C’est quoi Git ?

Git est un logiciel de versioning créé en 2005 par Linus Torvalds, le créateur de Linux.

C’est un système de versionning : copie à chaque version du projet. C’est le plus populaire de nos jours.

Un logiciel de versioning, ou logiciel de gestion de version est un logiciel qui permet de conserver un historique des modifications effectuées sur un projet afin de pouvoir rapidement identifier les changements effectuées et de revenir à une ancienne version en cas de problème.

Les logiciels de gestion de versions sont quasiment incontournables aujourd’hui car ils facilitent grandement la gestion de projets et car ils permettent de travailler en équipe de manière beaucoup plus efficace.

Parmi les logiciels de gestion de versions, Git est le leader incontesté et il est donc indispensable pour tout développeur de savoir utiliser Git.

Ce logiciel est un outil incontournables aux dev et aux devops.

Expliquer la problematique des sauvagardes (differentes versions, cas de bugs, sauvegarde en ligne) et la pb de travail en equipe

Resolution de pb du contenu centralisé

**GitHub** est un service web d'[hébergement](https://fr.wikipedia.org/wiki/H%C3%A9bergeur_web) et de gestion de développement de logiciels, utilisant le [logiciel de gestion de versions](https://fr.wikipedia.org/wiki/Logiciel_de_gestion_de_versions) [Git](https://fr.wikipedia.org/wiki/Git). Ce site est développé en [Ruby on Rails](https://fr.wikipedia.org/wiki/Ruby_on_Rails) et [Erlang](https://fr.wikipedia.org/wiki/Erlang_(langage)) par Chris Wanstrath, PJ Hyett et Tom Preston-Werner.

GitHub est un service en ligne qui permet d’héberger des dépôts ou repo Git. C’est le plus grand hébergeur de dépôts Git du monde.

Une grande partie des dépôts hébergés sur GitHub sont publics, ce qui signifie que n’importe qui peut télécharger le code de ces dépôts et contribuer à leur développement en proposant de nouvelles fonctionnalités.

Git permet à différentes versions d’un même fichier de coexister. Les développeurs travaillant avec Git ont accès à l’historique des modifications pour l’intégralité du projet et peuvent ainsi savoir quels changements ont été faits par rapport à leur version des fichiers, qui a fait ces changements, etc.

Types de modeles de systemes de versionning : centralisé et décentralisé

Centralisé : SVN (tres peu utilisé de nos jours). // pb : si un dev travaille sur un fichier, il est verrouillé et personne ne peut y accéder.

Décentralisé : Dans un modèle distribué, le principe de base est opposé : le code source du projet est toujours hébergé sur un serveur distant mais chaque utilisateur est invité à télécharger et à héberger l’intégralité du code source du projet sur sa propre machine.

Le modèle distribué a été popularisé par Git et présente différents avantages notables par rapport au modèle centralisé :

* Simplicité / flexibilité du travail : Comme chaque utilisateur peut héberger le code du projet, on n’a plus besoin d’être constamment connecté à un serveur central et on peut donc travailler en ligne sur sa propre machine ;
* Sécurité : Comme chaque utilisateur possède le code complet d’un projet, on peut utiliser la copie du projet d’un utilisateur comme back-up en cas de corruption du serveur central.

Dans le langage des systèmes de gestion de version, la copie de l’intégralité des fichiers d’un projet et de leur version située sur le serveur central est appelé un dépôt. Git appelle également cela “repository” ou “repo” en abrégé.

Pour récapituler, et afin d’être bien clair sur ce point : Git est un logiciel de gestion de version tandis que GitHub est un service en ligne d’hébergement de dépôts Git qui fait office de serveur central pour ces dépôts.

On a deux possibilités de manipuler GIT, via la console cad les lignes de commandes ou via GUI

La quasi-totalité des dev utilisent la console

**Installation de Git**

Telecharger git sur son site officiel et l’installer

**Paramétrage de Git**

Definir des paramètres : --global : cad sur tt le système (donc applicable pour tous les depots)

git config --global user.name "Rayen HIDRI"

git config --global user.email [rayen.hidri@gmail.com](mailto:rayen.hidri@gmail.com)

git config --global core.editor notepad.exe

**voir les option d’une commande**

git commande --help

ou

git help commande

**cloner un depot git**

il faut se positionner dans le dossier parent de son projet puis lancer cette commande

git clone lien-depot.git

**initiation d’un depot Git**

git init

NB : avant de lancer cette commande, les fichiers ne sont pas surveillés

Apres son execution, tous les fichiers et dossiers du depot sont surveillés (ajout, modif, supp)

Et on aura les elements suivants :

1. le répertoire de travail (working tree),
2. la zone d’index (staging area) : fichiers prets à etre commités
3. validés (committed) : fichiers commités

**creation de depot sur github**

la creation se fait sur son espace perso sur github

**relier son depot local à son depot github**

git remote add origin lien-depot.git

**afficher les noms des remotes**

Git remote show

**afficher le lien du remote**

Git remote show origin

**ajout de fichier à stage area**

git add fich.txt

🡺 ajoute un seul fichier

git add .

🡺 ajoute tous les fichiers qui ont subi un changement d’état (ajout, modif, supp) à la stage area

**Attention : les commandes restore et reset sont des commandes dangereuses !!**

Si vous souhaitez enlever un fichier de la zone d’index, il faut utiliser la commande restore, deux options sont alors possibles.

**Supprimer des fichier de la stage area**

git restore --staged fichier

**Restauration du fichier dans son état avant modification (annule ses modif/ il faut pas qu’il soit dans la stage area)**

git restore fichier

La première version fait exactement l’inverse de git add, la deuxième version supprime totalement les dernières modifications non commitées, soyez donc vigilent ! Cette commande prend obligatoirement un chemin de fichier.

**Supprimer tous les fichier de la stage area**

git restore --staged .

NB : Si vous voulez tout restaurer, pensez au caractère « . » (point) 🡺

**supp tous les fichiers de la stage area**

git restore .

**restore et reset**

Historiquement, on utilise à la commande reset pour annuler des changements et enlever des fichiers de l’index. Cependant, cette commande possède de très nombreuses possibilités et, mal comprise, elle pouvait mener à des effets désastreux.

Git a ainsi introduit dans la version 2.24 la commande restore afin d’annuler des changements. Toutefois, la commande reset reste tout à fait fonctionnelle. Nous verrons dans une prochaine étape ses possibilités.

Comme il y a de grandes chances que vous soyez confronté à la commande reset, notamment sur les doc et les forums, voici comment l’utiliser pour annuler des changements.

**Restaurer le répertoire de travail dans son état d'avant modif**

git reset --hard

Par ailleurs, reset --hard ne peut s’utiliser en précisant un chemin de fichier, cela s’applique donc forcément à tout le projet.

**Verifier l etat des fichiers**

git status

**faire un commit**

git commit –m  "message du commit"

**NB :** A chaque fois qu’on souhaite enregistrer une modification de fichier ou un ajout de fichier dans le dépôt Git, on va devoir utiliser les commandes git add et git commit comme on a vu précédemment.

J’attire ici votre attention sur un point important : le commit (la validation / l’enregistrement en base de données) d’un fichier se basera sur l’état de ce fichier au moment du git add.

Cela signifie que si vous effectuez une commande git add sur un fichier puis modifiez à nouveau le fichier puis effectuez un git commit, c’est le fichier dans son état au moment du dernier git add qui sera validé et les dernières modifications ne seront donc pas enregistrées dans le dépôt Git.

Si vous souhaitez enregistrer toujours la dernière version d’un fichier, pensez donc bien toujours à effectuer un git add juste avant un git commit. Pour mettre en un coup les fichiers modifiés et déjà sous suivi dans la zone d’index puis pour les valider, vous pouvez également utiliser git commit avec une option -a comme ceci : git commit -a. Cela vous dispense d’avoir à taper git add.

**Basculer d’un commit à l’autre**

git checkout <sha-commit>

pour revenir à la branche concernée, on met

git checkout nom\_branche

**Ecraser et remplacer un commit (sur un commit pas encore pushé)**

Pratique dans le cas de faute ou d’oubli de fichiers

On fait la modif necessaire puis il faut les ajouter puis faire un push avec l’option force pour forcer le push, sinon on aura une erreur

NB : cette manip change le SHA du commit

git add .

git commit --amend

Apres avoir tapé la commande commit --amend, on peut rectifier le message du commit

git psuh origin master --force

**si jamais on veut modifier seulement le message du dernier commit**

git commit --amend -m "nouveau message de commit"

**annuler un commit (xxx : sha du commit)**

annuler l’effet d’un commit, mais il reste dans l’historique

**RQ :** cette commande n’a un effet qu’en local, pour repercuter les changements dans le remote, il faut faire un commit et un push apres

Dans l’historique on voit apparaitre les deux commits (celui qui a été annulé et celui qui a fait l’annulation)

git revert xxxx

pousser les modif local 🡺GH

git push origin branche

recupérer les maj de son depot GH 🡺 local

git pull origin branche

**Empêcher l’indexation de certains fichiers dans Git**

Lorsqu’on dispose d’un projet et qu’on souhaite utiliser Git pour effectuer un suivi de version, il est courant qu’on souhaite exclure certains fichiers du suivi de version comme certains fichiers générés automatiquement, des fichiers de configuration, des fichiers sensibles, etc.

On peut informer Git des fichiers qu’on ne souhaite pas indexer en créant un fichier .gitignore et en ajoutant les différents fichiers qu’on souhaite ignorer. Ce fichier se trouve n’importe où dans le répertoire d’un dépôt Git et s’appliquera au répertoire courant et à tous ses sous-répertoires.

Notez qu’on peut également mentionner des schémas de noms pour exclure tous les fichiers correspondant à ce schéma et qu’on peut même exclure le contenu entier d’un répertoire en écrivant le chemin du répertoire suivi d’un slash.

Exp : creons un dossier qui s’appelle ignorer et mettre dedans des fichiers 🡺 créer un fichier .gitignore à la racine 🡺mettre dedans ignorer/ 🡺 tout le contenu de ce dossier ne sera jamais suivi

On peut utiliser les regex pour spécifier les fichier ou dossiers à ignorer

Exp : \*.log // log/\*.txt

**Suivi des répertoires vides**

Sachez bien que Git ne stocke pas les répertoires vides. Pour palier à cela, lorsqu’un répertoire possède une importance mais est vide dans le répo du projet – logs/ ou config/ sont assez courants – on peut placer un fichier vide afin de forcer Git à prendre le répertoire en compte.

Ce fichier vide est par convention nommé .gitkeep. Il indique par son nom que sa seule fonction est de permettre l’indexation de son répertoire parent. (on le met à l’intérieur du dossier vide, fichier vide qui s’appelle .gitkeep )

**Consulter l’historique**

git log

🡺affiche les commits en ordre chrono descendant, cette commande supporte plusieurs options, les plus importantes  sont --oneline et --pretty (ou -p)

git log --oneline

🡺 affciher l’historique chrono

git log --oneline -p

🡺affciher l’historique + changement pour chaque fichier

NB : utiliser les fleches haut et bas pour se deplacer // faire q pour quitter

Afficher l’historique d’un fichier bien déterminé (param fichier en chemin relatif)

git log --oneline -p fichier

**les branches**

Créer une branche, c’est en quelque sorte comme créer une “copie” de votre projet pour développer et tester de nouvelles fonctionnalités sans impacter le projet de base.

**Lister toutes les branches**

git branch

**NB :** Vous notez que l’étoile à gauche du nom de la branche indique sur quelle branche on se trouve présentement. Par ailleurs, par défaut, lors de l’initialisation du dépôt, Git créé la branche *master*.

**Pour créer une nouvelle branche**, on utilise la commande

git branch nom-de-la-branche

**se deplacer vers une branche**

**NB :** Pour changer de branche, il faut impérativement être dans un état propre, c’est à dire que le répertoire de travail ne peut contenir aucun fichier dans un état modifié. Si c’est le cas, il faut d’abord commiter les changements pour pouvoir changer de branche.

Étant donné que le changement de branche modifie le répertoire de travail, tout changement non commité mènerait à la perte des modifications non sauvegardées.

git checkout nom-branche

depuis la version 2.23, git a introduit la commande switch qui a la meme syntaxe et le meme fonctionnement que checkout

git switch nom-branche

**pour revenir à la branche principale**, on fait

git checkout master

**merge des branches** : il faut se placer sur la branche master puis rapatrier la branche souhaitée

git checkout master

git merge nom-de-la-branche

NB : cas de conflit : si jamais on modifie le même fichier sur deux branches et qu’on rapatrie une dans l’autre, cela crée un conflit 🡺git nous alerte sur le conflit et nous demande de le resoudre avant de terminer la fussion 🡺Utilser la commande

get status

pour voir précisément quels fichiers sont à l’origine du conflit.

git diff master dev

On peut alors ouvrir le fichier à la main et choisir ce qu’on conserve (en supprimant les parties qui ne nous intéressent pas par exemple).

Dès qu’on a terminé l’édition, on va taper une commande

git add

pour marquer le conflit comme résolu. On n’aura alors plus qu’à effectuer un

git commit

git push origin master

pour terminer le commit de fusion.

**Afficher les differences entre branches**

git diff <branche1 > <branche2 >

🡺affiches les differences de b2 par rapport à b1

récupérer les commits de la branche source vers la branche en cours

git rebase source

**Supprimer une branche :** Il ne faut pas être sur la branche en question lors de sa suppression

git branch -d nom-branch

**Contribuer à un projet avec GitHub**

Pour contribuer à un projet sur github (le notre ou public) on adopte le workflow suivant :

1. On copie un projet sur notre espace GitHub ;
2. On créer une branche thématique à partir de master ;
3. On effectue nos modifications / améliorations au projet ;
4. On pousse ensuite la branche locale vers le projet qu’on a copié sur GitHub et on ouvre une pull request depuis GitHub ;
5. Le propriétaire du projet choisit alors de refuser nos modifications ou de les fusionner dans le projet d’origine ;
6. On met à jour notre version du projet en récupérant les dernière modifications à partir du projet d’origine.

On peut également utiliser l’option “fork” de GitHub. Un fork est une copie d’un dépôt distant qui nous permet d’effectuer des modifications sans affecter le projet original.

La différence entre un fork et un clone est que lorsqu’on fork une connexion existe entre notre fork (notre copie) et le projet original. Cela permet notamment de pouvoir très simplement contribuer au projet original en utilisant des pull requests, c’est-à-dire en poussant nos modifications vers le dépôt distant afin qu’elles puissent être examinées par l’auteur du projet original.

Lorsqu’on clone un projet, on ne va pas pouvoir ensuite récupérer les changements à partir du projet d’origine ni contribuer à ce projet à moins que le propriétaire du projet d’origine ne nous accorde des droits spéciaux (privilèges).

Le fork est une fonctionnalité très utile puisqu’elle permet à n’importe quelle personne de pouvoir dupliquer un projet et d’y contribuer tout en garantissant à l’auteur l’intégrité de la copie originale car ce sera à lui de valider ou pas les différentes pull requests des contributeurs.

**Travailler sur le depot avec la meme equipe**

**Inviter ses collaborateurs à travailler sur son depot**

Settings 🡺 manage access🡺invite collaborator 🡺 taper son username 🡺 le collaborateur recevra une invitation par mail 🡺accepter l’invit.

Le collab se met sur la branche master et crée une branche selon la convention de nommage

Exp : numTicket\_nomCollab

Chaque collab se met sur sa branche et fais les modif nécessaires pour traiter son ticket.

Commit et push vers la branche numTicket\_nomCollab

Faire une pull request du compte du collab et l’assigner au compte principal du depot

Si la pull request convient au tech lead, il la valide.

**Eviter que git demande les identifiants à chaque fois**

git config --global credential.helper store