

Git

L'essentiel

par Denis Payet

version : 28-03-2026

➔ Menu « Accès rapide »

Accès rapide :

3 – Dépôt local

Sur le poste de chaque utilisateur

config init add commit
status log

4 – Les branches

Bienvenue dans le multivers !

branch switch merge

5 – Les conflits de fusion

Des modifications qui divergent
sur les 2 branches à fusionner

restore merge --abort

6 – Dépôt distant

Généralement un serveur GitHub

remote -v remote -add remote show
branch -u branche -vv

7 – Les commandes de synchronisation distante

clone fetch pull push
push -u push -f

8 – Les opérations sur l'historique

Voyager dans le temps !

checkout reset --hard checkout -b

Diagnostic régulier

- > **git status** Etat des lieux du dépôt.
→ Permet de voir s'il faut faire : des **add**, un **commit**, un **push** ou un **pull**.
- > **git log** Historique complet des **commits**, et consulter en particulier leur id respectif.
- > **git branch** Liste des **branches locales**.
- > **git fetch**
> **git branch -vv** Afficher la liste des **trackings** et de la situation avance/retard des branches.
→ Permet de voir si des **push** ou des **pull** sont nécessaires.

Diagnostic occasionnel

- > **git config --list** Vérifier son **identité Git**.
- > **git log --online --graph --all** Voir le **graphe des branches** (les points de création et de fusion).
- > **git remote -v** Afficher la liste des **remotes**.
- > **git fetch *nom_remote***
> **git remote show *nom_remote*** Afficher la liste des **branches d'un remote** en particulier.
- > **git fetch**
> **git branch -r** Afficher la liste des **branches de tous les remotes**.

1 – Introduction

VCS

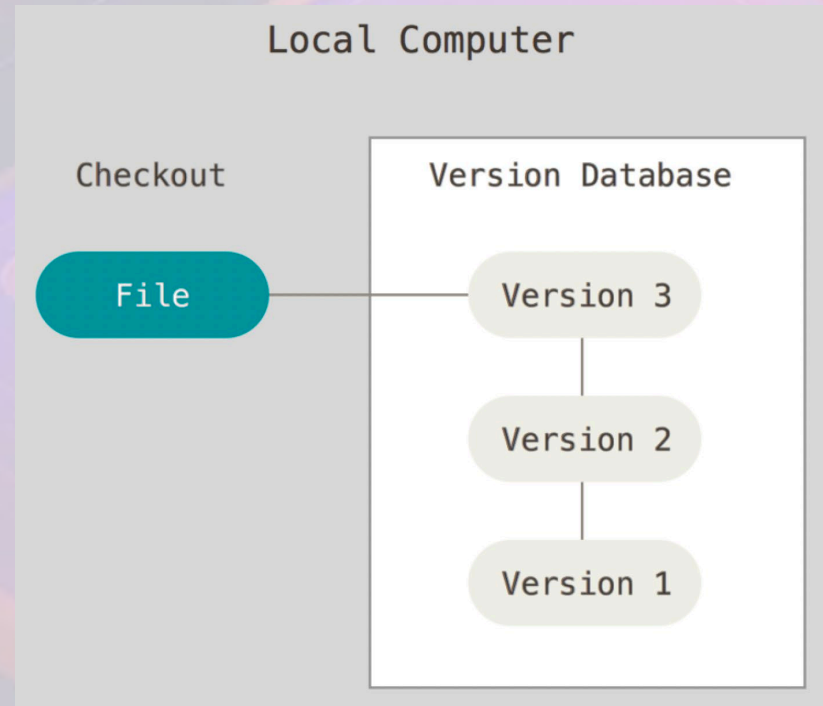
Les Systèmes de Contrôle de Version

Système de gestion de version

- Un **Système de gestion de version**, aussi appelé : **Système Contrôle de Version** (*Version Control System* => **VCS**) est un outil permettant de mémoriser les changements subit par un ensemble de fichier.
- Toutes les activités (création, suppression, modification) sont monitorées par l'outil, puis enregistrées dans une base de données.
- Ainsi, c'est un système qui permet de **suivre l'historique des changements** et, si besoin, de **restaurer une ancienne ancienne version** du contenu (en annulant tous les changements subit depuis cette version).
- Un tel outil est utilisé lorsqu'on travail sur un **projet** ayant des ressources numériques, afin d'assurer la conservation de ces ressources, et la traçabilité de leurs modifications.

VCS Local

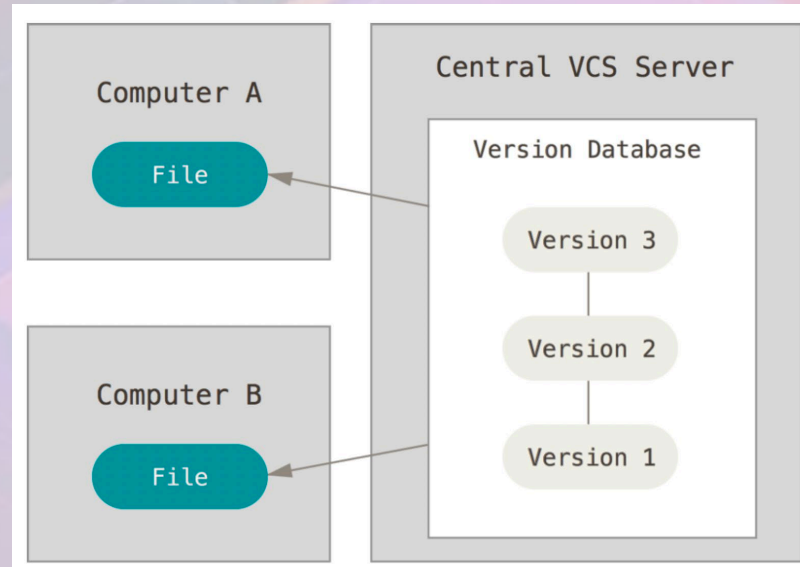
- Un **Système de gestion de version local**, est un VCS qui fonctionne localement sur la machine d'un utilisateur.



➔ Un tel système est suffisant si l'utilisateur est seul à travailler sur le projet, et s'il n'utilise qu'un seul ordinateur.

VCS Centralisé

- Un **Système de gestion de version centralisé**, est un VCS qui fonctionne sur un serveur central.

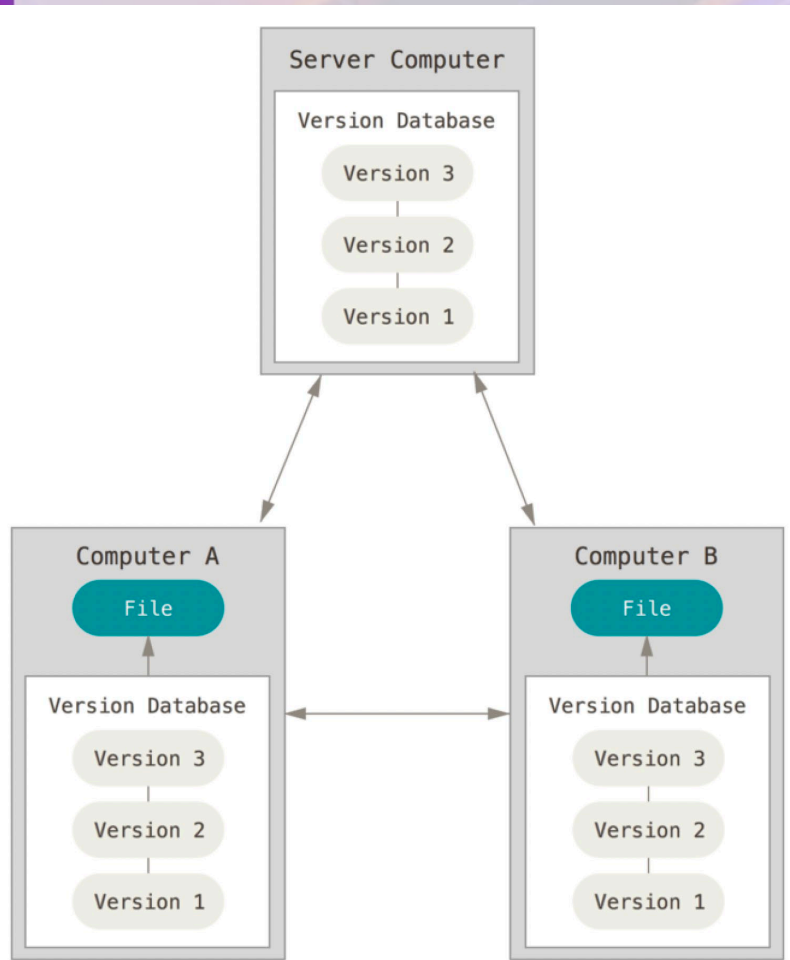


➔ Un tel système permet à plusieurs utilisateurs de travailler sur un même projet.

Mais : il y a un risque de tout perdre si le serveur central tombe en panne. Et, lorsqu'il est hors-ligne personne ne peut plus travailler sur le projet.

VCS Décentralisé

- Un **Système de gestion de version décentralisé**, est un VCS qui est dupliqué sur chaque poste utilisateur, ainsi que sur un (ou plusieurs) serveur(s).



➔ Avec cette architecture, plusieurs utilisateurs peuvent travailler sur un même projet, **et** continuer à travailler même si le serveur est hors-ligne. Dans ce dernier cas, une synchronisation sera opérée dès que cela sera possible.

➔ De plus, si le serveur tombe en panne, il pourra être remplacé, et une synchronisation sera réalisée depuis n'importe quelle instance conservée sur l'un des postes utilisateur.

2 – L'outil Git

Libre & Gratuit

Historique de Git

- **Git** a été créé en 2005 par **Linus Torvalds** (le créateur de Linux)
→ c'est un **logiciel de gestion de versions décentralisé**.
- C'est un logiciel libre et gratuit



- Il a été créé pour remplacer le logiciel propriétaire **BiKeeper**
- Aujourd'hui, ce logiciel **est devenu incontournable** pour conserver et partager le code source des projets.

Si vous souhaitez stocker les ressources d'un projet, **et / ou**, travailler à plusieurs sur ce projet → Vous devez utiliser **Git**.

Documentation & Installation

- Le site officiel de Git est : <https://git-scm.com/>
- ➔ Sur ce site, la documentation complète de Git est disponible en français (et elle peut être téléchargé sous la forme d'un ebook) :

<https://git-scm.com/book/fr/v2>

- Avant d'installer Git, vérifier s'il n'est pas déjà présent en tapant dans une console : `> git --version`
- Au besoin, téléchargez le programme et suivez les directives données sur cette page :

<https://git-scm.com/install/>

Les dépôts

- **Git** permet de **stocker un ensemble de fichier** (le contenu d'un dossier), **en conservant l'historique** de toutes les modifications apportées sur cet ensemble :
 - ➔ l'ajout de nouveaux fichiers
 - ➔ les suppressions de fichiers
 - ➔ les modifications des fichiers
- Le contenu du dossier géré, ainsi que toutes les informations de l'historique, est appelé **un dépôt** (*repository*).
- Ce stockage se fait de façon décentralisée :
 - le dépôt est principalement stocké sur un serveur Git, on l'appelle : **le dépôt distant**.
 - mais chaque utilisateur dispose aussi d'une copie du dépôt sur sa machine, c'est : **le dépôt local**.

3 – Dépôt local

Sur le poste de chaque utilisateur

config

init

add

commit

status

log

Configurer l'utilisateur local

- Afin de fixer toutes les informations nécessaires dans l'historique lors des différentes modifications, l'outil Git a besoin de connaître l'identité de l'utilisateur local (celui qui opère sur le dépôt local).
 - ➔ Tant que l'utilisateur local est indéfini, aucune modification du dépôt ne sera permise.
- Pour indiquer à Git qui est l'utilisateur local il faut utiliser les commandes suivantes :

```
> git config --global user.name "nom de l'utilisateur"  
> git config --global user.email "son_email@email.com"
```

- ➔ Cette configuration n'est à faire qu'une seule fois, sauf si l'utilisateur local change.

Pour vérifier cette configuration : **> git config --list**

Le dépôt local (*local repository*)

- **Le dépôt local** est un dossier qui se trouve sur le poste de l'utilisateur.
- Ce dossier peut par exemple être le dossier principal d'un projet de développement d'un programme. Il contient alors : les fichiers sources, les fichiers de configuration, les ressources images, etc... Mais cela peut-être un dossier de n'importe quel le sorte de projet. Comme par exemple un projet d'écriture d'un livre. Le dossier contient alors les fichiers Words correspondants à chacun des chapitres.

N'importe quel dossier, quelque soit son contenu, peut être transformé en un dépôt local.

➔ Il suffit pour cela d'ouvrir une console, de se placer dans ce dossier, et de taper la commande Git :

```
> git init
```

Contenu d'un dépôt local

- Un dépôt local est un dossier ordinaire, son contenu est donc un ensemble de fichier et de sous-dossier.
- L'action de la commande **git init** se traduit par la création d'un sous-dossier de nom **.git**
 - ➔ C'est ce dossier qui va contenir toutes les informations (la base de données) concernant l'historique des modifications.
 - ➔ Le fait que son nom commence par un point, rend ce dossier invisible sur les systèmes Mac et Linux.
- Pour détruire un dépôt local, et donc faire que le dossier redevienne un dossier « normal », il suffit de supprimer ce sous-dossier **.git**

Rythme des enregistrements de l'historique

- Un enregistrement dans l'historique n'est pas déclenché automatiquement à chaque changement.
 - C'est l'utilisateur qui décide **quand les modifications doivent être enregistrées** dans la base de l'historique.
- ➔ Cela évite d'avoir une base de données trop lourde (une sauvegarde à chaque nouveau caractère saisie...)
- ➔ Mais l'idée est surtout que chaque enregistrement réalisé doit normalement correspondre à un état « stable » (l'ensemble des modifications en cours sont terminées).
- ➔ On doit pouvoir arriver sur un état stable si on restaure le projet sur un ancien point enregistré dans la base.

Périmètre des éléments enregistrés

- Pour faire un enregistrement, on n'est pas forcé de demander que tous les fichiers du projet soient concernés.
 - C'est l'utilisateur qui décide **quelles sont les fichiers à traiter** lors d'un enregistrement dans la base.
- ➔ Cela permet de solliciter un enregistrement dès qu'on a fini de modifier un fichier, même si d'autres fichiers ne sont pas encore « stabilisés ». **Mais**, si ce n'est pas utile, on garde la possibilité de demander que tous les fichiers soient traités.
- ➔ C'est une flexibilité importante, surtout quand de nombreux fichiers doivent être modifié dans une même phase de travail, et que finaliser cette phase prend du temps.

L'INDEX (*Staging Area*) → *le quoi*

- Le choix du périmètre de l'enregistrement est réalisé grâce à l'**INDEX**. Il s'agit d'une liste, interprétée comme **une salle d'attente virtuelle**, contenant les éléments choisis pour le prochain enregistrement.
- Les éléments que l'on place dans cette liste sont **les fichiers dont on souhaite la prise en compte lors du prochain enregistrement** dans l'historique (quand on choisira de déclenché ce dernier).
- On « rempli » l'**INDEX** progressivement à l'aide de la commande :
- Ou pour ajouter tous les fichiers qui ont été modifiés :

```
> git add fichier_a_ajouter
```

```
> git add .
```

Les **commit** → *le quand*

- Le moment où doit être déclenché un nouvel enregistrement dans l'historique s'appelle : **commit**
- Quand l'utilisateur déclenche cette action, l'outil analyse l'ensemble des éléments de l'**INDEX** pour les comparer à leur dernier état sauvegardé, afin d'enregistrer les modifications qu'ils ont subies.
- Cette demande d'enregistrement **doit être associé à un texte d'information** qui va être associé à l'enregistrement pour expliquer *l'objectif* (la raison, ou simplement un étiquetage) des changements.
- La commande à taper pour cette action est :

```
> git commit -m "texte d'information..."
```

La commande: **git status**

- Il faut donc toujours au moins 2 commandes pour engager un commit (**git add ... + git commit ...**). Et on peut parfois être un peu perdu en terme de visibilité sur les éléments qui ont été modifié depuis le dernier enregistrement.

➔ Pour aider à s'y retrouver il y a la commande :

> **git status**

- Son action est d'analyser l'état du dépôt, puis d'indiquer les changements qui ne sont pas encore dans l'**INDEX**, et si un **INDEX** est déjà présent et qu'il attend un **commit**.

➔ Son usage permet donc de savoir quoi faire à tout moment.

La commande: **git log**

- Pour avoir un visuel sur l'historique et afficher la liste des commits (avec leur texte d'information), on peut taper la commande :

```
> git log
```

➔ Notez que si le dépôt vient d'être créé, et qu'il n'y a pas encore eu de commit, cette commande va afficher une erreur.

➔ Cette commande indique l'**id** correspondant à chaque commit. Il s'agit d'un hash (une chaîne hexadécimale de 7 ou 40 caractères).

Le fichier .gitignore

- Des projets peuvent parfois contenir des fichiers pour lesquels la gestion de l'historique des modifications **n'est pas nécessaire**, voire même non souhaité.
 - Il peut s'agir par exemple des fichiers compilés **.o** dans un projet de programmation C/C++ (car on les génère depuis un code source, et c'est lui qui est déterminant).
 - On peut facilement configurer le dépôt pour exclure ces fichiers lors des traitements.
- ➔ il suffit de créer un fichier texte de nom **.gitignore** à la racine du dépôt, et d'écrire sur chaque ligne le nom de chacun des fichiers à ignorer (on peut utiliser les caractères jokers ***** et **?** Pour cibler plusieurs fichiers) .

⚠ les fichiers qui ont déjà fait l'objet d'un commit, sont placés en cache et ne seront pas concernés par cette exclusion.

➔ **.gitignore** n'influe que les fichiers non déjà en cache

➔ Pour purger les fichiers ignorés qui sont en cache, il faut taper :

> git rm --cached

4 – Les branches

Bienvenue dans le multivers !

branch

switch

merge

des lignes d'évolution parallèle

- Avec **Git**, à tout moment, il est possible de créer une « copie virtuelle » du projet qui pourra suivre sa propre ligne d'évolution.
 - ➔ Les modifications réalisées sur cette « copie » n'affecteront pas le projet « original ».
- Une telle « copie virtuelle » est appelée : une **branche**
- On peut créer autant de branche que l'on veut dans un dépôt, mais une seule sera active à un instant t.
- Chaque branche identifiée par **un nom**.
- Le projet « original » est lui-même une branche, c'est la branche initiale. Elle porte généralement le nom **main** (préférence moderne) ou **master** (ancienne mode).

Afficher la liste des branches existantes

- Pour afficher la liste de toutes les branches existantes dans un dépôt il faut taper : `> git branch`
- Au départ, seule la branche **main** (ou **master**) existe. Mais notez qu'**elle ne sera affichée** par cette commande que si **au moins un commit** a déjà été réalisé.
- La branche actuellement active est appelée la **branche courante**. Dans la liste affichée **elle est signalée par une étoile *** devant son nom.
- Toutes les actions réalisées sur le dépôt (**add, commit, status, log,...**) ne vont s'appliquer que sur la branche courante.
- Le plus souvent, les branches sont des alternatives temporaires du projet. Elles ont vocation soit à être abandonnée, soit à être validée et de « fusionner » avec la branche initiale.

Créer une nouvelle branche

- Pour créer une nouvelle branche on utilise la commande :

```
> git branch nom_de_la_nouvellebranche
```

- A sa création, l'état du dépôt (contenu et historique du projet) sera identique en tout point à celui de la branche courante (c'est une copie, un *snapshot*).
- La création d'une nouvelle branche peut être motivée pour diverses raisons :
 - Plusieurs personnes travaillent sur le projet, et on veut pouvoir opérer une série de modifications sans gêner leurs activités.
 - On souhaite expérimenter de possibles évolutions avant de véritablement retenir l'une d'entre elle.
 - On souhaite maintenir le projet dans un état stable, chaque nouveau chantier est réalisé sur une branche secondaire.
 - ...

Modifier le nom d'une branche

- On peut parfois vouloir changer le nom d'une branche (erreur sur le nom, changer le nom master pour le nom main,...) pour cela :

➔ Si la branche à renommer est la branche courante :

```
> git branch -m nouveau_nom
```

➔ Si c'est une autre branche :

```
> git branch -m ancien_nom nouveau_nom
```

Changer de branche

- Pour changer la branche courante, et basculer sur une autre branche on doit écrire :

```
> git switch nom_de_la_branche
```

- A noter que cette commande peut aussi s'écrire (ancienne façon) :

```
> git checkout nom_de_la_branche
```

- Après cette bascule, l'état du dépôt est modifié afin de correspondre à celui de la nouvelle branche courante.
- On peut aussi créer une nouvelle branche et basculer en même temps sur celle-ci :

```
> git switch -c nom_de_la_nouvelle_branche
```

Ou encore :

```
> git checkout -b nom_de_la_nouvelle_branche
```

Fusionner 2 branches

- Quand l'évolution du projet sur une branche arrive dans un état jugée *satisfaisant*, on peut décider de retenir (valider) l'ensemble des changements réalisés sur cette branche et de les appliquer sur la branche initiale. On réalise alors ce qu'on appelle **une fusion de branche**.
- Pour fusionner une branche sur la branche courante (il peut donc s'agir en réalité d'une autre branche que la branche initiale), on utilise la commande :

```
> git merge branche_a_appliquer
```

- ➔ Cette opération va appliquer tous les **commit** de la branche indiquée, sur la branche courante.
- ➔ La branche indiquée n'est pas modifiée, seule la branche courante est impactée.

Fusion == intégrer les changements

- Attention au fait qu'**une fusion n'est pas une copie !**
- Quand on fusionne une branche **B** sur la branche courante, ou plus généralement sur une branche **A** (celle qui reçoit la fusion) :
 - ➔ la branche **B** (source) ne subit aucun changement.
 - ➔ la branche **A** (cible) reçoit toutes les modifications connus dans l'historique de **B** au moment de la fusion.
 - ➔ Mais **A** peut comporter des changements non connus de **B** avant la fusion. Si ces changements n'occasionnent pas de conflit de fusion, ils resteront intacts dans **A**. Ainsi, **A** et **B** seront différents à ce niveau après la fusion.

Pour que A et B deviennent identique, il faut fusionner **B** sur **A** (et résoudre les conflits éventuels) **puis fusionner A sur B**.

Afficher le graphe des branches

- Après la création de plusieurs branches, et l'application de plusieurs fusions, l'ensemble des commit du projet forment globalement un graphe.
- Dans ce graphe, les branches constituent les différentes lignes reliant les commit entre eux.
- Les croisements de ces lignes représentent les points sur lesquelles les fusions de branches ont été réalisées.
- On peut obtenir une représentation visuelle de ce graphe en tapant la commande :

```
> git log --oneline --graph --all
```

5 – Les conflits de fusion

Des modifications qui divergent
sur les 2 branches à fusionner

restore

merge --abort

Conflit de fusion (*merge conflict*)

- Une opération de fusion peut conduire à un conflit quand la même portion d'un fichier a subi des modifications différentes dans les deux branches impliquées.
➔ Dans cette situation, **Git** n'est pas capable de déterminer laquelle des deux versions doit être conservée.
- La fusion est quand même appliquée, mais la branche courante est marquée comme étant dans un état spécial appelé : "En cours de fusion" (**Merging state**).
- Dans cet état, Git bloque toutes opérations sur le dépôt (plus de commit, de changement de branche, ...).
- Pour lever ce blocage, l'utilisateur doit inspecter le, ou les éléments en conflits, et résoudre chacun d'eux. On appelle cette phase : **la résolution des conflits**.

Les marqueurs de conflit (le cas texte)

- Après une fusion conduisant à un conflit, Git marque les portions de fichiers problématique de la façon suivante :

```
<<<<<< HEAD
```

partie présente dans la branche courante

```
=====
```

contenu de cette partie dans la branche à appliquer

```
>>>>>> branche_a_appliquer
```

- Un tel marquage est appelé : **marqueur de conflit**
- L'utilisateur doit analyser chacun de ces marqueurs, pour prendre une décision sur ce qu'il convient de faire.

→ Ce marquage n'est opéré que sur les fichiers dont le contenu est au format texte !

Résolution des conflits *(le cas texte)*

Pour résoudre les conflits l'utilisateur peut :

- Inspecter chaque marqueur de conflit et choisir : soit de conserver l'une des deux versions du contenu, soit de conserver les deux versions, soit aucunes d'elles.

➔ Il acte sa décision en supprimant les 3 lignes de marquages :

```
<<<<<< HEAD  
=====  
>>>>>> branche_a_appliquer
```

- C'est la disparition de l'ensemble des marqueurs de conflit qui « débloque » la branche courante. L'utilisateur peut alors clore l'opération en validant tous les changements réalisés via un commit :

```
> git add .  
> git commit -m "texte d'information..."
```

Les conflits sur les fichiers binaires

- **Git** fonctionne de façon particulière pour le cas où des conflits concernent des fichiers binaires (*images, pdf, zip, ...*).
 - ➔ En effet, sur de tels fichiers la notion de « ligne » n'existe pas, et le marquage par des zones par du texte rendrait inexploitable l'ensemble du fichier.
- Quand Git détecte un conflit sur ces fichiers, il les déclare en « **conflit binaire** ».
 - ➔ L'indication de la liste des fichiers en conflit (donc dans cet état) est donnée par le « constat de conflit » qui est affiché à la suite de la commande > **git merge ...**
 - ➔ Ce constat de conflit pourra également être consulté à tout moment par la suite via la commande > **git status**

Résolution des conflits des fichiers binaires

- La résolution des conflits sur les fichiers binaires nécessite que, **pour chaque fichier**, l'utilisateur choisisse quelle version conserver entre : celle de la branche courante, et celle de la branche à appliquer.
- Le choix de la version à conserver se fait via l'une de ces 2 commandes suivante :

```
> git restore --ours fichier_concerné
```

➔ Pour conserver la version de la branche courante

```
> git restore --theirs fichier_concerné
```

➔ Pour conserver la version de la branche à appliquer

NB: l'ancienne commande (tjs valable) ici est **checkout** à la place des **restore**

Puis on doit acter toutes ces décisions dans un **commit**.

Annuler une fusion en conflit

- Dans le cas où la résolution de conflit pause des problèmes trop complexes, on peut choisir d'annuler l'ensemble de la fusion avec la commande :

```
> git merge --abort
```

- ➔ La branche courante repasse alors dans l'état où elle se trouvait avant la demande de fusion.
- ➔ Cette annulation est possible tant que la phase de résolution de conflit n'a pas été finalisée.

6 – Dépôt distant

Généralement un serveur GitHub

remote -v remote -add remote show

branch -u branche -vv

Le dépôt distant (*remote repository*)

- C'est un dépôt git conservée sur un serveur (le plus souvent dans le cloud).
- Le serveur rend le contenu de ce dépôt accessible via le réseau Internet. En se connectant à ce serveur, on peut :
 - soit demander à obtenir une copie locale du dépôt : si on se met à travailler sur une nouvelle machine, ou qu'on est un nouveau membre du projet → c'est l'action **clone**
 - soit demander à actualiser sa copie locale, plus précisément une branche de son dépôt local → c'est l'action **pull**
 - soit envoyer une nouvelle version du projet (l'ensemble des changements réalisés sur une branche).
→ c'est l'action **push**

Création d'un dépôt distant

- Pour créer un dépôt distant il faut détenir un compte utilisateur sur une plateforme qui propose ce type de service (nous choisirons **GitHub**).
 - La création et la configuration se passe le plus souvent sur le site web du serveur. Dans la section 6, nous verrons comment réaliser cette création sur un serveur **GitHub**
 - Une fois crée, **une url spécifique** permettra de designer ce dépôt distant de façon universel.
- ➔ Cette url contient **l'adresse du serveur** qui héberge le dépôt, et **la chaine uri désignant le dépôt** sur ce serveur.
- ➔ Cette url sera utilisée dans les commandes qui vont permettre d'associer un dépôt local à un dépôt distant.

Les plateformes d'hébergement Git

Il existe plusieurs plateformes d'hébergement de dépôt distant.

- **GitHub :**

- appartient à Microsoft
- service en ligne, gratuit pour les particuliers

- **GitLab :**

- intègre des outils de gestion de projet avancés (*DevOps*)
- service en ligne, mais existe aussi en auto-héberger

- **Bitbucket :**

- appartient à Atlassian
- service en ligne spécialisé pour les entreprises

- **Forgejo / Gitea :**

- logiciels libres
- auto-héberger
- s'installe facilement sur des petits serveurs (*Raspberry pi*)

Remote : liaison dépôt local => dépôt distant

- On appelle **remote** une liaison établie entre un dépôt local et un dépôt distant.
- En réalité, on peut lier plusieurs dépôts distants à un même dépôt local. Ainsi, un dépôt local peut avoir plusieurs **remotes**.
- Chaque dépôt distant ajouté à un dépôt local sera associé à un nom. Ce nom de dépôt distant est appelé le « **nom de remote** » (*remote name*).
- Le plus souvent, on aura qu'un seul dépôt distant associé au dépôt local, et son nom de remote sera **origin**.
- Pour afficher la liste des dépôts distants actuellement associés au dépôt local on tape : `> git remote -v`

Ajouter et modifier des remotes

- Pour associer un nouveau dépôt distant on tape :

```
> git remote add nom_de_remote url_du_depot_distant
```

➔ Par convention, le premier (et souvent le seul) dépôt distant qu'on associe porte le nom **origin**, on tapera donc :

```
> git remote add origin url_du_depot_distant
```

- Pour changer le nom de remote d'un dépôt :

```
> git remote rename ancien_nom nouveau_nom
```

- Pour modifier l'url (donc le dépôt distant) d'un remote :

```
> git remote set-url nom_de_remote nouvelle_url
```

- Pour consulter la liste des branches d'un **remote** :

Un seul :

```
> git fetch nom_remote  
> git remote show nom_remote
```

Tous :

```
> git fetch  
> git branch -r
```

Tracking : association branche locale ⇔ branche distante

- L'intérêt d'une liaison entre un dépôt local et un dépôt distant (un remote) s'établit **au niveau des branches**.
- Un remote va permettre, à tout moment, **d'associer une branche du dépôt local avec l'une des branches du dépôt distant**.
- On appelle cette association de branche : un **tracking**.
 - ➔ On dira par exemple : « la branche locale XXX **est en tracking** sur la branche distante YYY »
- Dans un tracking, la branche distante ciblée est appelée **l'upstream**.
 - ➔ Si on tape une commande nécessitant un tracking sur une branche qui n'en a pas encore, le système dira :
« *fatal: The current branch has no upstream branch* »

Intérêt des trackings

- Un tracking **va permettre de synchroniser le contenu des branches entres elles** via les commandes :
 - **pull** : pour fusionner la branche distante sur la branche locale
 - **push** : pour fusionner la branche locale sur la branche distante.
- Il n'est pas obligatoire de configurer un tracking pour toutes les branches du dépôt local. Certaines branches peuvent servir à des sessions de modification locale qui n'ont pas vocation à être diffusées, ou partagées.
- Les branches locales d'un même dépôt, peuvent cibler par tracking des dépôts distants différents (si plusieurs remotes ont été configurés).

Mise en place des trackings

- Aucune association entre les branches n'est établie au moment de l'ajout d'un serveur distant (d'un remote).
- Il faut explicitement taper une commande pour mettre en place chacun des trackings nécessaires.
 - ➔ ces commandes sont décrites ci-après
- Si une commande de synchronisation distante (pull/push) est déclenchée sur une branche locale qui n'a pas de tracking, **cela provoque une erreur**.
- Via une option spécifique (-u), il est aussi possible de configurer le tracking (l'ajouter ou le modifier) au moment même où on déclenche la commande de synchronisation distante **push** (mais pas **pull**).
 - ➔ nous verrons comment quand nous détaillerons cette commande **pull** un peu plus loin.

Ajouter et modifier des trackings

- Pour ajouter/modifier le tracking **de la branche locale courante** :

```
> git branch --set-upstream-to=nom_remote/branche_distante
```

Ou plus simplement :

```
> git branch -u nom_remote/branche_distante
```

- Pour faire ajouter/modifier le tracking **de n'importe quelle branche locale** :

```
> git branch --set-upstream-to=nom_remote/branche_distante branche_locale
```

Ou plus simplement :

```
> git branch -u nom_remote/branche_distante branche_locale
```

Consulter la liste des trackings

- Afficher la liste des trackings existants il faut taper :

```
> git branch -vv
```

- ➔ Pour chacune des branches locales, le tracking associé (s'il existe) sera affiché entre crochet [...]
- ➔ Dans ces crochets on a les informations suivantes :
 - L'indication de **l'upstream** (nom du remote + nom de la branche).
 - si la branche locale a *X* commits de plus que **l'upstream**, il y aura la mention : **ahead X**
 - si **l'upstream** a *X* commits de plus que la branche locale, il y aura la mention : **behind X**

7 – Les commandes de synchronisation distante

clone

fetch

pull

push

push -u

push -f

L'action **clone** → créer une copie locale

- Quand le **dépôt distant existe**, et qu'on n'a **pas encore créée un dépôt local** sur sa machine, on peut procéder avec une simple commande :

```
> git clone url_du_depot_distant
```

- Avec cette commande, l'outil **Git** va se connecter au serveur distant, et demander **une copie intégrale** du dossier du projet.
- ➔ Le dossier du projet va donc être **créée localement** sur le disque, dans le répertoire dans lequel la commande a été tapée.
- ➔ Ce dossier va constituer le dépôt local pour ce projet, et il sera **déjà configuré comme étant connecté (remote) au dépôt distant** correspondant à l'url indiquée.

Les remotes et les trackings produits par l'action **clone**

- Lorsqu'on clone un dépôt distant, pour rappel avec :

```
> git clone url_du_depot_distant
```

Git va produire le dépôt local et ajouter automatiquement le **remote** vers ce dépôt distant.

➔ Il n'est donc pas utile de faire l'action : **remote add**

Git va également créer une branche locale courante identique à **celle défini par défaut pour le dépôt distant**. Cette dernière est appelée la **branche par défaut**, en général son nom est **main** (ou *master*).

Enfin, **Git** va **fixer un tracking** de cette branche locale avec la branche par défaut du dépôt distant.

➔ Il n'est donc pas non plus utile de produire ici l'action : **branch --set-upstream-to= ... ou -u**

Etats des branches suite à l'action **clone**

➔ **clone** copie l'intégralité du dépôt distant

Mais : seule **la branche par défaut est activée**, c'est-à-dire qu'elle donnera lieu à une branche identique créée localement (**donc visible**). Avec un tracking établie automatiquement entre ces deux branches.

➔ **Les autres branches** sont présentes localement mais elles sont mises dans une sorte de « réserve », et **ne sont pas visible**. Pour les voir il faut taper : **branch -a** *(all)*

➔ Une branche en réserve deviendra active (visible) la première fois qu'on basculera dessus avec la commande **switch**. Elle donnera alors automatiquement lieu à une branche locale identique (copie), avec un tracking établie automatiquement avec la branche distante de même nom.

L'action **fetch** → mettre à jour le cache (.git)

- Le dépôt local ne maintient pas une connaissance permanente des remotes : si des éléments d'un dépôt distant sont modifiés, le dépôt local ne sera pas informé.

→ La conséquence est que la commande **> git status** ne nous dira pas que notre contenu n'est pas à jour.

- Pour avoir **une information pertinente** avec cette commande **status**, il faut toujours commencer par faire : **> git fetch**

→ Son action est de rapatrier toutes les données en cache concernant **l'upstream** de la branche courante. Par la suite, le système local sera donc en mesure de comparer l'état de la branche courante par rapport à celui de son **upstream**.

La commande **> git fetch *nom_remote*** fait la même chose avec toutes les branches du remote indiqué. De plus, il ajoute toutes ces branches dans la « réserve » (comme le fait **clone**). Ce qui permet de copier ces branches avec une commande **switch**.

L'action **pull** → mettre à jour la copie locale

- Quand un tracking **existe pour la branche courante**, et que l'on souhaite récupérer localement les modifications connus (le dernier commit) de sa branche **upstream** (donc sur le dépôt distant), on tape :

> **git pull**
- Cette action va conduire à **fusionner** la branche distante (l'upstream), avec la branche locale courante.
 - ➔ s'il n'y a aucun conflit de fusion, la branche locale va recevoir toutes les modifications connues sur le dépôt distant. On aura ainsi rapatrié localement les changements réalisés par les autres membres du projet sur cet upstream.
 - ➔ s'il y a un, ou plusieurs **conflits de fusion** il faudra résoudre ces conflits (ou annuler la fusion) afin de pouvoir continuer à travailler sur le dépôt local.

Créer/modifier le tracking au moment du **pull** ?

- L'usage de la commande **pull** nécessite qu'un **tracking existe déjà** pour la branche courante.

➔ Il n'est donc **pas possible de créer, ou de modifier, le tracking** au moment du **pull**. Il faut le créer avant.

- Néanmoins il est possible (mais l'utilité est discutable) de réaliser un **pull** d'une branche distante, sans créer (ou modifier) le tracking.

➔ Cela s'appelle faire **un pull explicite** :

```
> git pull nom_remote branche_distante
```

Cela réalise une fusion de la branche ciblée sur la branche courante. **Aucun tracking n'est créé ici**, et si la branche en a déjà un, celui-ci restera inchangé.

Créer une copie locale d'une branche distante

- On peut parfois vouloir copier une branche distante qui n'existe pas encore sur le dépôt local.

➔ Si le dépôt local a été produit par une commande **clone** (celle-ci ne récupère que la branche par défaut), et que la branche distante souhaitée est sur ce dépôt distant. Cette branche distante est déjà dans la « réserve », il suffit donc de basculer dessus (**switch**) pour la copier localement.

➔ S'il s'agit d'une branche distante d'un autre dépôt, il faut que le dépôt soit ajouté en tant que **remote**, puis faire :

```
> git fetch nom_remote
```

```
> git switch branche_distante
```

← met à jours la « réserve » .

Cela crée une branche locale de même nom, contenant une copie du contenu de la branche distante. Cette nouvelle branche locale deviendra la branche courante, et elle disposera automatiquement d'un tracking vers la branche distante ciblée.

L'action **push** → mettre à jour le dépôt distant

- Si on souhaite envoyer les derniers changements locaux (les derniers **commit**) sur le dépôt distant, on utilise la commande :

> **git push**
- Cette action va produire **une fusion de la branche courante avec son upstream** (la branche distante).
 - ➔ s'il n'y a aucun conflit de fusion, la branche distante recevra toutes modifications que l'on a réalisées dans la branche locale. On aura ainsi mis ces changements à la disposition des autres membres du projet.
 - ➔ s'il y a un, ou plusieurs **conflits de fusion** alors **l'opération sera automatiquement annulée**. Il faut alors commencer par faire un **pull** de sorte à résoudre les conflits localement, puis refaire un **push** pour transmettre les modifications locales avec les conflits résolus.

Créer/modifier le tracking au moment du **push**

Sans argument push opère sur le tracking de la branche courante.

- Mais on peut choisir de cibler une branche distante spécifique (soit pour modifier le tracking existant, soit pour en créer un si il en existe pas déjà). Il faut pour cela utiliser la syntaxe :

```
> git push -u nom_remote branche_distante
```

- On peut aussi réaliser le push **sans ajouter (ou changer)** le tracking actuellement configuré, il suffit de retirer **-u** :

```
> git push nom_remote branche_distante
```

➔ **Attention** : dans les deux cas, si la branche distance n'existe pas, cela va conduire à la **créer automatiquement** sur le dépôt distant. Dans ce dernier cas, il ne peut pas y avoir de conflit possible. Et si la branche existe, et que la fusion produit un conflit, il faudra résoudre le conflit avec un **pull** avant de relancer ce **push**.

Forcer un **push** ou un **pull** *(options à éviter)*

- On peut passer outre les conflits et forcer le dépôt distant à accepter une action **push**. Pour cela il faut taper la commande :

```
> git push -f
```

➔ L'**upstream** aura alors le même contenu que la branche courante.

- Pour forcer un **pull**, il faut fonctionner différemment (l'option **-f** n'existe pas) :

```
> git fetch nom_remote
```

```
> git reset --hard nom_remote/branche_distante
```

➔ La branche courante aura le même contenu que celui de la branche distante ciblée dans la commande **reset**.

Notez que ces commandes remplacent complètement le contenu de la branche en ignorant les conflits potentiels. Aussi, les changements non synchronisés sur la branche qui subit la fusion seront tous perdus.

8 – Les opérations sur l'historique

Voyager dans le temps !

checkout

reset --hard

checkout -b

Observer le passé

- On peut vouloir examiner précisément le contenu du projet sur un point précis de l'historique, c'est-à-dire : sur la situation du projet juste après un certain commit.
- Pour cela, il faut commencer par relever **l'id du commit** à observer via la commande : `> git log`
- Ensuite, on bascule sur ce point temporel avec la commande :

```
> git checkout id_du_commit
```

➔ Après cette commande, **l'ensemble du contenu du projet est restauré** sur sa version juste après ce commit. On peut donc consulter les fichiers pour observer cette situation.

Attention : avec cette commande on bascule sur une branche « fantôme » ➔ on est dans l'état « **detached HEAD** ».

➔ **On ne doit faire aucune modification, ni aucune action git**

- Quand on a fini cette consultation, on revient à la situation normale en tapant : `> git checkout branche_locale`

Revenir (définitivement) dans le passé

- On peut aussi vouloir restaurer le contenu du projet sur un point précis de l'historique, c'est-à-dire : **annuler tous les changements** réalisés après un certain commit.
- On commence par relever **l'id du commit** : `> git log`
- Ensuite, on lance la restauration du projet sur ce point temporel avec la commande :

```
> git reset --hard id_du_commit
```

➔ Après cette commande, **l'ensemble du contenu du projet est restauré** sur sa version juste après ce commit, et **tous les commits réalisés après sont détruits**.

Attention : cette commande est irréversible ! C'est une téléportation définitive dans le passé.

➔ La seule possibilité de « retour » est l'existence d'un remote avec lequel vous allez pouvoir **forcer un pull** sur l'une de ces branches.

Repartir (sereinement) du passé

- Pour éviter la criticité d'un retour définitif, il est préférable de produire la restauration en initiant celle-ci sur une nouvelle branche.
- On commence par relever **l'id du commit** : `> git log`
- Ensuite, on bascule sur ce point temporel, exactement comme on le fait pour aller l'observer : `> git checkout id_du_commit`
- On crée alors une nouvelle branche depuis ce commit :
`> git checkout -b nom_de_la_nouvelle_branche`

➔ Cette action va permettre de sortir de l'état « **detached HEAD** » et de basculer sur la branche nouvellement créée. Cette branche aura un contenu correspondant à celui produit par le commit ciblé.

➔ L'avantage ici est que rien n'est perdu : le « futur » du commit ciblé est conservé dans l'historique. On peut y revenir en basculant sur la branche depuis laquelle on a initié toute cette démarche.

9 – Bilan des commandes de diagnostique

➔ À consommer sans modération !

Diagnostic régulier

> git status

Etat des lieux du dépôt.

➔ Permet de voir s'il faut faire : des **add**, un **commit**, un **push** ou un **pull**.

> git log

Historique complet des **commits**, et consulter en particulier leur id respectif.

> git branch

Liste des **branches locales**.

> git fetch

> git branch -vv

Afficher la liste des **trackings** et de la situation avance/retard des branches.

➔ Permet de voir si des **push** ou des **pull** sont nécessaires.

Diagnostic occasionnel

> **git config --list**

Vérifier son **identité Git**.

> **git log --oneline --graph --all**

Voir le **graphe des branches** (les points de création et de fusion).

> **git remote -v**

Afficher la liste des **remotes**.

> **git fetch** *nom_remote*

> **git remote show** *nom_remote*

Afficher la liste des **branches d'un remote** en particulier.

> **git fetch**

> **git branch -r**

Afficher la liste des **branches de tous les remotes**.

- *Fin* -