
git

git

Dans ce cours, le code et les exercices seront diffusés via git. Git est un logiciel de gestion de version (VCS).
Il apporte les fonctionnalités suivantes lorsqu'un groupe travaille sur un même projet :

1. Synchroniser le travail entre les personnes qui programment
2. Enregistrer l'évolution du code au cours du temps
3. Stocker le code en lieu sûr

Git stocke ses données dans le répertoire de votre projet, dans le dossier *.git*, de manière indépendante pour chaque projet.
Git se contrôle par ligne de commande, le contenu de *.git* n'est jamais modifié manuellement.

Références :

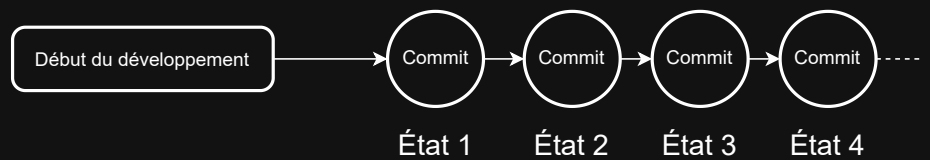
- Site officiel : https://git-scm.com/
- Téléchargement : https://git-scm.com/downloads
- Livre : https://git-scm.com/book/en/v2

git est un incontournable du développement logiciel. C'est un programme open source utilisé partout dans le monde.
Nous allons voir dans ce chapitre un aperçu des fondamentaux. Il existe de nombreuses fonctions que nous n'allons pas aborder.
Il existe plusieurs alternatives à git (Perforce, Mercurial, etc) mais git est le plus généralement utilisé, particulièrement dans le monde open-source.
Git gère tous les types de données textuelles, pas seulement du code. Des livres, des documentations, des configurations, des notes, etc peuvent très bien être des projets gérés par git. Par exemple, ces slides sont un projet git stocké sur GitHub. En revanche git n'est pas fait pour gérer des fichiers compressés ou binaires, car ces derniers changent beaucoup d'une version à une autre.

git

Les commits

La notion fondamentale dans git est le *commit*.
Il s'agit d'un instantané de votre projet, un "point de sauvegarde" qui enregistre son état.
Git permet de revenir, de comparer, de mélanger différents commits.
Ce qui se passe entre les commit n'est pas sauvegardé.



Quand un commit est enregistré, git attribue un identifiant unique au commit et enregistre sa différence par rapport au commit précédent.
Tout est placé de manière transparente dans le dossier .git de votre projet.

git

Les commits

Dans un projet, pour indiquer à git qu'un fichier doit être *tracké* (pris en compte), la commande est :

```
git add <chemin_vers_le_fichier>
```

Pour faire un commit, la commande est :

```
git commit -am "Message"
```

Git créé alors un nouveau commit

- L'option *-a* indique de prendre tous les fichiers trackés modifiés
- L'option *-m* indique le message du commit. Il s'agit d'un texte décrivant ce qui a été modifié dans ce commit.

Si l'option *-m* n'est pas précisée, git ouvrira un éditeur de texte pour entrer le message de commit

Par défaut, git n'enregistre pas l'ensemble des fichiers. Il faut lui indiquer quels fichiers sont importants avec la commande *git add*

La commande *git status* permet de voir l'état actuel d'un projet. Elle indiquera les fichiers modifiés depuis le dernier commit, et s'ils sont trackés ou non.
Les messages de commit doivent être courts et le plus descriptif possible.
Mauvaise exemple : "*J'ai modifié plusieurs fichiers de la configuration globale pour la prochaine mise à jour, et corrigé plusieurs bug divers mais pas très importants*" (C'est long et très peu précis)
Bon exemple : "*Ajoute la couche bâtiments de OpenStreetMap*" (Court et précis)
Il est également possible de ne commit que certains fichiers modifiés, quand on ne précise pas l'option *-a*, mais ceci sort du cadre de ce cours.

git

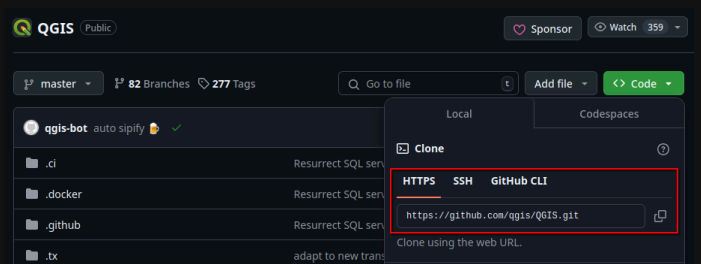
Le clonage

Pour travailler sur un projet existant, il faut d'abord le copier en local sur sa machine. On appelle cela le *clonage*. La commande est :

```
git clone <url_du_projet_distant>
```

Git créé alors un dossier contenant tout le projet.

Les projets sont stockés dans un dépôt (*repository*) distant. Il existe plusieurs fournisseurs de stockage : GitHub, GitLab, ou des stockages auto-hébergés tels que Gitea.

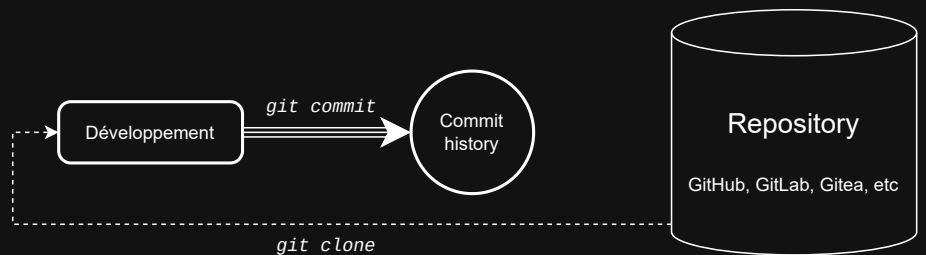


On confond souvent *git* et *GitHub*, mais GitHub n'est "que" un système de dépôt pour projets git. Le logiciel est bel et bien **git**.
Aujourd'hui les fournisseurs de stockage proposent de nombreuses options en plus du simple stockage. GitHub permet d'executer des actions sur le code (déploiements, compilations, etc), de faire des analyses de sécurité, de tenir une liste des tâches, et bien d'autres fonctions.
GitHub affichera, sous la liste des fichiers et dossiers, le contenu du fichier "README.md" du dossier courant. C'est particulièrement pratique pour créer une page d'accueil de votre projet.

git

Workflow de base

Le workflow est donc le suivant :



Pour travailler sur un projet, on procède généralement de la manière suivante :

1. On clone un repository existant, contenant le code sur lequel on se base
2. On y apporte des modifications
3. On fait petit à petit des commits pour chaque modification apportée

A noter que les commits sont pour le moment stockés en local sur la machine où le projet à été cloné. Jusqu'ici il n'y a pas de synchronisation avec le repository.

git

Push & Pull

Après plusieurs commits, il est possible d'envoyer (*push*) les changements au repository distant:

```
git push
```

Cela enverra tous les commits sur le repository distant. Ce qui n'a pas été commit ne sera pas pris en compte. Pour faire un `git push`, il faut avoir le droit de push sur le repository distant.

Pour récupérer (*pull*) la dernière version du code en ligne, si celui-ci a été modifié par quelqu'un d'autre:

```
git pull
```

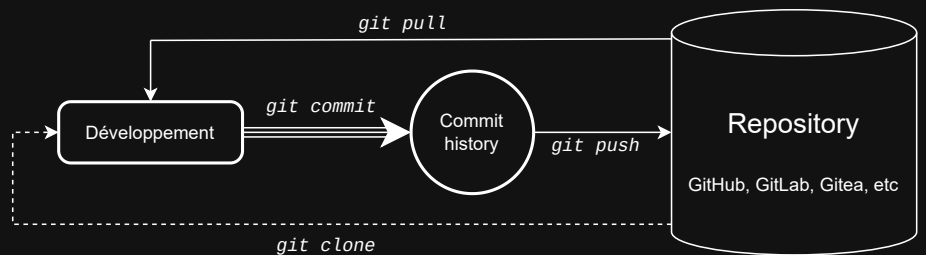
> Faites toujours un commit de vos changements avant un pull

Il est important de toujours *commit* avant un *pull*, sinon git se retrouvera avec votre version du code non sauvegardée, et une nouvelle version venue du repository distant. Si les modifications se chevauchent, Git ne saura pas quoi faire et refusera de *pull*.
Git est bien conçu et n'effacera/n'écrasera **jamais** de travail non sauvegardé, sauf lors de l'utilisation de commandes très explicites. Il en va de même pour les commits. Une fois un code commité, git n'effacera jamais le commit, sauf lors de l'utilisation de commandes très particulières.

git

clone & commit & push & pull

Le workflow est donc le suivant :

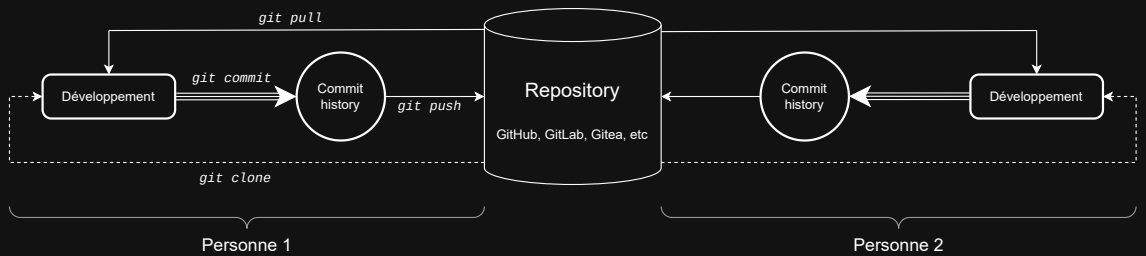


On retrouve ici les 4 commandes les plus utilisées avec git :

1. git clone
2. git commit
3. git push
4. git pull

git

En groupe



Les personnes travaillant sur un projet on chacune un clone du code et de l'historique des commits. Cela permet à tout le monde de travailler de son côté, le repository servant de point de synchronisation.
A ce stade, concernant les trois problèmes évoqués au début du cours :

- Stocker le code en lieu sûr : Il s'agit du repository distant, problème résolu
- Avoir un historique du code : Il s'agit des commits : problème résolu
- Travailler à plusieurs sur le même code : On comprends le principe, mais que se passe-t-il si la Personne 1 et la Personne 2 font des modifications chacune de leur côté en même temps ?

git

Les conflits

Supposons :

1. Alice et Bob clonent leur projet, chacun de son côté
2. Alice écrit `div{ color:blue; }`, puis fait un commit
3. Bob écrit `div{ color:green; }`, puis fait un commit
4. Bob push son code avec `git push`
5. Alice push son code avec `git push`

Que se passe-t-il chez Alice ?

```
! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/...'
```

Alice ne peut pas push, car elle ne possède pas la dernière version du code

Au moment d'executer le point 5, git effectue un push. Mais le code présent sur le repository est plus récent que le code que possède Alice (car le code du repository contient le commit de Bob).
Git ne va jamais écraser un changement, ce serait beaucoup trop dangereux, git refuse alors de faire le push. En pratique git indique toujours dans le message d'erreur ce qu'il est conseillé de faire. Voir slide suivante...

git

Les conflits

La solution : Alice doit d'abord faire un `git pull` pour récupérer la dernière version du code. A ce moment, git va mélanger (*merge*) les deux versions du code, celle d'Alice et celle de Bob.

Deux cas peuvent se produire :

1. Les modifications ne sont pas contradictoires, git parvient à faire automatiquement le merge
   - Alice possède alors une version du code mélangée
   - Il lui suffit de faire un `git commit` et un `git push` pour push sa nouvelle version.

git travaille par différence entre les fichiers. Si deux fichiers différents ont été modifiés, alors les modifications de chacun seront fusionnées. Si le même fichier a été modifié, et qu'il s'agit de lignes différentes, là aussi il y a fusion.
En revanche, si la même ligne d'un même fichier a été modifiée, alors les modifications sont "contradictoires", et git ne sait pas quelle version choisir.

git

Les conflits

La solution : Alice doit d'abord faire un `git pull` pour récupérer la dernière version du code. A ce moment, git va mélanger (merge) les deux versions du code, celle d'Alice et celle de Bob.

Deux cas peuvent se produire :

2. Il y a conflit

Lors du pull Alice recevra le message :

```
Auto-merging style.css
CONFLICT (content): Merge conflict in style.css
Automatic merge failed; fix conflicts and then commit the result.
```

Les conflits sont indiqués sous la forme :

```
<<<<<<< HEAD
div{ color:blue; }
=======
div{ color: green; }
>>>>>>> b6eeeaef7d4c17e8b7ad2b90968e2d17720ba319
```

Alice devra alors résoudre les conflits manuellement, puis `git commit` et `git push`

Git indique les conflit en commençant par la version distante du code ("incomming change") et ensuite la version locale du code ("current change"). HEAD indique quelle est la version distante. Dans ce cas il s'agit du tout dernier commit effectué dans le repository, appelé "HEAD". "b6eeeaef7d4c17e8b7ad2b90968e2d17720ba319" indique le hash du commit. Chaque commit dans git possède un identifiant unique appelé son hash. Ici il y a donc un conflit entre le HEAD et le commit b6eeeaef...

git

Astuces

La commande `git log` permet de voir l'historique des commits

La commande `git status` permet de voir la liste des fichiers modifiés depuis le dernier commit

Si un fichier nommé `.gitignore` est placé à la racine d'un projet, les dossier et fichiers listés à l'intérieur ne seront jamais trackés. Cela est très pratique pour directement exclure des fichiers et des dossiers entiers qu'on ne souhaite pas synchroniser. Il est possible d'utiliser des expressions génériques. Par exemple `*.txt` empêchera tous les fichiers avec l'extension `.txt` d'être trackés.

Ce chapitre est une introduction à git, il existe de nombreuses autres fonctions qui n'ont pas été mentionnées et que vous découvriez au travers des exercices et de la pratique.