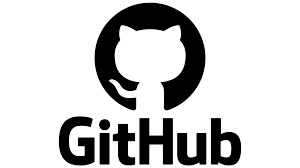
Rapport du projet de versionning avec GIT



SOMMAIRE :

Historique de la gestion de versions et son utilité

Principes fondamentaux de GIT

Fonctionnalités avancées de GIT

GIT HUB et ses alternatives

Quelques cas d’utilisations

Conclusion

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

La gestion de versions : un pilier essentiel du développement collaboratif :

Les premiers systèmes de gestion de versions (VCS) ont vu le jour dans les années 1970 avec des méthodes manuelles de suivi des modifications. En 1982, le système **RCS** a été l'un des premiers à automatiser cette gestion, suivi par **CVS** en 1986, permettant la gestion simultanée des versions par plusieurs utilisateurs. En 2000, **Subversion (SVN)** a amélioré CVS, notamment dans la gestion des fichiers binaires et des conflits. Puis, en 2005, **Git**, créé par Linus Torvalds, a révolutionné la gestion des versions en introduisant un modèle décentralisé, offrant plus de flexibilité et de gestion des branches. En 2008, **GitHub** a popularisé Git en ajoutant des fonctionnalités sociales et de collaboration en ligne, le rendant incontournable pour les projets open-source.

Un système de gestion de versions (*Version Control System* ou VCS) est un outil permettant de suivre et de gérer les modifications apportées aux fichiers d’un projet au fil du temps. Il offre aux développeurs la possibilité de collaborer efficacement, de revenir à des versions précédentes en cas d’erreur et d’organiser le travail en différentes branches pour éviter les conflits de code.

Les logiciels de gestion de versions sont un élément central du DevOps, car ils facilitent la collaboration et l’itération rapide sur le code source d’un projet. Lorsque vous créez un nouveau dépôt, une branche principale est automatiquement ouverte, souvent appelée "branche principale" ou "branche maître" (main ou master). Ce tronc constitue la base du code, qui est progressivement intégré, compilé et déployé vers l’utilisateur final.

**Les différents types de gestion de versions**

Il existe plusieurs approches de la gestion de versions, chacune avec ses avantages et inconvénients :

1. Gestion de versions locale : une approche simple mais limitée

Les logiciels de gestion de versions locaux sont la forme la plus basique de gestion de versions, principalement utilisés par des développeurs indépendants. Toutes les données du projet sont stockées sur un seul ordinateur, et les modifications sont enregistrées sous forme de correctifs successifs. Chaque correctif ne contient que les mises à jour effectuées depuis la version précédente.

Toutefois, cette approche présente des inconvénients majeurs. Comme les fichiers ne sont stockés que sur un seul appareil, la collaboration avec d'autres développeurs est complexe. De plus, si la machine est endommagée ou compromise, il peut être difficile, voire impossible, de restaurer les données perdues.

Si un projet commence à évoluer, même en passant d'un seul développeur à une petite équipe, il est fortement recommandé de passer à une gestion de versions centralisée ou partagée pour plus de flexibilité et de sécurité.

2. Gestion de versions centralisée : un flux de travail structuré mais dépendant du réseau

Les logiciels de gestion de versions centralisés fonctionnent selon un principe de connexion à un serveur principal. Toute modification du code source est immédiatement stockée dans un dépôt unique, sous forme de nouvelles versions successives. Cette approche permet une meilleure organisation et une sécurité accrue, car aucun clonage du dépôt n'est nécessaire sur plusieurs machines.

Cependant, la gestion de versions centralisée présente aussi des inconvénients. Comme tous les développeurs travaillent sur une seule version du projet stockée sur un serveur, une interruption de service ou un problème réseau peut gravement ralentir le travail. De plus, plus le nombre de contributeurs augmente, plus la gestion des modifications devient complexe, ce qui peut entraîner des conflits lors des fusions de code.

Les logiciels de gestion de versions centralisés sont idéaux pour les équipes recherchant une solution facile à configurer et à gérer, à condition de disposer d’une infrastructure réseau fiable.

3. Gestion de versions partagée (distribuée) : autonomie et flexibilité

Les logiciels de gestion de versions partagés (ou distribués) permettent d’enregistrer, de créer des branches et de fusionner des modifications sans nécessiter une connexion constante au serveur principal. Chaque développeur dispose d’un dépôt cloné stocké localement ou dans le cloud, ce qui lui permet de travailler indépendamment et de synchroniser ses modifications uniquement lorsqu’il le souhaite.

Cette approche offre plusieurs avantages :

* Travail hors ligne possible : les développeurs peuvent travailler sans connexion Internet et n’ont besoin de se connecter que pour synchroniser leurs changements.
* Moins de dépendance au réseau : contrairement aux solutions centralisées, les interruptions de connexion n’empêchent pas de progresser sur le projet.
* Collaboration efficace : chaque contributeur peut avancer à son propre rythme sans affecter directement la branche principale.

Cependant, les logiciels de gestion de versions partagés peuvent présenter des temps d’attente importants lors du premier téléchargement du projet, surtout pour des dépôts volumineux avec de nombreux fichiers binaires. De plus, la gestion du stockage peut devenir un enjeu pour les studios travaillant avec des fichiers très lourds.

Pour les entreprises et équipes cherchant un équilibre entre flexibilité et performance, une gestion de versions partagée est souvent la solution idéale.

L'importance des branches dans la gestion de versions :

Les branches jouent un rôle clé en permettant aux développeurs de travailler sur des modifications sans impacter le code principal. Elles offrent un historique complet des changements effectués et évitent d’avoir à gérer l'intégralité de l’historique sur un serveur unique. Un logiciel de gestion de versions permet de créer, fusionner et organiser ces branches de manière fluide. Si un développeur n’est pas prêt à intégrer ses modifications directement dans le tronc, il peut les regrouper dans une branche distincte avant de les fusionner ultérieurement.

Une stratégie efficace de gestion des branches est essentielle pour éviter les conflits de code et les erreurs de compilation. Heureusement, ces logiciels facilitent la synchronisation avec la branche principale et la résolution des conflits, même après l’intégration du code.

Pourquoi utiliser un logiciel de gestion de versions ?

Travailler avec un système de gestion de versions garantit une sauvegarde continue du projet. Chaque mise à jour individuelle est regroupée sous forme de « commit » avant d’être fusionnée dans le code source principal. Ces outils assurent un suivi précis de toutes les modifications apportées aux fichiers, et en cas d’erreur ou de régression, il est possible de revenir instantanément à une version précédente.

De plus, la gestion de versions offre un journal détaillé recensant toutes les modifications, les auteurs et les différentes versions du code. Grâce à ces logiciels, les équipes peuvent collaborer simultanément sur plusieurs fichiers, en toute fluidité, tout en assurant une intégration optimale des changements dans un dépôt unique.

En fournissant un historique complet du projet et la possibilité de revenir sur des erreurs, ces outils simplifient le suivi des bugs et facilitent la restauration de fonctionnalités supprimées. Ainsi, ils sont indispensables pour garantir un développement agile, sécurisé et efficace.

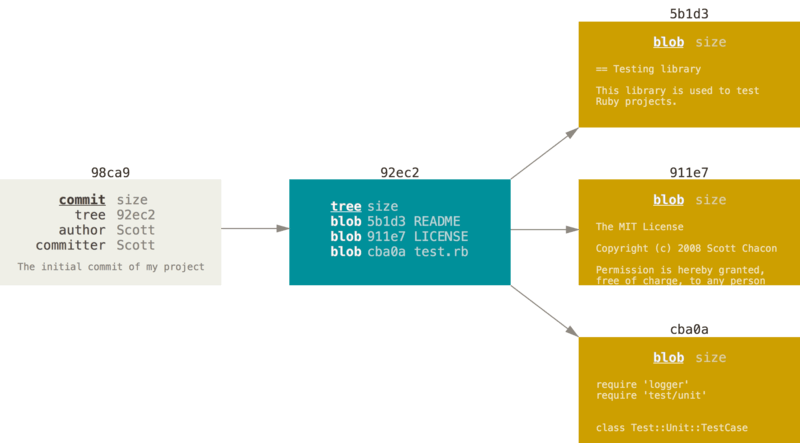
Conclusion

Les logiciels de gestion de versions sont des outils incontournables pour tout projet de développement, qu’il soit individuel ou en équipe. En fonction des besoins et de l’organisation du travail, il est possible d’opter pour une gestion de versions locale, centralisée ou partagée.

🔹 Pour un développeur solo, une gestion locale peut suffire, mais elle devient rapidement limitée.  
 🔹 Pour une petite équipe, une solution centralisée peut être intéressante pour garder un contrôle strict des modifications.  
 🔹 Pour une équipe plus grande ou un projet en pleine croissance, une gestion de versions partagée offre une plus grande flexibilité et une meilleure résilience aux problèmes techniques.

Choisir le bon système de gestion de versions permet d’optimiser le flux de travail, de minimiser les erreurs et d’améliorer la collaboration entre les membres d’une équipe.

Schéma d’illustration du VCS:

Schém

1. Les fondamentaux de la virtualisation :
2. C’est quoi Git ?

Créer le 29 avril 2005 par Linux Torvald, Git est un système de contrôle de version open source. Concrètement, c’est un outil qui te permet de traquer tous les fichiers de ton projet.

Git est un outil de contrôle de version qui est rapidement devenu incontournable dans l'écosystème du développement logiciel. Grâce à sa capacité à conserver un historique minutieux des modifications apportées sur un projet, Git représente un allié précieux pour tout développeur soucieux de gérer efficacement ses projets. Maîtriser Git est, de ce fait, maintenant une compétence essentielle à connaître pour quiconque souhaite exceller dans le domaine du développement logiciel.

Git est un outil qui va te permettre de savoir qui a touché à quel fichier, quand et comment grâce à son versionning . Chaque développeur va pouvoir travailler en parallèle avec leur propre copie du projet sur une branche personnelle. Et pour s’assurer que les développeurs ne se gênent pas lors de la manipulation des fichiers, un système de branche.

1. Les Concepts clés de Git

Dépôt (Repository) :  
Un dépôt Git est un dossier contenant l'ensemble des fichiers du projet ainsi que l'historique complet des modifications. Il existe deux types de dépôts :

Local : Il se trouve sur ton ordinateur.

Distant : Il est stocké sur un serveur (par exemple, sur GitHub ou GitLab). Le dépôt distant permet de centraliser le projet et de faciliter la collaboration entre plusieurs personnes.

2.1. Les commit

Un commit est une sorte de "photo" de l'état actuel du projet à un moment donné. Il contient des informations sur les fichiers modifiés et un message qui décrit les changements effectués. Chaque commit est identifié par un identifiant unique, ce qui permet de revenir à une version antérieure si besoin.

2.2. Les branches

Une branche est une copie du projet sur laquelle tu peux effectuer des modifications sans affecter la version principale du projet. Les branches sont utiles pour développer de nouvelles fonctionnalités ou corriger des bugs en parallèle de la version stable du projet. La branche principale est généralement appelée main ou master.

1. Fusion (Merge) :  
   Une fois que les modifications effectuées sur une branche sont terminées, on peut les fusionner avec la branche principale. Git tente de combiner les changements automatiquement. Si deux personnes ont modifié la même ligne de code dans deux branches différentes, un conflit peut survenir et devra être résolu manuellement.
2. Staging Area :  
   Avant de commettre des modifications dans Git, il est possible de préparer ces changements dans une zone intermédiaire appelée staging area (ou index). Cela permet de sélectionner précisément les modifications à inclure dans le commit. Une fois les modifications prêtes, tu peux les valider (committer).
3. **Commandes de base Git et leurs effets**

🔹 **Configuration**

git config --global user.name "Ton Nom" → Définit le nom d’utilisateur global.

git config --global user.email "[ton.email@example.com](mailto:ton.email@example.com)" → Définit l’e-mail global.

git config --list → Affiche la configuration actuelle de Git.

**Initialisation & Clonage**

git init → Initialise un nouveau dépôt Git.

git clone <URL> → Clone un dépôt distant en local.

**Suivi des fichiers**

git status → Affiche l’état des fichiers (suivis, modifiés, en attente de commit).

git add <fichier> → Ajoute un fichier à la zone de staging.

git add . → Ajoute tous les fichiers modifiés à la zone de staging.

git reset <fichier> → Retire un fichier de la zone de staging.

git rm <fichier> → Supprime un fichier et enregistre la suppression dans Git.

**Validation des modifications**

git commit -m "Message" → Valide les modifications avec un message explicatif.

git commit --amend → Modifie le dernier commit (ex. : pour corriger un message).

**Historique & Affichage**

git log → Affiche l’historique des commits.

git log --oneline --graph --all → Affiche un historique simplifié et visuel.

git diff → Montre les différences entre fichiers modifiés et validés.

**Gestion des branches**

git branch → Liste les branches existantes.

git branch <nom> → Crée une nouvelle branche.

git checkout <nom> → Change de branche.

git checkout -b <nom> → Crée et bascule sur une nouvelle branche.

git merge <nom> → Fusionne une branche dans la branche actuelle.

git branch -d <nom> → Supprime une branche.

**Gestion des dépôts distants**

git remote add origin <URL> → Associe un dépôt distant.

git remote -v → Liste les dépôts distants configurés.

git push -u origin <branche> → Envoie les commits vers un dépôt distant.

git pull origin <branche> → Récupère et fusionne les modifications d’un dépôt distant.

git fetch origin → Récupère les modifications distantes sans les fusionner.

**Annulation & Correction**

git reset --soft HEAD~1 → Annule le dernier commit sans supprimer les modifications.

git reset --hard HEAD~1 → Annule le dernier commit et supprime les modifications.

git revert <commit> → Crée un commit qui annule un commit spécifique.

## **1. Commandes avancées en Git**

Git propose de nombreuses commandes avancées permettant une gestion plus fine des dépôts et de l'historique. Voici quelques commandes utiles :

### **Gestion des modifications et de l'historique**

* **git stash** : Sauvegarde temporairement les modifications sans les valider.
* **git stash pop** : Récupère les modifications sauvegardées avec stash et les applique.
* **git stash list** : Liste les stashes enregistrés.
* **git stash drop** : Supprime un stash spécifique.
* **git reflog** : Affiche l'historique des actions (même après un reset).
* **git reset --hard HEAD~1** : Annule le dernier commit et supprime les modifications.
* **git reset --soft HEAD~1** : Annule le dernier commit mais conserve les modifications en staging.
* **git revert <commit>** : Crée un commit inverse pour annuler un commit spécifique.
* **git cherry-pick <commit>** : Applique un commit spécifique sur la branche courante.
* **git rebase -i HEAD~3** : Réécrit les trois derniers commits (modification, fusion, suppression).

### **Débogage et navigation**

* **git bisect start** : Recherche un commit problématique en mode binaire.
* **git worktree add ../new-branch new-branch** : Crée un nouveau dossier avec une autre branche.
* **git log --graph --oneline --all** : Affiche un historique compact et visuel des branches.
* **git clean -df** : Supprime les fichiers non suivis et les dossiers non vides.
* **git shortlog -sn** : Affiche la liste des contributeurs et leur nombre de commits.
* **git grep "mot-clé"** : Recherche un mot-clé dans tout le dépôt.

## **2. Alternatives à GitHub**

Bien que GitHub soit la plateforme la plus populaire pour la gestion de code source, d'autres solutions existent avec des spécificités différentes :

### **2.1 GitLab** 🚀

* Similaire à GitHub, mais avec plus d’options de **CI/CD intégrées**.
* Peut être **auto-hébergé** pour plus de contrôle.
* Offrant des fonctionnalités avancées de gestion des dépôts et de la sécurité.

### **2.2 Bitbucket** 🏗️

* Optimisé pour les entreprises, avec **intégration Jira** et support des **dépôts privés gratuits**.
* Compatible avec **Git** et **Mercurial** (bien que Mercurial ne soit plus officiellement pris en charge par Atlassian).

### **2.3 Gitea** 🖥️

* Version **légère et auto-hébergeable** de GitHub.
* Idéal pour des projets personnels ou en entreprise, offrant plus d'indépendance.

### **2.4 SourceForge** 📂

* Anciennement l'une des plateformes principales pour l'open-source.
* Toujours utilisée pour certains projets open-source avec des outils de gestion de bugs et forums.

### **2.5 AWS CodeCommit** ☁️

* Service Git **hébergé sur Amazon Web Services (AWS)**.
* Idéal pour les déploiements cloud et l’intégration avec d'autres services

CONCLUSION

En résumé, Git permet la gestion de versions adaptées aux projets en entreprise comme personnels. Son évolution a conduit à l'émergence de plateformes comme GitHub et GitLab, devenues incontournables dans le monde du développement. Elles apportent une dimension collaborative essentielle à l'informatique moderne.