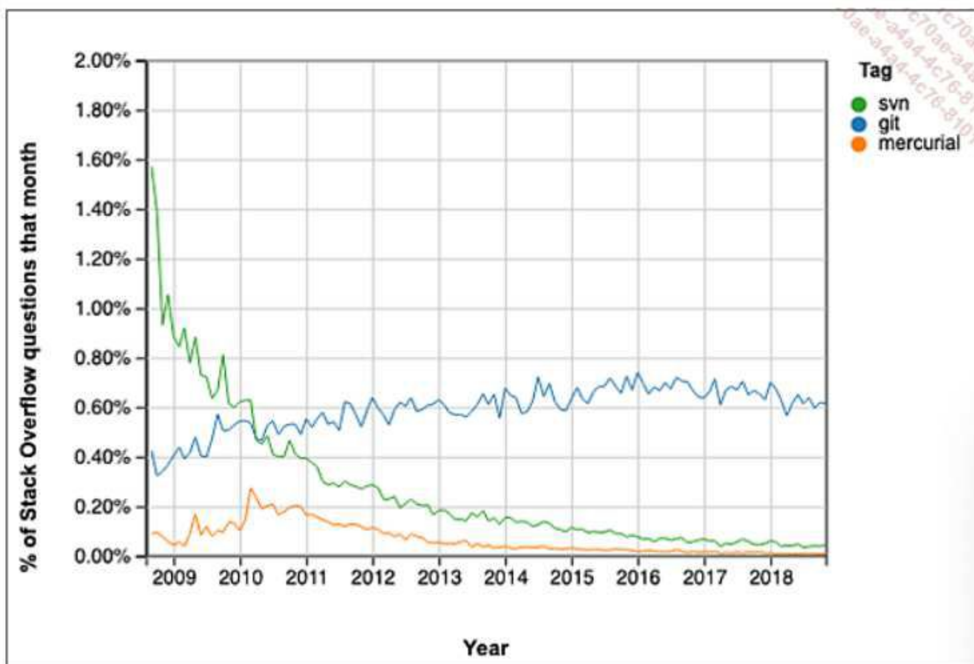
L’utilisation de GitHub pour les projets libres (et sans restriction d’accès) est gratuite. En revanche, lorsqu’une entreprise veut utiliser GitHub pour des projets privés elle est obligée de passer à la caisse. Pour donner un aperçu de la popularité de GitHub en 2013, il y avait plus de 3,5 millions d’utilisateurs ainsi que plus de 10 millions de dépôts. Dans le même type de service, il existe Bitbucket qui se base sur un autre business model que GitHub.

4. *Pourquoi choisir Git comme outil gestion de version*

Tout d'abord, comme nous l'avons appris dans les pages précédentes, Git est un système libre. Il ne peut donc pas y avoir de limitations contractuelles sur l'utilisation de Git. Quel que soit le nombre de collaborateurs, de projets ou de mises à jour, Git sera toujours gratuit.
Git possède aussi tous les avantages liés aux systèmes de gestion de versions décentralisés. C'est-àdire
qu'il est possible de travailler en mode déconnecté avec son dépôt tout en gardant des fonctionnalités avancées de collaboration lorsqu'on est connecté.
Git possède également un autre avantage important : sa performance. Git intègre beaucoup d'outils internes pour le rendre très performant (notamment pour l'utilisation des branches).
Un dernier avantage très important, Git est actuellement le système le plus populaire et il gagne en popularité chaque année. Il est difficile d'avoir des statistiques précises sur les systèmes de gestion de versions installés et utilisés dans les entreprises. Néanmoins, il est possible de juger de la popularité de certains termes utilisés lors de recherches Google grâce à Google Trends. Il est donc possible de prendre conscience du volume de recherches de certains mots-clés. Voici le graphique comparant Git, Mercurial et Apache Subversion.



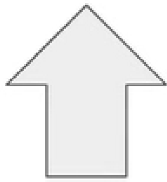
Stackoverflow propose également un outil permettant de comparer la popularité des différents tags. L'image, ci-dessous présente les résultats de comparaison de Git, Mercurial et Subversion (SVN). Les résulats sont exprimés sur l'axe des ordonnées en pourcentage de questions pour chaque mois entre 2008 et 2019.

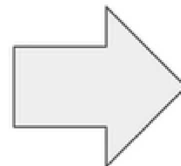


III) Qu’est-ce Git

Git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre et gratuit, créé en 2005 par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. Le principal contributeur actuel de Git, et ce depuis plus de 16 ans, est Junio C Hamano.

Depuis les années 2010, il s’agit du logiciel de gestion de versions le plus populaire dans le développement logiciel et web, qui est utilisé par des dizaines de millions de personnes, sur tous les environnements (Windows, Mac, Linux). Git est aussi le système à la base du célèbre site web GitHub, le plus important hébergeur de code informatique.





IV) Architecture générale de Git

1. Généralités

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.

NB : A la différence du système de gestion de version centralisée, il est possible de travailler hors connexion. Aussi, chacun peut travailler à son rythme sur son dépôt local et permet d'avoir plus de sécurité, car il n'y a pas qu'un seul dépôt de référence, mais plusieurs.

2. 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.

3. *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.

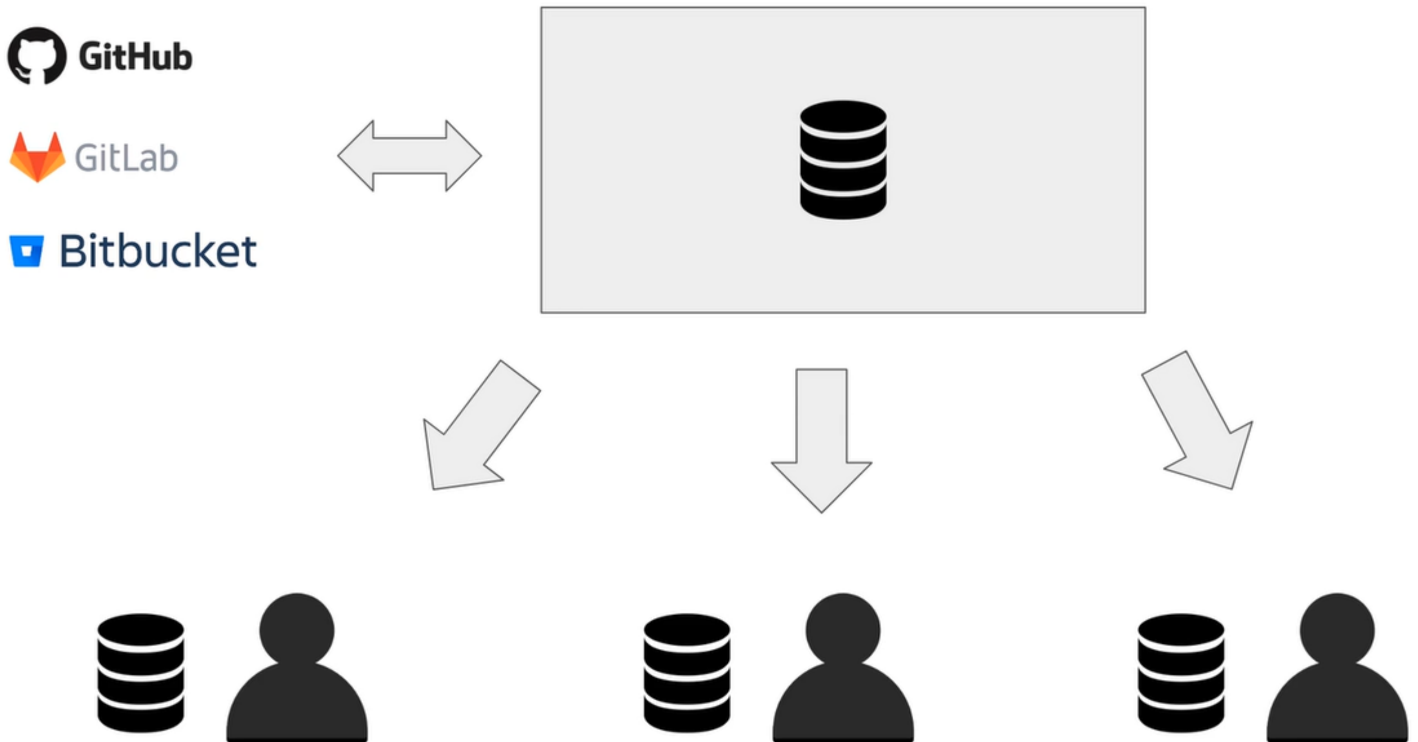
Les clés étant les sommes de contrôles SHA-1 et les valeurs le contenu des objets Git (notamment les fichiers).

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.

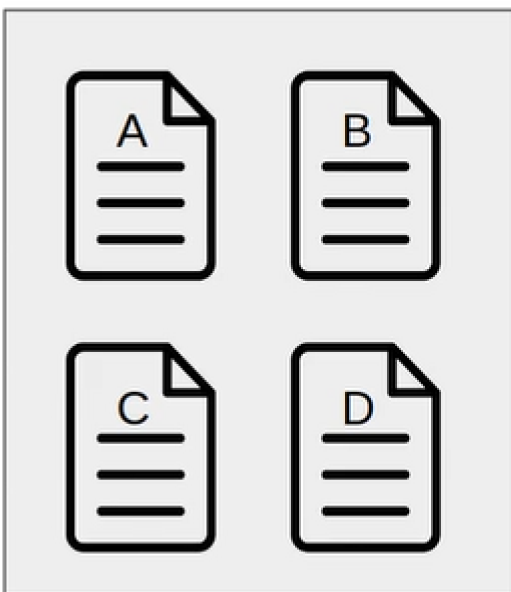
V) Qu'est-ce qu'un dépôt Git

1. Introduction

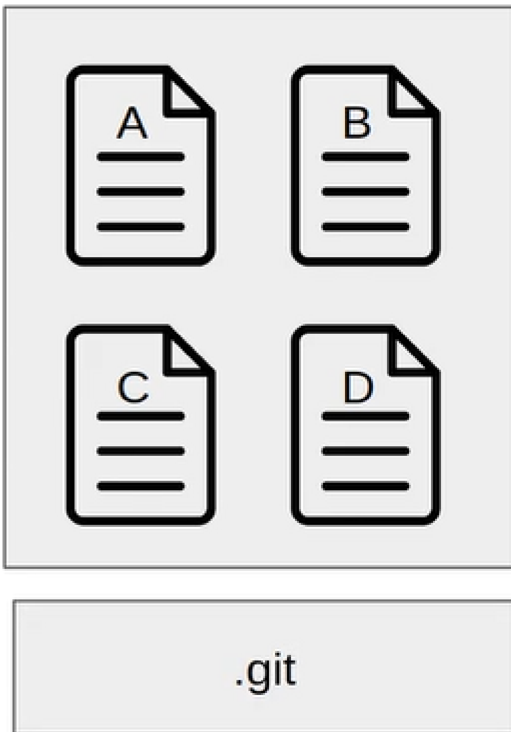
Supposons qu'on ait l'architecture d'un système de contrôle de version décentralisé ci-dessous :



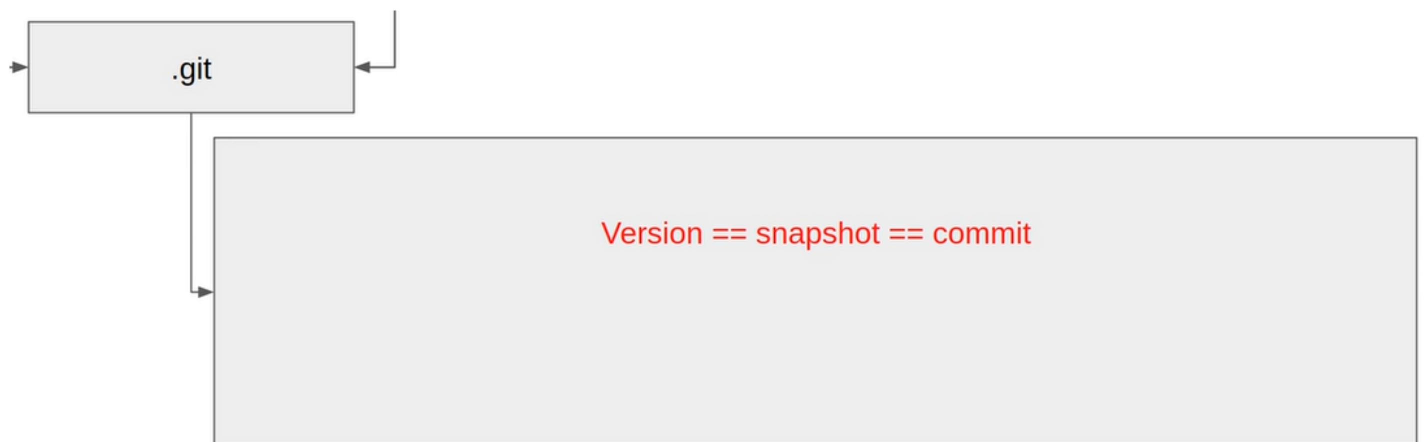
Chaque développeur a sur sa machine en local une copie du dépôt Git.
Prenons le cas des dépôts en particulier contenant les fichiers A,B,C et D.



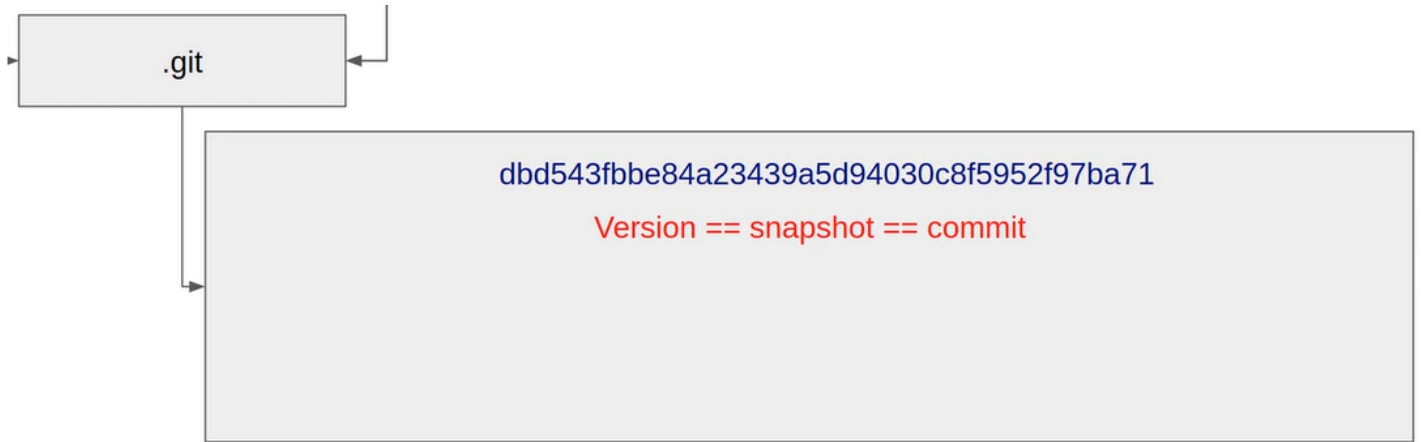
Si on initialise un repertoire git on se retrouve avec :



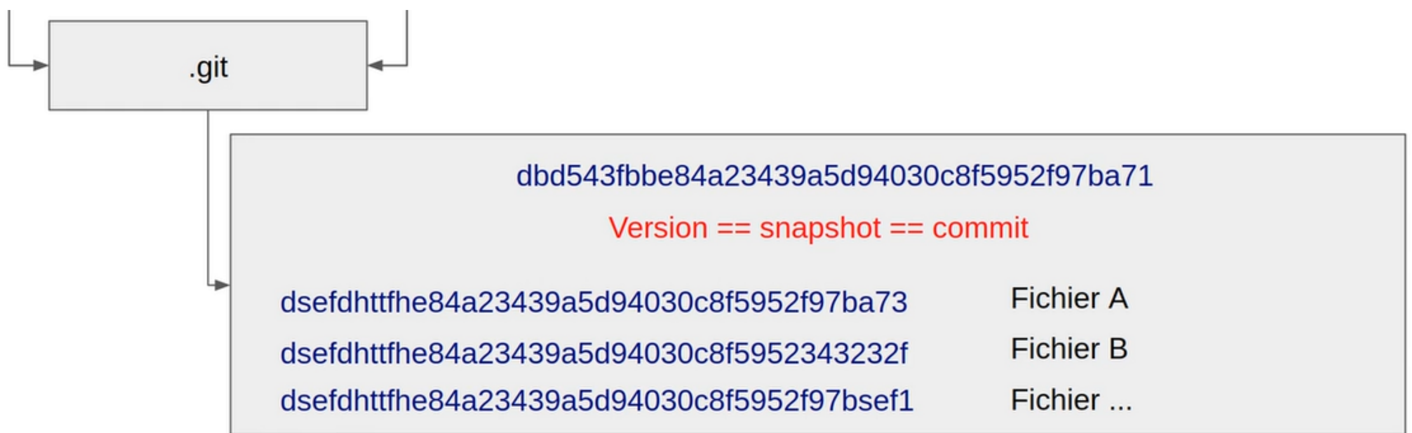
Dans le repertoire contenant les dossiers A,B,C et D il va apparaitre le sous dossier .git.
A un instant t Git va créer une version par rapport à l'état du repositoty Git.



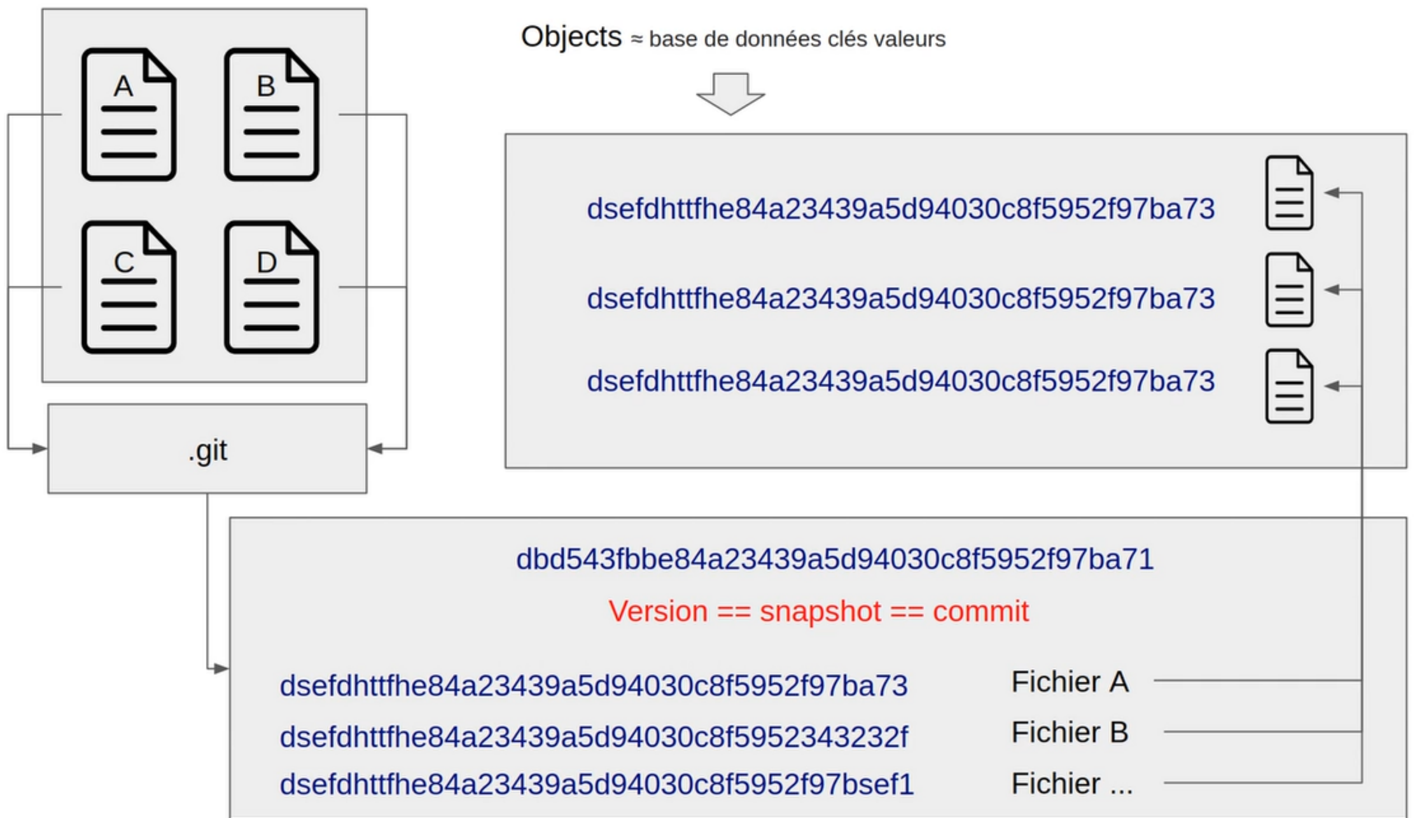
Dans un repos Git chaque version correspond à un commit avec son Hash.



On associe à la version du commit un ensemble de Hash correspondant aux états des fichier A,B … avec leur Hash correspondant.

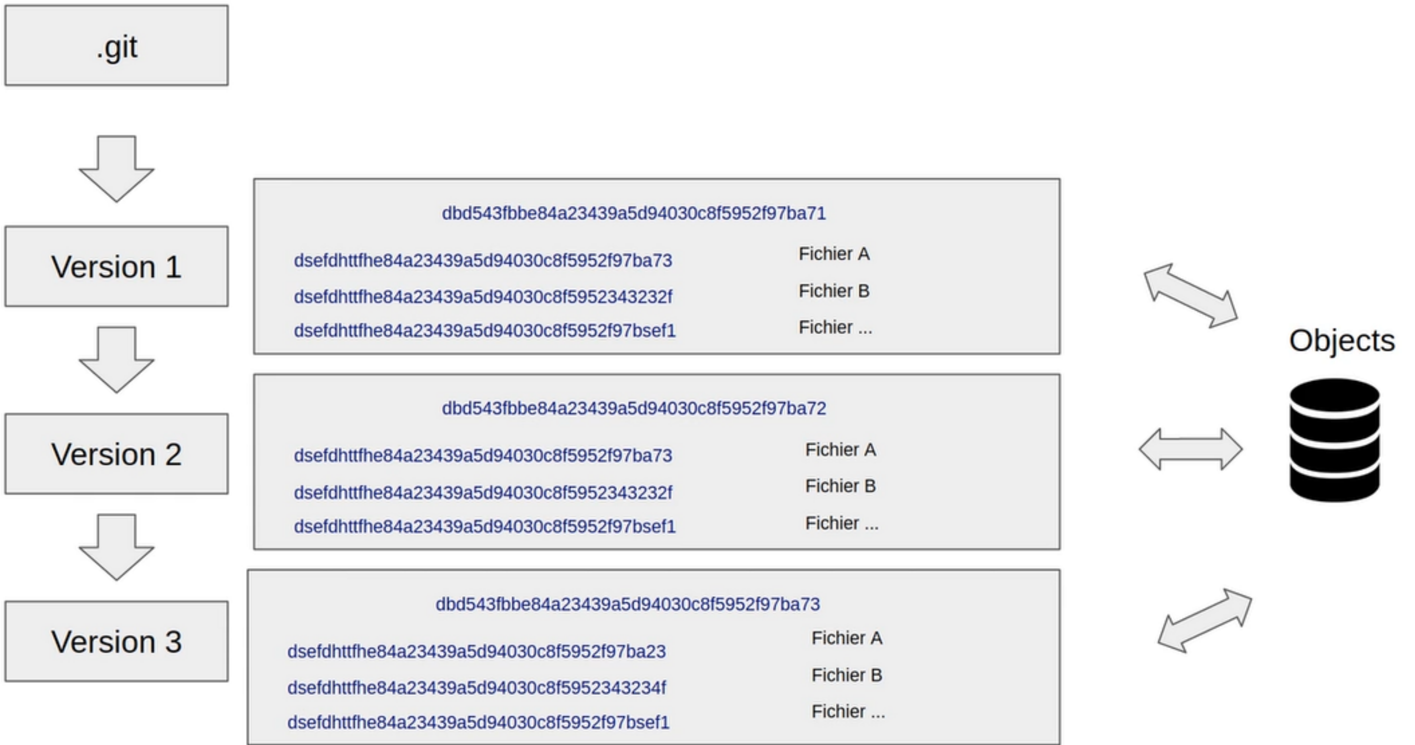


Il faut noter que les fichiers **A,B,C** et **D** ne sont pas stockés dans le commit mais dans la base de donnée clés valeurs de Objects.



Nous obtenons pour plusieurs version de commit :

A chaque version on recupere le contenu du fichier dans le dossier Objects.


.git
Version 1
Version 2
Version 3
dbd543fbbe84a23439a5d94030c8f5952f97ba71
dsefdhttfhe84a23439a5d94030c8f5952f97ba73
Fichier A
dsefdhttfhe84a23439a5d94030c8f5952343232f
Fichier B
dsefdhttfhe84a23439a5d94030c8f5952f97bsef1
Fichier ...
dbd543fbbe84a23439a5d94030c8f5952f97ba72
dsefdhttfhe84a23439a5d94030c8f5952f97ba73
Fichier A
dsefdhttfhe84a23439a5d94030c8f5952343232f
Fichier B
dsefdhttfhe84a23439a5d94030c8f5952f97bsef1
Fichier ...
dbd543fbbe84a23439a5d94030c8f5952f97ba73
dsefdhttfhe84a23439a5d94030c8f5952f97ba23
Fichier A
dsefdhttfhe84a23439a5d94030c8f5952343234f
Fichier B
dsefdhttfhe84a23439a5d94030c8f5952f97bsef1
Fichier ...
Objects

2. *Le contenu du dossier .git*

Chacun de ces fichiers ou dossiers contient des éléments que Git utilise pour suivre notre code. Voici une liste des fichiers et dossiers et leur contenu :

- **HEAD** : contient la référence du commit à partir duquel on travaille. C'est une référence qui est dépendante de la branche sur laquelle nous sommes situés. HEAD désignera le commit le plus récent d'une branche et par défaut le commit le plus récent de la branche courante.

- **branches** : ce dossier contient les branches du projet. Les branches sont des versions qui divergent du développement principal et sont particulièrement utiles pour développer de nouvelles fonctionnalités sans créer de conflit avec la version stable du projet. Les branches seront abordées plus en détail dans le chapitre Les branches et les tags.

- **config** : ce fichier contient les éléments de configuration et les alias propres au dépôt.

- **description** : ce fichier sera affiché lors de l'utilisation de l'interface web GitWeb.

- **hooks** : ce dossier contient les hooks du dépôt. Les hooks sont des mécanismes de contrôle utilisés par Git. Il est possible de créer ses propres hooks pour répondre à des besoins spécifiques.

- **info** : ce dossier contient un fichier **exclude** utilisé pour demander à Git d'ignorer les fichiers spécifiés. Une autre possibilité existe pour ignorer des fichiers, elle sera abordée dans le
- **objects** : contient tous les objets de notre projet, c'est-à-dire que c'est dans ce dossier que seront placées toutes les informations concernant nos fichiers, dossiers, commits, ou tout autre objet que Git a besoin de traiter.

- **refs** : ce dossier comprend les références qui pointent vers des commits. Ces références sont en réalité soit des branches, soit des tags qui seront abordés dans le chapitre Les branches et les tags.