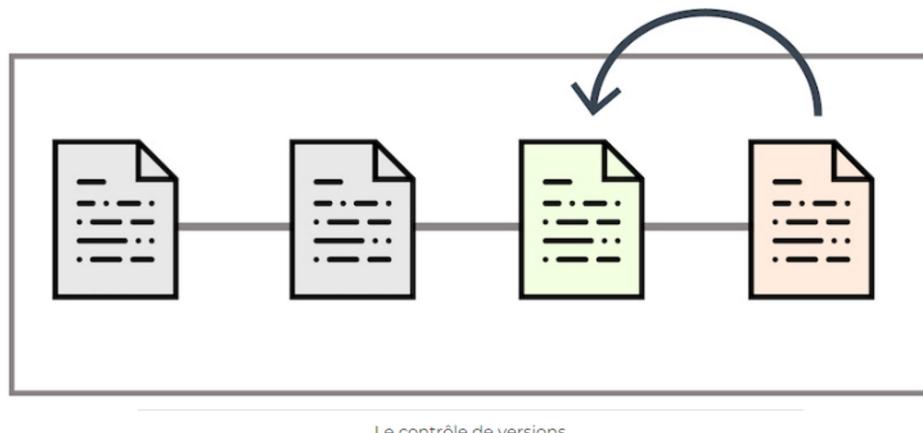


# Introduction à Git et GitHub

mardi 6 septembre 2022 15:18

Lien du cours : <https://openclassrooms.com/fr/courses/7162856-gerez-du-code-avec-git-et-github/7165703-decouvrez-la-magie-du-controle-de-versions>

## Définition de Git et des outils de contrôle de version



Un gestionnaire de versions est un programme qui permet aux développeurs de conserver un historique des modifications et des versions de tous leurs fichiers.

Le gestionnaire de versions permet de garder en mémoire :

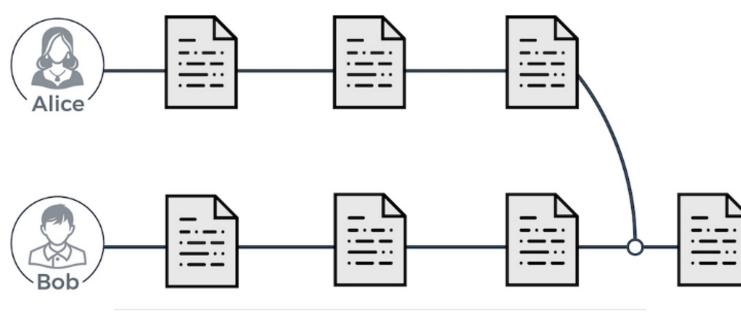
- chaque **modification** de chaque fichier
- pourquoi elle a eu lieu
- et par qui

Cet outil a donc trois grandes fonctionnalités :

- Revenir à une **version précédente** de votre code en cas de problème.
- Suivre l'**évolution** de votre code étape par étape.
- Travailler à plusieurs sans risquer de supprimer les **modifications** des autres collaborateurs.

C'est un programme qui a une structure décentralisée : Avec Git, l'historique complet du code n'est pas conservé dans un **unique emplacement**. Chaque copie du code effectué correspond à un **nouveau dépôt** dans lequel est conservé l'historique des modifications.

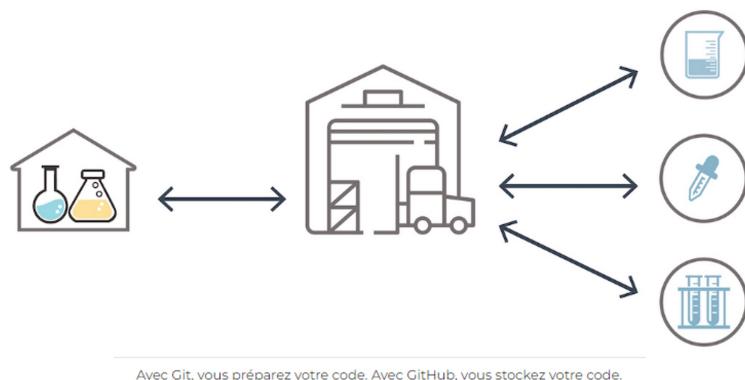
L'utilisation de Git permet la **modification** des fichiers, **envoyer et recevoir des mises à jour** à tout moment, **sans risquer d'écraser les modifications de l'autre**.



Alice et Bob ont initialisé Git. Ils travaillent chacun sur leur partie du code.  
Quand ils les regroupent, leurs versions précédentes sont archivées.

Définition Git et Github :

- **Git** : **Git** est un **gestionnaire de versions**. Il est utilisé pour créer un dépôt local et gérer les versions de fichiers.
- **GitHub** : **GitHub** est un **service en ligne qui va héberger des dépôts de différents projets**. Dans ce cas, on parle de dépôt distant puisqu'il n'est pas stocké sur en local.



En Résumé :

- **Un gestionnaire de versions** permet aux développeurs de **conserver un historique des modifications** et des **versions** de tous leurs fichiers.
- **Git** est un **gestionnaire de versions** tandis que **GitHub** est un **service en ligne qui héberge les dépôts Git**. On parle alors de **dépôt distant**.

### Saisir l'utilité des dépôts distants - GitHub

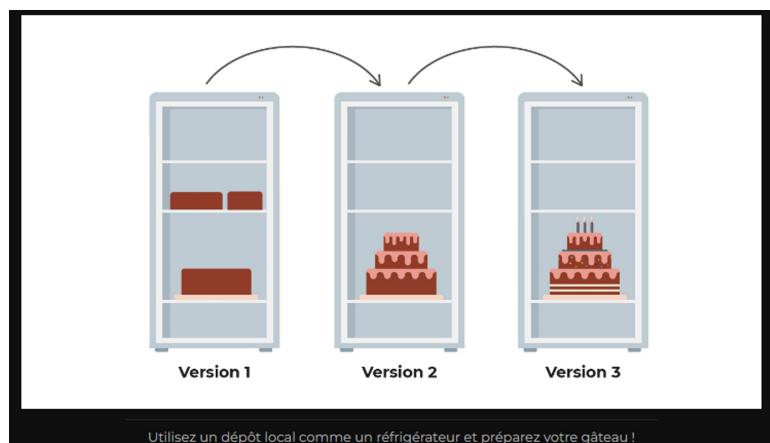
*Un dépôt est comme un dossier qui conserve un historique des versions et des modifications d'un projet. Il peut être local ou distant. Dans la documentation en ligne ou en milieu professionnel, on parle souvent de repository, qui est la traduction anglaise du terme "dépôt".*

Un dépôt local est un entrepôt virtuel de votre projet. Il vous permet d'**enregistrer les versions** de votre code et d'y accéder au besoin.

Pour illustrer cette idée, prenons l'image de la réalisation d'un gâteau. Pour faire un gâteau, vous allez réaliser les étapes suivantes :

- préparer la pâte du gâteau
- stocker cette pâte au réfrigérateur
- réaliser la crème et en garnir la pâte
- stocker le gâteau assemblé au réfrigérateur
- décorer votre gâteau
- remettre le gâteau au réfrigérateur.

Dans cet exemple, le réfrigérateur est comme un dépôt local : c'est l'endroit où vous stockez vos préparations au fur et à mesure.



Un dépôt local est utilisé de la même manière ! **On réalise une version**, que l'on va petit à petit **améliorer**. Ces versions sont stockées au fur et à mesure dans le dépôt local.

Le dépôt distant permet de **stocker les différentes versions** de votre code afin de garder un **historique délocalisé**, c'est-à-dire un **historique hébergé sur Internet ou sur un réseau**. Vous pouvez avoir **plusieurs dépôts distants** avec des **droits différents** (lecture seule, écriture, etc.).

Ainsi, les dépôts sont utiles si :

- Vous souhaitez **conserver un historique** de votre projet
- Vous **travaillez à plusieurs**
- Vous souhaitez **collaborer à des projets open source**
- Vous devez **retrouver par qui** a été faite chaque modification
- Vous voulez savoir **pourquoi** chaque modification a eu lieu.

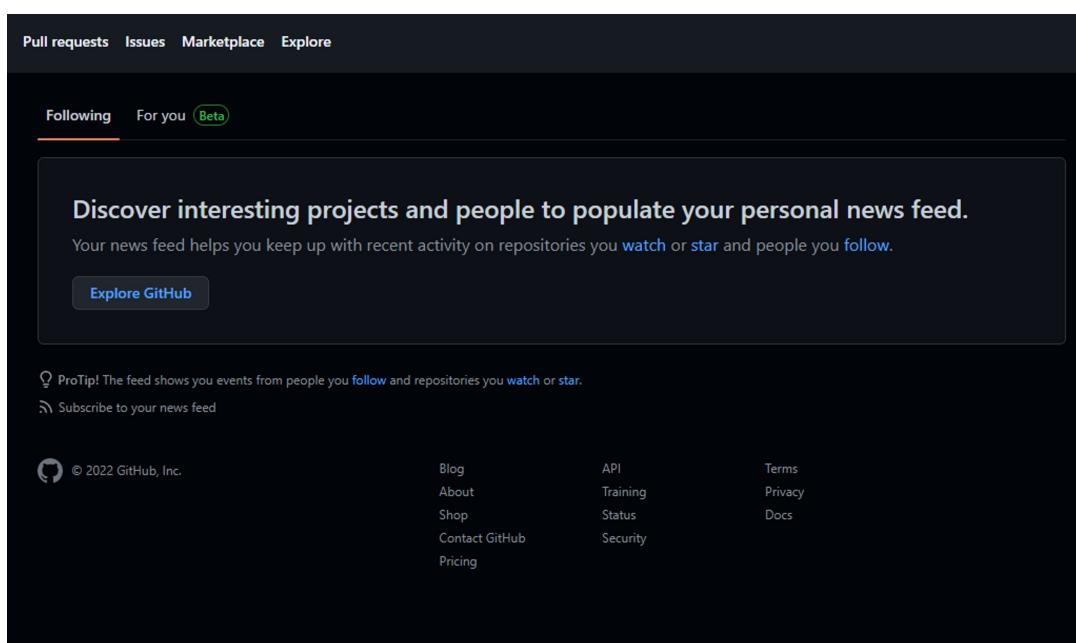
En résumé :

- **Un dépôt** est comme un **dossier qui conserve un historique des versions et des modifications d'un projet**. Il est essentiel pour travailler en équipe ou collaborer à un projet open source.
- **Un dépôt local** est l'endroit où l'on stocke, **sur sa machine**, une copie d'un projet, ses différentes versions et l'historique des modifications.
- **Un dépôt distant** est une **version dématérialisée** du dépôt local, que ce soit sur Internet ou sur un réseau. Il permet de centraliser le travail des développeurs dans un projet collectif.
- Il existe plusieurs services en ligne pour héberger un dépôt distant, **GitHub étant l'un des plus populaires**.

## Github

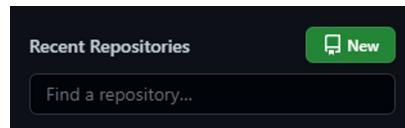
GitHub est composé de différent menu :

- **tableau de bord personnel** :
  - o **Suivre les problèmes** et **extraire les demandes** sur lesquelles vous travaillez ou que vous suivez ;
  - o Accéder à vos principaux **repositories** et pages d'équipe ;
  - o **Rester à jour** sur les **activités récentes** des organisations et des **repositories** auxquels vous êtes **abonné**.



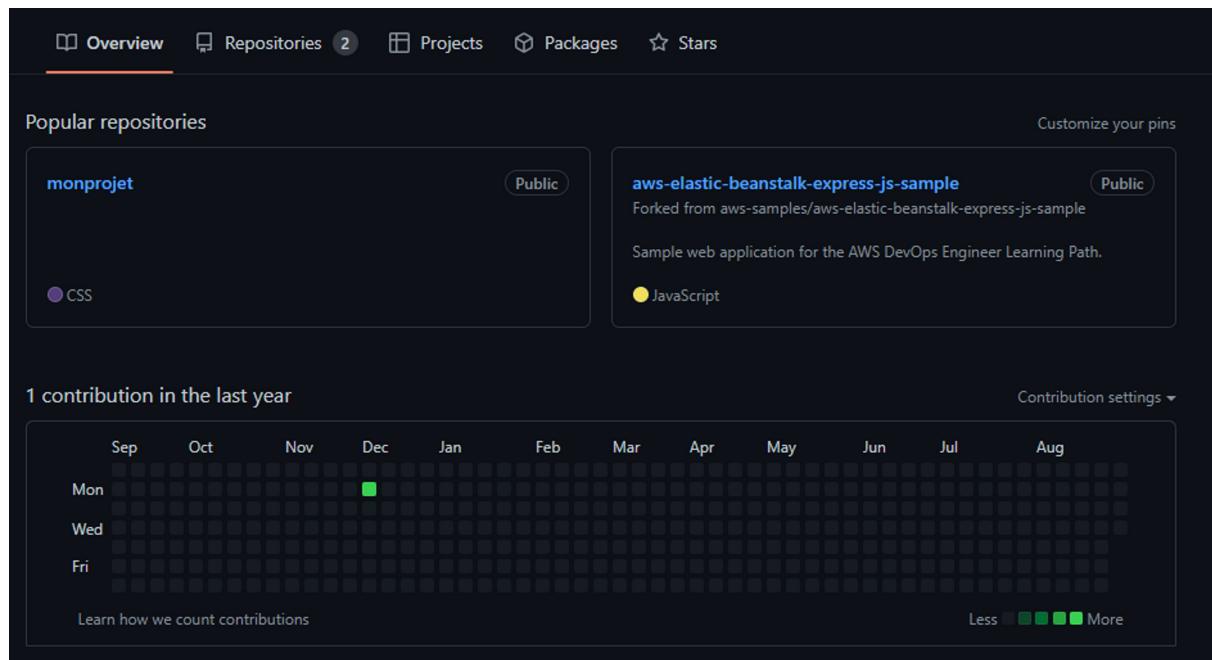
## - L'interface repository

- L'**interface Repositories** est l'emplacement où vous pourrez **créer et retrouver vos dépôts existants**.
- Pour créer un projet, il suffit de cliquer sur "**Start a project**".



## - Votre profil

- Sur votre profil, vous pourrez **éditer vos informations**, mais aussi voir le **total de vos contributions sur les différents projets**.
- Les **contributions** sont toutes les **actions sur des repositories que vous allez effectuer**. Que ce soient **vos repositories, ceux d'autres personnes ou des repositories publics**.



## - L'onglet Pull requests :

- Permet de faire des  **demandes de modifications** réalisées sur le code
- Les **pull requests** (ou **demandes de pull**), vous permettent d'**informer** les autres utilisateurs des **modifications que vous avez appliquées à une branche d'un repository sur GitHub**, et que vous voulez **fusionner avec le code principal**.

The screenshot shows a GitHub search interface with a dark theme. At the top, there are tabs for 'Created', 'Assigned', 'Mentioned', and 'Review requests'. A search bar contains the query 'is:open is:pr author:VelStack archived:false'. Below the search bar, it says '0 Open' and '0 Closed'. On the right, there are buttons for 'Visibility', 'Organization', and 'Sort'. The main area displays a message: 'No results matched your search.' followed by a note: 'You could search all of GitHub or try an advanced search.' At the bottom, there is a 'ProTip!' message: 'Exclude everything labeled bug with -label:bug.'

- La fonctionnalité **Explore** :

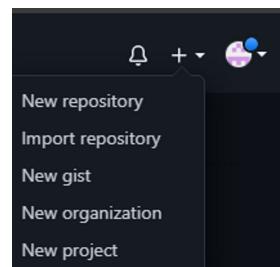
- Un des derniers points importants sur GitHub est la fonctionnalité Explore.
- Via Explore, vous pourrez **trouver de nouveaux projets open source** sur lesquels travailler, en parcourant les projets recommandés, en vous connectant à la communauté GitHub et en recherchant des repositories par sujet ou par libellé.

The screenshot shows the GitHub Explore page. At the top, there are navigation links: 'Explore', 'Topics', 'Trending', 'Collections', 'Events', and 'GitHub Sponsors'. Below this, a message reads 'Here's what we found based on your interests...'. It shows a card for the topic '# hash-drbg', which has 1 public repository matching. The repository listed is 'ANSSI-FR / libdrbg', described as 'A library implementing NIST SP 800-90A DRBGs'. There is a 'See more matching repositories' button at the bottom of the card.

### Création d'un nouveau Dépôt

Pour mettre votre projet sur GitHub, vous devez **créer un repository** (ou dépôt en français) dans lequel il pourra être installé.

Cliquez sur le "+" dans le coin supérieur droit, pour faire apparaître l'option New repository.

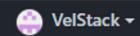


Choisissez un nom simple pour votre dépôt. Puis, choisissez si vous souhaitez créer un **dépôt public ou privé**.

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*



VelStack

Repository name \*

OpenClassroomProject



Great repository names are short and memorable. Need inspiration? How about [upgraded-octo-garbanzo](#)?

Description (optional)

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None

Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)

License: None

You are creating a public repository in your personal account.

**Create repository**

README est un fichier qui indique les **informations clés de votre projet** : description, environnement à utiliser, dépendances possibles et droits d'auteurs. C'est un peu comme le **mode d'emploi** de votre projet.

.gitignore est un fichier qui permet d'**ignorer certains fichiers de votre projet Git**.

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# OpenClassroomProject" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/VelStack/OpenclassroomProject.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/VelStack/OpenclassroomProject.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

Pour la suite du cours, il est nécessaire d'installer **Git** et **Git Bash** pour procéder aux manipulations via le **shell**

Nous allons créer un nouveau dossier qui servira de dépôt Git :

```
aresv@MSI MINGW64 ~
$ cd Documents
aresv@MSI MINGW64 ~/Documents
$ mkdir ProjetGit
aresv@MSI MINGW64 ~/Documents
$ cd ProjetGit/
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ ...
```

Une fois le dossier créé, la première chose à faire est de **configurer son identité** avec les commandes `git config --global user.name` et `git config --global user.email` :

- `git config --global user.name "John Doe"`
- `git config --global user.email johndoe@example.com`

```
Tiffany@DESKTOP-RAAC109 MINGW64 ~/Documents/premierProjet
$ git config --global user.name "John Doe"

Tiffany@DESKTOP-RAAC109 MINGW64 ~/Documents/premierProjet
$ git config --global user.email johndoe@example.com
```

Ces informations sont importantes car elles sont **obligatoire** pour les validations dans Git. Si vous souhaitez, pour un **projet spécifique**, changer votre nom d'utilisateur, vous devrez repasser cette ligne mais sans le `--global`.

Pour vérifier les paramètres, il suffit de passer la commande `git config --list`

```
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager-core
credential=https://dev.azure.com/usehttppath=true
init.defaultbranch=master
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
user.name: [REDACTED]
user.emai: [REDACTED]
```

## Activer les couleurs

Git n'active pas les couleurs de base, pour améliorer la lisibilité il est conseillé de les activer : pour ce faire, on peut passer ces 3 lignes de commande :

- **git config --global color.diff auto**
- **git config --global color.status auto**
- **git config --global color.branch auto**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ git config --global color.status auto
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ git config --global color.branch auto
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ ...
```

## Configurer l'éditeur

Par défaut, Git utilise **Vim** comme **éditeur** et **Vimdiff** comme **outil de merge**. Vous pouvez les modifier en utilisant :

- **git config --global core.editor notepad++**
- **git config --global merge.tool vimdiff**

L'outil de merge permet de fusionner deux parties distinctes d'un projet.

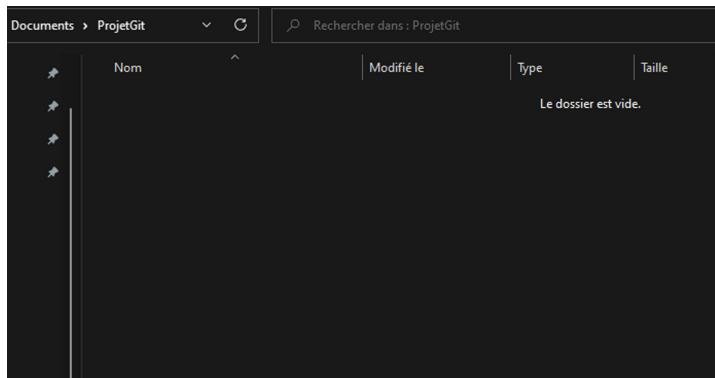
```
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ git config --global core.editor notepad++
aresv@MSI MINGW64 ~/Documents/ProjetGit
$ git config --global merge.tool vimdiff
```

## Initialisez votre dépôt en créant un dépôt local vide

Pour cet exemple, nous allons créer un dossier "**ProjetGit**" dans le dossier "**Documents**" :

- Allez dans "Document".

- Créez le dossier “ProjetGit”.
- Accédez à votre dossier.



Puis lancez ces deux lignes de commande dans Git Bash :

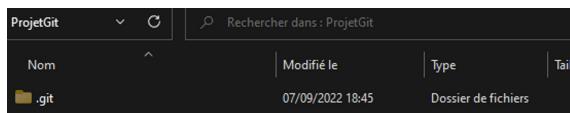
- **cd Documents/ProjetGit**
- **git init**

La première ligne permet de vous positionner dans le dossier que vous venez de créer sur l'ordinateur. La seconde ligne va initialiser ce “simple dossier” comme un dépôt.

```
aresv@MSI MINGW64 ~
$ cd Documents/ProjetGit/

aresv@MSI MINGW64 ~/Documents/ProjetGit
$ git init
Initialized empty Git repository in C:/Users/aresv/Documents/ProjetGit/.git/
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$
```

Rien de nouveau n'apparaît dans le dossier, mais un **dossier caché ".git"** a été créé :

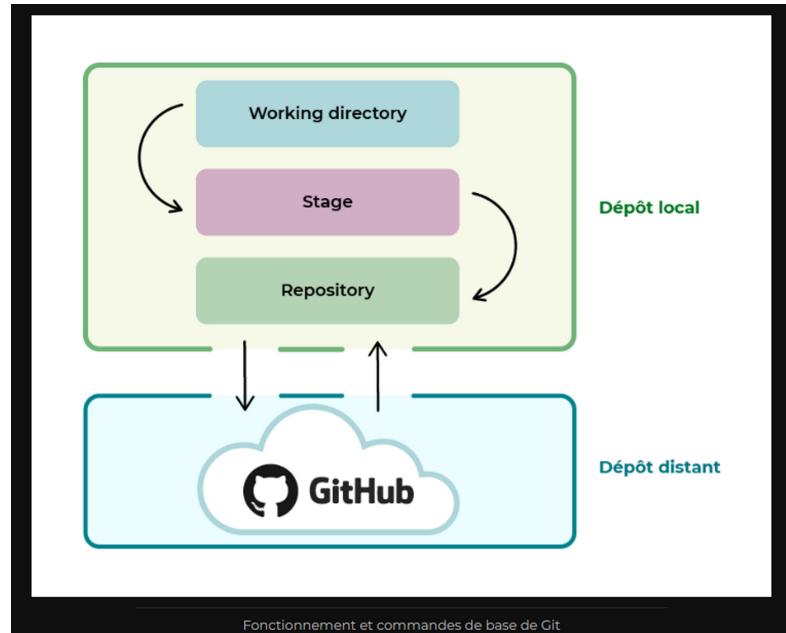


Ce dossier caché contient tous les éléments non visibles de Git : la configuration, les “logs”, les branches...

En résumé :

- Pour initialiser un dépôt Git, vous pouvez soit **créer un dépôt local vide**, soit **cloner un dépôt distant**.
- **git init** permet d'**initialiser un projet Git**.

## [Travailler depuis un dépôt Git local](#)



Ce schéma représente le **fonctionnement de Git**. Il est composé de **3 zones** qui forment le dépôt local, et du dépôt distant GitHub :

- **Le Working Directory** : Cette zone correspond au **dossier du projet sur votre ordinateur**.
- **Le Stage ou Index** : Cette zone est un **intermédiaire** entre le **working directory** et le **repository**. Elle représente **tous les fichiers modifiés** que vous souhaitez voir apparaître dans votre **prochaine version** de code.
- **Le Repository** : Lorsque l'on **crée de nouvelles versions d'un projet**, c'est dans cette zone qu'elles sont **stockées**.

Ces 3 zones sont présentes dans votre ordinateur, en **local**.

Prenons un exemple : Imaginons un projet composé de 3 fichiers : fichier1, fichier2 et fichier3.

Nous faisons une modification sur fichier1, puis une modification sur fichier2, depuis le **working directory**. Nous aimerais **sauvegarder** cette version grâce à Git, c'est-à-dire la **stocker dans le repository**.

Comment faire ?

nous allons envoyer les fichiers modifiés (fichier1 et fichier2) **du working directory vers l'index**. On dit qu'on va **indexer** fichier1 et fichier2. Une fois les fichiers indexés, nous pouvons **créer une nouvelle version** de notre projet.

#### Mise en situation

Nous allons mettre en place une base de projet web, avec un fichier HTML et un fichier CSS pour mettre tout cela en pratique.

Avant tout, vous devez vous situer dans le dépôt Git du projet : Pour vous en assurer, tapez la commande "pwd" dans Git Bash.

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ pwd
/c/Users/aresv/Documents/ProjetGit
```

Nous allons ensuite initialiser un dépôt en nous basant sur l'exemple "ProjetGit" que j'ai créer sur mon ordinateur :

Nous allons créer un fichier "index.html" avec la commande UNIX "touch" :

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ touch index.html

aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ ls
index.html
```

La commande UNIX "ls" me permet d'afficher les répertoires et les fichiers dans le dossier dans lequel je suis positionné, ici "ProjetGit"

Le fichier "index.html" se trouve dans la partie **Working Directory** de Git, nous pouvons vérifier sa présence dans cette section avec la commande : **git status**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    index.html

nothing added to commit but untracked files present (use "git add" to track)
```

Grâce à la commande **git status** nous voyons notre fichier apparaître en **rouge** : cela signifie qu'il vient d'être **créer**, **modifié** ou **supprimé** mais qu'il n'est pas encore **indexé**.

Pour y remédier, faite la commande : **git add index.html**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git add index.html
```

La commande ne renvoie rien quand elle est exécuté

Pour vérifié les modification apporté par la commande **git add** , il faut retaper la commande **git status**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   index.html
```

Notre fichier se trouve maintenant dans le répertoire **index/stage**

Si vous devez à nouveau apporté des modifications à votre "index.html", il faudra à nouveau retaper la commande **git add** pour prendre en compte les modifications

Une fois notre fichier "index.html" dans la section **stage**, nous allons voir comment le **commit** : la phase d'enregistrement de git

Nous allons utiliser **git commit** pour **créer une nouvelle version du projet et passer les fichiers présent dans le stages/index vers le repository**

Ici nous avons deux possibilité :

- **git commit -m "description de commit"**
- **git commit**

*/!\ Si vous utilisez git commit, il faudra écrire la description du commit dans la fenêtre de l'éditeur VIM /!\*

Ici nous allons utiliser le 1er choix en tapant la commande **git commit -m "Ajout du fichier index.html"**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git commit -m "Ajout du fichier index.html"
[master (root-commit) e351474] Ajout du fichier index.html
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.html
```

Nous venons de créer la **1ère version** de notre **projet**, sur la branche **principale** du projet : la branche **master**

Nous allons continuer à modifier nos fichiers :

Nous allons rajouter ce contenu à notre fichier **index.html**

```
<!DOCTYPE html>
<html>
<head>
<title></title>
<link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
<h1>Un super titre</h1>
</body>
</html>
```

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  |   <title></title>
5  |   <link rel="stylesheet" type="text/css" href="styles.css">
6  </head>
7  <body>
8  |   <h1>Un super titre</h1>
9  </body>
10 </html>
```

*J'utilise personnellement VSCode pour modifier mes fichiers, mais vous pouvez utiliser n'importe quel autre éditeur de code en fonction de vos préférences*

Nous allons également créer un nouveau fichier, **CSS** cette fois-ci, que nous allons appeler "**styles.css**"

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ touch styles.css

aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ ls
index.html  styles.css
```

Nous allons le modifier avec les lignes de code suivante :

```
h1 {
  color: red;
}
```

```
1  h1 {
2  |   color: red;
3  }
```

*N'oubliez pas de sauvegarder vos modifications sur votre éditeur de code préféré !*

Vous pouvez ouvrir le fichier “**index.html**” dans un navigateur pour vérifier que tout fonctionne bien. Si c'est le cas, vous devriez avoir une **page blanche** avec le **titre “Un super titre” en rouge**.

# Un super titre

## Indexez vos fichiers avec la commande git add

Faisons un git status pour vérifier l'état de notre branche master :

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    styles.css

no changes added to commit (use "git add" and/or "git commit -a")
```

Maintenant que nous avons modifier les fichiers présent dans notre dépôt Git, nous allons les **indexer** :

Pour créer une nouvelle version de notre projet, il faut se rendre dans git bash et utiliser la commande **git add suivi du nom des fichiers** à pousser dans l'index.

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git add index.html styles.css
```

Vérifions le résultat de l'indexation avec **git status** :

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   index.html
    new file:   styles.css
```

Nos fichiers sont maintenant **indexé**

## Envoyez votre commit sur le dépôt distant avec la commande git push

Maintenant que nous avons créer une version de notre projet, il faut désormais passer notre commit du repository vers notre dépôt distant : On dit qu'il faut “**pusher**” notre commit.

Notre premier push va vous demander un peu de configuration.

Pour commencer, nous allons devoir “**reliez**” notre **dépôt local au dépôt distant** que nous avons créé sur GitHub précédemment. Pour cela :

- Allez sur GitHub.

- Cliquez sur la petite image en haut à droite.
- Cliquez sur “your repositories”.
- Cliquez sur le repository créé dans la première partie du cours, “OpenClassroomsProject”.

Nous devrions avoir un écran comme celui-ci :

The screenshot shows a GitHub repository page for 'EtudiantOC / OpenClassroomsProject'. At the top, there are buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). Below the header, there are tabs for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. The 'Code' tab is selected. The main content area has three sections:

- Quick setup — if you've done this kind of thing before**: This section provides instructions for setting up a local repository. It includes a 'Set up in Desktop' button, an 'HTTPS' button, an 'SSH' button, and a URL 'https://github.com/etudiantOC/OpenClassroomsProject.git'. It also suggests creating a new file or uploading an existing one, and recommends including a README, LICENSE, and .gitignore.
- ...or create a new repository on the command line**: This section contains a block of terminal commands for initializing a new repository and adding it to GitHub:
 

```
echo "# OpenClassroomsProject" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/lucbourrat/OpenClassroomsProject.git
git push -u origin main
```
- ...or push an existing repository from the command line**: This section contains a block of terminal commands for pushing an existing repository to GitHub:
 

```
git remote add origin https://github.com/lucbourrat/OpenClassroomsProject.git
git branch -M main
git push -u origin main
```
- ...or import code from another repository**: This section provides instructions for initializing a repository with code from other version control systems like Subversion, Mercurial, or TFS. It includes a 'Import code' button.

Il faut copier le lien qui figure à l'écran :

#### Quick setup — if you've done this kind of thing before

<https://github.com/etudiantOC/OpenClassroomsProject.git>

Une fois le lien copié, il faut retourner dans git bash et taper la commande suivante :

- git remote add origin *votre lien GitHub*

Il faut ensuite taper la commande **git branch -M main**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (master)
$ git branch -M main
```

Nous avons relié le dépôt local au dépôt distant. Nous pouvons donc envoyer des commits du repository vers le dépôt distant GitHub en utilisant la commande suivante : **git push -u origin main**

```

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 231 bytes | 231.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/VelStack/OpenclassroomProject.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.

```

Notre version du projet est maintenant stocké sur **GitHub** : dans le Cloud

Reprenez le fichier HTML créé. Disons que vous allez modifier le titre "Un super titre" en "Tataru est un super titre !".

```

C: > Users > aresv > Documents > ProjetGit > index.html > html > body > h1
1  <!DOCTYPE html>
2  <html>
3  <head>
4  |   <title></title>
5  |   <link rel="stylesheet" type="text/css" href="styles.css">
6  </head>
7  <body>
8  |   <h1>Tataru est un super titre</h1>
9  </body>
10 </html>

```

Pour **créer une version du projet** avec le fichier HTML à jour et **l'envoyer sur le dépôt distant GitHub**, nous allons :

**Indexer** le fichier HTML modifié grâce à la commande :

- **git add index.html**

**Créer une nouvelle version** grâce à la commande :

- **git commit -m "Modification du titre H1"**

**Envoyer la nouvelle version sur le dépôt distant** grâce à la commande :

- **git push origin main**

Nous devrions obtenir ce résultat :

```

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git add index.html

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git commit -m "Modification du titre H1 sauce FFXIV"
[main a6d85c5] Modification du titre H1 sauce FFXIV
2 files changed, 13 insertions(+)
create mode 100644 styles.css

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git push origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 463 bytes | 463.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/VelStack/OpenclassroomProject.git
 e351474..a6d85c5 main -> main

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ ...

```

 index.html	Modification du titre H1 sauce FFXIV	2 minutes ago
 styles.css	Modification du titre H1 sauce FFXIV	2 minutes ago

En résumé :

- **git add** permet d'ajouter des fichiers dans l'**index**, qui est une **zone intermédiaire** dans laquelle stocker les fichiers modifiés.
- **git commit** permet de créer une nouvelle version avec les **fichiers situés dans l'index**.
- **git commit -m** permet de créer une nouvelle version et de préciser le message rattaché au commit.
- **git push** permet d'envoyer les modifications faites en local vers un dépôt distant.

## Comprendre le système de branche

Le principal atout de Git est son **système de branches** : Les différentes branches correspondent à des **copies** de votre code principal à un instant T, où vous pourrez tester des fonctionnalités sans que cela n'impacte votre code principal.

Sous Git, la branche principale est appelée la branche **main**, ou **master** pour les dépôts créés avant octobre 2020.

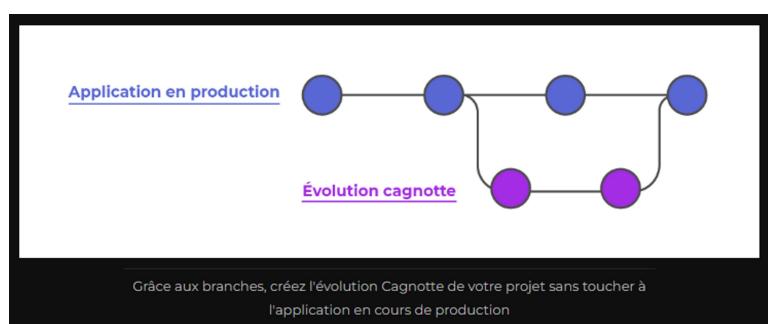
**La branche principale (main ou master)** portera l'**intégralité** des modifications effectuées. Le but n'est donc pas de réaliser les modifications directement sur cette branche, mais de les réaliser sur d'autres branches, et après divers tests, de les intégrer sur la branche principale.

Il faut voir les branches comme autant de **dossiers différents**. Prenons un exemple concret :

Imaginez que vous avez réalisé une superbe application bancaire pour M. Robert, et que ce dernier ait une superbe idée de cagnotte à ajouter à son application. Faire la modification directement sur la branche main risquerait de faire **planter l'application**

Avec Git et son système de branches, vous pouvez créer une branche correspondant à l'évolution "cagnotte", et cela sans toucher à votre application en cours de production.

Une fois que toutes vos modifications auront été testées, vous pourrez les envoyer en production sans crainte en les intégrant à la branche main (et dans le pire des cas, revenir en arrière simplement).



Git va créer une **branche virtuelle**, mémoriser tous vos changements, et seulement quand vous le souhaitez, les ajouter à votre application principale. Il va également vérifier s'il n'y a pas de conflits avec d'autres fusions.

## Apprendre à utiliser les branches

La commande pour lister les branches est : **git branch**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git branch
* main
```

Pour savoir sur quelle branche nous sommes situés, une étoile apparaît avant son intitulé (ici nous sommes sur la branche main, donc \*main)

L'idéal est de créer une branche par fonctionnalité : nous allons donc créer la branche "cagnotte" avec la commande "git branch cagnotte".

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git branch cagnotte

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git branch
  cagnotte
* main
```

Nous sommes toujours sur la branche main : il faut utiliser la commande "git checkout cagnotte" pour changer de branche

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git checkout cagnotte
Switched to branch 'cagnotte'

aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ git branch
* cagnotte
  main

aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ :
```

Nous allons maintenant créer un fichier **cagnotte.txt** avec la commande **touch** (vérifiez bien que vous êtes sur la branche **cagnotte**) puis l'indexer avec la commande "**git add cagnotte.txt**". Nous allons ensuite **commit** le fichier avec la commande **git commit -m "Réalisation de la cagnotte côté Front-End"**

```
aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ touch cagnotte.txt

aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ git add cagnotte.txt

aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ git commit -m "Réalisation de la cagnotte côté Front-End"
[cagnotte e8faabf] Réalisation de la cagnotte côté Front-End
 1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 cagnotte.txt
```

Nous allons l'envoyer sur le dépôt distant GitHub : il faut passer la commande "**git push -u origin cagnotte**"

```

aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ git push -u origin cagnotte
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 353 bytes | 353.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'cagnotte' on GitHub by visiting:
remote:     https://github.com [REDACTED]/OpenclassroomProject/pull/new/cagnotte
remote:
To https://github.com/VelStack/OpenclassroomProject.git
 * [new branch]      cagnotte -> cagnotte
Branch 'cagnotte' set up to track remote branch 'cagnotte' from 'origin'.

```

Cette commande va créer la branche cagnotte sur le dépôt distant GitHub

Nous allons maintenant **fusionner** la branche cagnotte et main pour appliquer nos modifications de la branche cagnotte : c'est ce qu'on appelle un **merge**. Pour effectuer cette manipulation, nous allons devoir changer de branche car **un merge nécessite de se situer sur la branche sur laquelle nous voulons fusionner nos modifications**.

```

aresv@MSI MINGW64 ~/Documents/ProjetGit (cagnotte)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git branch
  cagnotte
* main

```

Pour effectuer notre merge des deux branches, il faut utiliser la commande **git merge cagnotte**

```

aresv@MSI MINGW64 ~/Documents/ProjetGit (main)
$ git merge cagnotte
Updating a6d85c5..e8faabf
Fast-forward
  cagnotte.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 cagnotte.txt

```

Notre fusion s'est déroulé correctement

En résumé :

- **Une branche** est une “copie” d’un projet sur laquelle **on opère des modifications de code**.
- **La branche main** (ou autrefois master) est la **branche principale d’un projet**.
- **git checkout** permet de **basculer d’une branche à une autre**.

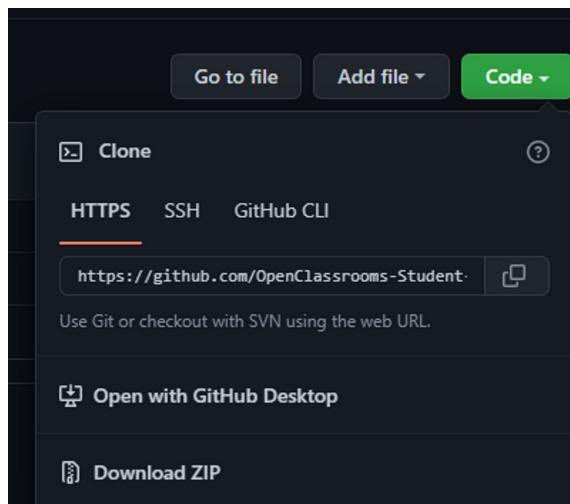
- git merge permet de fusionner deux branches.

## Travaillez avec un dépôt distant

Imaginons que nous devions travailler sur un projet avec des amis, ces derniers ont créé un repository sur GitHub : Nous allons voir comment travailler avec un **repository distant**.

Tout d'abord, vous allez récupérer l'URL du **dépôt distant** : cela se passe sur **GitHub** avec ce lien : <https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub>

Pour récupérer le code, nous allons cliquer sur le bouton "**code**", puis la section **HTTPS**



Nous pouvons récupérer l'URL dont nous avons besoin pour récupérer le dossier : pour cela nous allons utiliser la commande **git clone** dans notre **git bash** (fais attention à bien vous trouver dans le dossier "Documents")

- git clone <https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub.git>

```
aresv@MSI MINGW64 ~/Documents/OpenclassroomGit
$ git clone https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub.git
Cloning into '7162856-G-rez-Git-et-GitHub'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 18 (delta 3), reused 1 (delta 1), pack-reused 10
Receiving objects: 100% (18/18), done.
Resolving deltas: 100% (5/5), done.
```

Nous pouvons créer un raccourci pour gagner du temps avec la commande :

- git remote add OC <https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub.git>

```
aresv@MSI MINGW64 ~/Documents/OpenclassroomGit/7162856-G-rez-Git-et-GitHub (main)
$ git remote add OC https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub.git
```

**OC** représente le **nom court** que vous utiliserez ensuite pour **appeler votre dépôt**. Nous pouvons l'appeler comme bon nous semble, mais un nom court et simple est toujours plus facile.

Cette ligne ne nous permet pas de copier le dépôt, mais permet de dire au dépôt que l'on pointe vers le dépôt distant.

Et voilà! Nous pouvons nous diriger dans le dossier sur votre ordinateur et accéder au code.

## Mettre à jour le dépôt local

Imaginons que des modifications ont été apporté sur la branche main et que nous voulons les récupérer, nous pouvons utiliser la commande :

- `git pull OC main`

```
aresv@MSI MINGW64 ~/Documents/OpenClassroomGit/7162856-G-rez-Git-et-GitHub (main)
$ git pull OC main
From https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub
 * branch            main      -> FETCH_HEAD
 * [new branch]      main      -> OC/main
Already up to date.
```

Nous devrions maintenant avoir l'ensemble des fichiers et dossiers du repository à jour dans notre répertoire courant. La duplication s'effectue très rapidement, puisque les fichiers vont être compressés avant le transfert.

## Collaborez sur GitHub

*en quoi GitHub aide les développeurs à collaborer ?*

GitHub est avant tout une **interface**, son rôle est de **faciliter la collaboration** en ajoutant un **aspect design, facile, ludique et simple d'utilisation pour les développeurs**. Jusqu'ici nous avons observé les changements effectués sur des branches via des lignes de commande. Si nous travaillons à plusieurs sur un projet, il serait bien plus pratique d'accéder aux modifications sans ligne de commande.

## Vérifiez les branches de votre projet

Nous allons vérifier les différentes branches du projet OpenClassroom : utilisons la commande `git branch`

Nous devrions voir :

```
* main
new-version-css
update-readme
```

Nous avons en plus de la branche principale main, deux autres branches : `new-version-css` et `update-readme`

Dans les chapitres précédents, `git merge` nous permettait de **fusionner** les modifications de notre branche avec la branche principale.

Dans un contexte professionnel, c'est un peu plus compliqué : Lorsque nous travaillons en équipe sur un repository, la **branche principale est souvent bloquée**. Nous ne pouvons **pas pusher directement votre code sans qu'il soit vérifié**. Nous ne pouvons donc pas fusionner nos **modifications nous-même** !

Pour effectuer des modifications, il faut effectuer une **pull request**

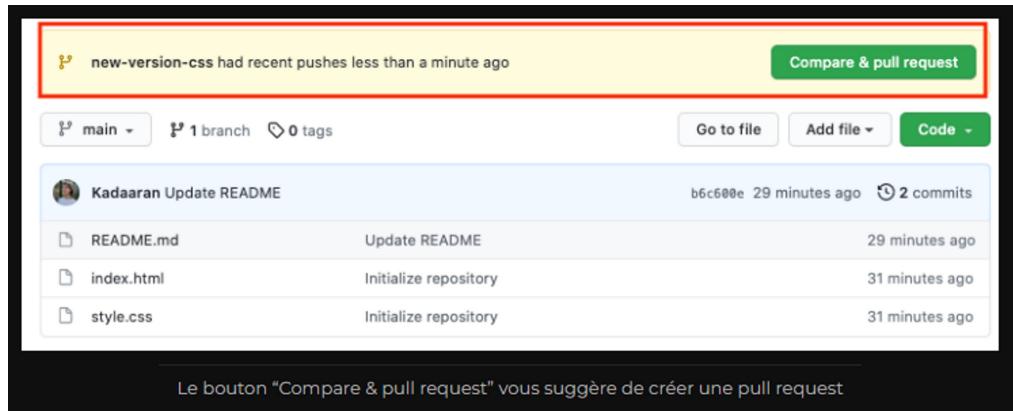
## Réalisez une pull request

Une **pull request**, ou **demande de pull**, est une **fonctionnalité de GitHub** qui permet de **demandez aux propriétaires d'un repository l'autorisation de fusionner nos changements sur la branche principale ou toute autre branche** sur laquelle on souhaite apporter nos modifications.

Si nous créons une **pull request**, nous devons au préalable :

- Crée une nouvelle branche.
- Envoyé notre code sur cette même branche.

D'ailleurs, lorsque ces deux conditions sont remplies, un **bandeau apparaît à l'écran pour vous suggérer de créer une pull request** :



The screenshot shows the "Open a pull request" window. At the top, it says "base: main" and "compare: new-version-css" with a note "Able to merge. These branches can be automatically merged.". Below this is a text input field with the placeholder "Update style, turn all pages into blue" and a toolbar with "Write" and "Preview" buttons. To the right are sections for "Reviewers" (No reviews), "Assignees" (No one—assign yourself), "Labels" (None yet), "Projects" (None yet), "Milestone" (None yet), and "Linked issues" (Use Closing keywords in the description to automatically close issues). Below these are "Helpful resources" links to "GitHub Community Guidelines" and "GitHub Help". At the bottom, there are summary statistics: "1 commit", "1 file changed", "0 comments", and "1 contributor". The commit details show "Commits on Aug 22, 2021" with a single commit "Update style, turn all pages into blue" (1bcf85d). The diff view shows a change in "style.css": 

```
@@ -1,3 +1,3 @@  
 1 1  h1 {  
 2 2 -  color: red;  
 3 3 +  color: blue;  
 }
```

 At the bottom of the window, a caption reads "La fenêtre Open a pull request s'ouvre". Below the window, a bullet point lists: "Ajoutez un commentaire pour expliquer les raisons de vos modifications."

Ici, notre modification consiste à changer la couleur d'une balise. Nous pouvons commenter : "Change h1 color from red to blue".

GitHub indique les modifications effectuées par un code couleur. Les lignes en rouge indiquent une suppression, et les lignes vertes une addition. Ici on voit bien qu'il y eu un changement sur une ligne, l'attribut red a été supprimé, et a été remplacé par blue.

Pour valider la pull request, nous devons cliquer sur le bouton "Create a pull request"

The screenshot shows a GitHub Pull Request page. The title is "Update style, turn all pages into blue #1". A green button labeled "Merge pull request" is highlighted. A comment from "Kadaaran" says "Change h1 color from red to blue." On the right side, there are sections for Reviewers, Assignees, Labels, Projects, Milestone, Linked issues, and Notifications. At the bottom, there's a "Comment" button and a "Lock conversation" link.

### Demandez une relecture de code

Sur des projets d'envergure, il peut arriver que votre code ne puisse être fusionné sur la branche principale sans être relu et validé par d'autres membres du projet. C'est ce qu'on appelle une **Code Review**, ou **revue de code**, en français. Cela permet de prévenir les erreurs éventuelles, de discuter sur un choix, une prise de position ou même de poser des questions.

The screenshot shows a GitHub Pull Request review. A comment from "Kadaaran" on March 30, 2019, states: "Oops. 404. La redirection arrive sur une page Rails hors il faut que ce soit sur la vue Login qui est sur Angular." Below the comment is a "Reply..." button and a "Resolve conversation" button. At the bottom, there's a note: "La revue de code permet d'échanger sur une Pull Request".

En résumé :

- Sur GitHub, nous pouvons récupérer l'URL d'un dépôt distant.

- **git clone** permet de copier en local un dépôt distant.
- **git remote add** permet de lier un dépôt à un "nom court", pour une plus grande facilité d'utilisation.
- **git pull** permet de dupliquer un dépôt GitHub en local.
- Une **Pull Request** permet de demander à fusionner votre code sur la branche principale.

## Pratiquer et corriger ses erreurs sur un dépôt local

*Git est un outil merveilleux, mais on a vite fait de créer une branche alors qu'on ne le souhaitait pas, de modifier la branche principale ou encore d'oublier des fichiers dans ses commits. Nous allons voir comment corriger ces erreurs avec les bonnes techniques.*

Nous allons dans un 1er temps créer un "bac à sable" pour nous entraîner, nous l'appellerons test

```
aresv@MSI MINGW64 ~/Documents
$ ls
'Ma musique'@ 'Mes vidéos'@   OpenClassroomGit/   Spelunky2/
'Mes images'@ 'My Games'@     ProjetGit/        test/
aresv@MSI MINGW64 ~/Documents
$ :
```

Initialisez le dépôt avec la commande **git init**.

```
aresv@MSI MINGW64 ~/Documents/test
$ git init
Initialized empty Git repository in C:/Users/aresv/Documents/test/.git/
```

Votre dépôt est maintenant **initialisé**. Si vous faites apparaître les dossiers masqués, vous pouvez voir le dossier `.git`.

Grâce à la ligne de commande `ls -la`, vous pouvez faire apparaître les dossiers cachés.

```
aresv@MSI MINGW64 ~/Documents/test (master)
$ ls -la
total 8
drwxr-xr-x 1 aresv 197609 0 Sep 12 10:54 .
drwxr-xr-x 1 aresv 197609 0 Sep 12 10:53 ..
drwxr-xr-x 1 aresv 197609 0 Sep 12 10:54 .git/
```

Un des avantages majeurs de Git réside dans l'**aspect local des travaux réalisés** : un dépôt Git gère son cycle de vie **localemement, indépendamment de la connectivité avec son dépôt distant**. Tout se passe directement sur votre ordinateur.

## J'ai créé une branche que je n'aurais pas dû créer

Notre bac à sable est à présent opérationnel :

Avant de créer une branche, vous devez créer votre branche principale :

- Il vous suffit d'**ajouter un fichier** et de le **commiter**.

- Créez un fichier "PremierFichier.txt" dans votre répertoire Test : `touch PremierFichier.txt`

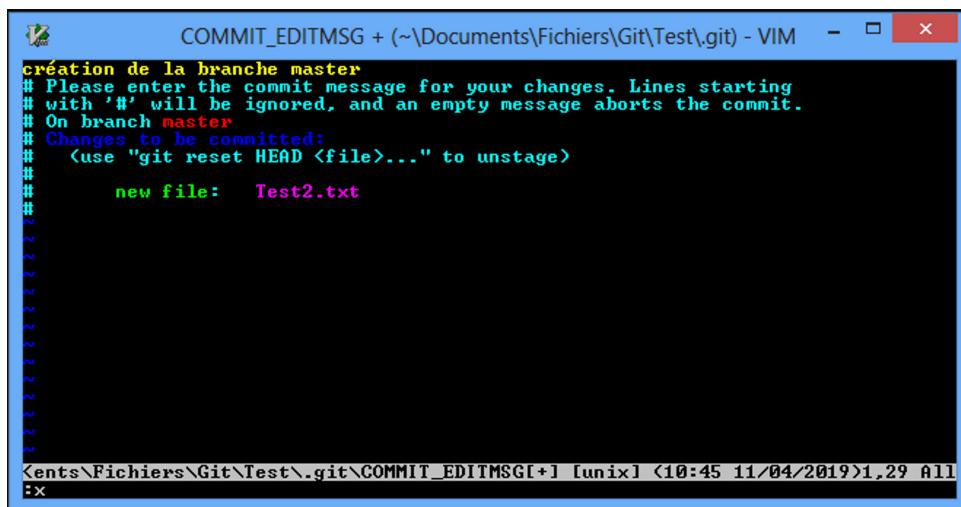
```
aresv@MSI MINGW64 ~/Documents/test (master)
$ touch PremierFichier.txt

aresv@MSI MINGW64 ~/Documents/test (master)
$ ls
PremierFichier.txt
```

- Ajoutez-le avec les commandes : `git add PremierFichier.txt` et `git commit`

On vous demande alors d'indiquer le message du commit puis de valider. Pour valider le message, une fois que vous l'avez écrit, appuyez sur Echap (votre curseur va basculer sur la dernière ligne) et tapez `:x`.

Cette commande va sauvegarder et quitter l'éditeur des messages de commit.



Vous pouvez également utiliser la commande `git commit -m "rajout du 1er fichier"`

```
aresv@MSI MINGW64 ~/Documents/test (master)
$ git commit -m "rajout du 1er fichier texte"
[master (root-commit) 3443bc6] rajout du 1er fichier texte
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 PremierFichier.txt
```

Pour renommer une branche : `git branch -M nouveau nom de la branche sur laquelle nous sommes situés`

```
aresv@MSI MINGW64 ~/Documents/test (master)
$ git branch -M main

aresv@MSI MINGW64 ~/Documents/test (main)
$
```

Suppression d'une branche

Pour supprimer une branche dans notre dépôt, il faut utiliser la commande `git branch -d nom de la branche`

```

aresv@MSI MINGW64 ~/Documents/test (master)
$ git branch brancheTest

aresv@MSI MINGW64 ~/Documents/test (master)
$ git branch
  brancheTest
* master

aresv@MSI MINGW64 ~/Documents/test (master)
$ git branch -d brancheTest
Deleted branch brancheTest (was 3443bc6).

aresv@MSI MINGW64 ~/Documents/test (master)
$ git branch
* master

aresv@MSI MINGW64 ~/Documents/test (master)
$ ...

```

*J'ai ici créé une nouvelle branche que j'ai tout de suite supprimé avec la commande `git branch -d`*

Si vous avez déjà fait des modifications dans la branche que vous souhaitez supprimer, vous pouvez la supprimer avec la commande :

- `git branch -D brancheTest`

La suppression de cette branche entraînera la suppression de tous les fichiers et modifications que vous n'aurez pas commis sur cette branche.

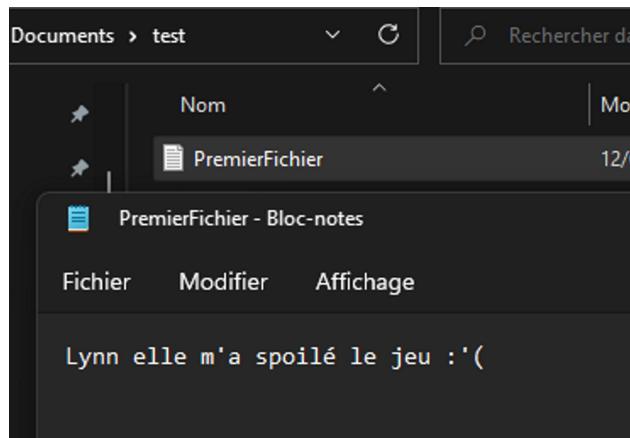
#### Modification de la branche principale

Si vous avez modifié votre branche principale (main ou master) avant de créer votre branche et que vous n'avez pas fait le commit, ce n'est pas bien grave. Il vous suffit de faire une remise - ou un stash en anglais.

**Stash : La remise, ou stash, permet de mettre vos modifications de côté, les ranger, le temps de créer votre nouvelle branche et d'appliquer cette remise sur la nouvelle branche.**

Passons à la pratique :

Allez sur votre branche principale pour modifier des fichiers.



Vous pouvez à tout moment voir l'état dans lequel sont vos fichiers, c'est-à-dire voir les changements qui ont été indexés ou ceux qui ne l'ont pas été, avec la commande suivante :

- `git status`

```

aresv@MSI MINGW64 ~/Documents/test (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
      modified:   PremierFichier.txt

no changes added to commit (use "git add" and/or "git commit -a")

aresv@MSI MINGW64 ~/Documents/test (main)
$ git add PremierFichier.txt

aresv@MSI MINGW64 ~/Documents/test (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   PremierFichier.txt

```

*En rouge, car je n'avais pas fait de git add à mon fichier, il n'était donc pas encore passé en index*

Vous pouvez voir vos fichiers modifiés mais qui n'ont pas encore été commisés.

Ici, nous nous rendons compte que nous avons **modifié la branche main...** Hors vous savez comme moi qu'il est **interdit de modifier la branche main** Sous peine de voir un lanceur de sort (où sysadmin) débarquer dans son bureau armé d'une baie Synology.

Pas de panique, nous n'avons pas encore commit, nous pouvons donc passer la commande **git stash** pour effectuer une remise

```

aresv@MSI MINGW64 ~/Documents/test (main)
$ git stash
Saved working directory and index state WIP on main: 3443bc6 rajout du 1er fichier texte

aresv@MSI MINGW64 ~/Documents/test (main)
$ git status
On branch main
nothing to commit, working tree clean

```

*On s'assure avec un git status nous n'avons plus de fichier dans la zone d'indexion*

Pour **transférer nos modifications** de la branche main vers une nouvelle branche, il faut dans un 1er temps créer une nouvelle branche que nous appellerons pour l'exemple **brancheCommit**:

```

aresv@MSI MINGW64 ~/Documents/test (main)
$ git branch brancheCommit

aresv@MSI MINGW64 ~/Documents/test (main)
$ git branch
  brancheCommit
* main

```

Il nous faut basculer sur cette branche avec la commande **git checkout brancheCommit**

```

aresv@MSI MINGW64 ~/Documents/test (main)
$ git checkout brancheCommit
Switched to branch 'brancheCommit'

aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git branch
* brancheCommit
  main

```

Nous pouvons maintenant appliquer le stash pour :

- **Récupérer les modifications** que vous avez **rangées dans le stash**
- **Appliquer** ces modifications sur votre **nouvelle branche**.

Cette commande va appliquer le dernier stash qui a été fait.

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git stash apply
On branch brancheCommit
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:  franchementCPasOufLeStash

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  PremierFichier.txt
```

Si pour une raison ou une autre, vous avez créé plusieurs stash, et que le dernier n'est pas celui que vous souhaitez appliquer, pas de panique, il est possible d'en appliquer un autre.

En premier lieu, regardez la liste de vos stash avec la commande suivante :

```
git stash list
```

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git stash list
stash@{0}: WIP on main: 3443bc6 rajout du 1er fichier texte
```

Cette commande va vous retourner un "tableau" des stash avec des identifiants du style : il nous suffit ensuite d'appeler la commande `git stash` en indiquant l'identifiant : ici `git stash apply stash@{0}`

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git stash apply stash@{1}
On branch brancheCommit
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  PremierFichier.txt
```

*Le git stash apply stash@1 vient de plusieurs essaie que j'ai effectué, faites très attention avec les stash*

#### Modifier la branche après avoir fait un commit

Maintenant, admettons que vous ayez réalisé vos modifications et qu'en plus vous ayez fait le commit. Le cas est plus complexe, puisque vous avez enregistré vos modifications sur la branche principale, alors que vous ne deviez pas.

Allez-y, modifiez des fichiers, et réalisez le commit.

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:  franchementCPasOufLeStash

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  PremierFichier.txt
```

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git commit -m "On fait des conneries LALALALALALALALALA"
[main 96ddf05] On fait des conneries LALALALALALALALALA
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 franchementCPasOufLeStash
```

Pour réparer cette erreur, vous devez analyser vos derniers commits avec la fonction : `git log` . Vous allez alors récupérer l'identifiant du commit que l'on appelle couramment le hash.

Par défaut, `git log` va vous lister par ordre chronologique inversé tous vos commits réalisés.

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git log
commit 96ddf05429ebb8582e8b5fba78e1e51e5125b4ff (HEAD -> main)
```

Date: Mon Sep 12 12:03:31 2022 +0200

Maintenant que vous disposez de votre identifiant, gardez-le bien de côté. Vérifiez que vous êtes sur votre branche principale et réalisez la commande suivante :

`Git reset --hard HEAD^`

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git reset --hard HEAD^
HEAD is now at 3443bc6 rajout du 1er fichier texte
```

Le dernier commit a été supprimé, l'avant dernier prend donc sa place.

Vous allez maintenant basculer sur cette branche :

`git checkout brancheCommit`

Maintenant, vous êtes sur la bonne branche.

Renouvez la commande `git reset`, qui va appliquer ce commit sur votre nouvelle branche !

*Il n'est pas nécessaire d'écrire l'identifiant en entier. Seuls les 8 premiers caractères sont nécessaires.*

`git reset --hard ca83a6df`

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git checkout brancheCommit
Switched to branch 'brancheCommit'

aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git reset --hard 96ddf054
HEAD is now at 96ddf05 On fait des conneries LALALALALALALALA
```

### Le message de commit est erroné

*!\\ Cette commande ne fonctionne que pour le dernier commit réalisé /\\*

Lorsque l'on travaille sur un projet avec Git, il est très important de marquer correctement les modifications effectuées dans le message descriptif. Cependant, si vous faites une erreur dans l'un de vos messages de commit, il est possible de changer le message après coup avec la commande :

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

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git commit --amend -m "Je m'excuse auprès de la Lynn Corporation"
[brancheCommit ebd0552] Je m'excuse auprès de la Lynn Corporation
Date: Mon Sep 12 12:33:52 2022 +0200
1 file changed, 1 insertion(+)
```

### J'ai oublié un fichier dans mon dernier commit

Imaginons maintenant que vous ayez fait votre commit mais que vous réalisez que **vous avez oublié un fichier**. Ce n'est pas bien grave ! Il suffit de

réutiliser la commande `git --amend`, mais d'une autre manière.

La fonction `git --amend`, si vous avez bien compris, permet de **modifier le dernier commit**.

Réutilisez cette fonction **sans le -m** qui permettait de modifier son message.

Dans un premier temps, **ajoutez votre fichier**, puis réalisez le `git --amend` :

- `git add FichierOublie.txt`
- `git commit --amend --no-edit`

Votre fichier a été ajouté à votre commit et grâce à la commande `--no-edit` que vous avez ajoutée, vous n'avez pas modifié le message du commit.

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git commit --amend --no-edit
[brancheCommit 7089418] Je m'excuse auprès de la Lynn Corporation
  Date: Mon Sep 12 12:33:52 2022 +0200
  2 files changed, 1 insertion(+)
    create mode 100644 lalalalalala.txt
```

`git commit --amend` vous permet de sélectionner le dernier commit afin d'y ajouter de nouveaux changements en attente. Vous pouvez **ajouter ou supprimer des changements afin de les appliquer avec `commit --amend`**.

Si aucun changement n'est en attente, la commande `--amend` vous permet de **modifier le dernier message de log du commit** avec `-m`.

En résumé :

- `git branch -d` permet de **supprimer une branche**.
- `git status` permet de **voir l'état des fichiers**.
- `git stash` enregistre les **modifications non indexées** pour une **utilisation ultérieure**.
- `git log` affiche l'**historique des commits** réalisés sur la **branche courante**.
- `git reset --hard HEAD^` permet de **réinitialiser l'index et le répertoire de travail** à l'**état du dernier commit**.
- `git commit --amend` permet de **sélectionner le dernier commit** pour y **effectuer des modifications**.

## Corrigez vos erreurs sur votre dépôt distant

### Corriger vos erreurs en local et à distance

Par mégarde nous nous rendons compte que nous avons **pushé des fichiers erronées**, comment revenir en arrière ?

Il est possible d'**annuler son commit public** avec la commande `git revert`. L'**opération revert annule un commit en créant un nouveau commit**. C'est une méthode sûre pour annuler des changements, car **elle ne risque pas de réécrire l'historique du commit**.

- `git revert HEAD^`

Nous avons maintenant **reverté notre dernier commit public** et cela a **créé un nouveau commit d'annulation**. Cette commande n'a donc **aucun impact sur l'historique**. Par conséquent, il vaut mieux utiliser `git revert` pour annuler des changements apportés à une branche publique, et `git reset` pour faire de même, mais **sur une branche privée**.

`git revert` sert à **annuler des changements commités**, tandis que `git reset HEAD` permet d'**annuler des changements non commités**.

Toutefois, **attention**, `git revert` peut écraser vos fichiers dans votre répertoire de travail, il vous sera donc demandé de **commiter vos modifications** ou de les **remiser**.

### L'accès distant ne fonctionne pas

Si votre accès à distance ne fonctionne pas, cela peut être dû à un **problème d'authentification de votre réseau**. Pour le résoudre, il vous faut créer une **paire de clés SSH**.

Nous allons maintenant **générer notre duo de clés SSH** :

Dans Git Bash, exécutez la commande :

```
- $ ssh-keygen -t rsa -b 4096 -C "johndoe@example.com"
```

Vous obtenez ceci :

```
$ ssh-keygen -t rsa -b 4096 -C "johndoe@example.com"
Generating public/private rsa key pair.
Enter file in which to save the key </c/Users/tlestroubac.ADGROUP/.ssh/id_rsa>:
```

Vous pouvez soit appuyer sur Entrée, soit indiquer un nom de fichier. Un mot de passe vous est ensuite demandé.

Félicitations ! Vous avez obtenu votre clé SSH !

.gimp-2.8	11/04/2019 09:15	Dossier de fichiers
.git	11/04/2019 10:48	Dossier de fichiers
.ssh	11/04/2019 12:01	Dossier de fichiers
.thumbnails	04/01/2019 09:56	Dossier de fichiers
.VirtualBox	25/03/2019 09:12	Dossier de fichiers
AppData	02/01/2019 09:34	Dossier de fichiers
Bureau	25/03/2019 15:52	Dossier de fichiers
Contacts	02/01/2019 09:34	Dossier de fichiers
Favoris	02/01/2019 09:34	Dossier de fichiers

Pour la trouver, il suffit d'aller à l'adresse : C:\Users\VotreNomD'Utilisateur\, et d'afficher les dossiers masqués.

Dans ce dossier, vous avez donc deux fichiers, **votre clé publique** et **votre clé privée**.

La clé **id\_rsa.txt** est votre **clé privée** alors que la clé **id\_rsa.pub** est votre **clé publique**. Ici nous allons utiliser votre **clé publique seulement**. Vous pouvez copier votre clé publique en l'ouvrant dans un bloc-notes.

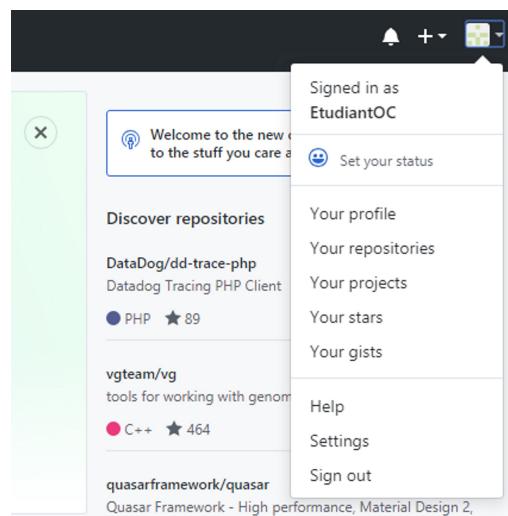


Maintenant que vous disposez de votre clé SSH, voyons comment l'ajouter pour GitHub !

### Modifiez vos informations d'identification et supprimez la clé

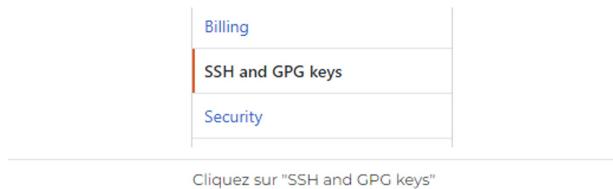
Ajoutez maintenant la clé à votre compte GitHub.

Connectez-vous à votre espace GitHub, puis allez dans l'angle droit de votre compte et cliquez sur Settings.



Espace "Settings" dans votre espace GitHub

Cliquez sur SSH and GPG keys :



Cliquez sur "SSH and GPG keys"

Puis sur New SSH Key :

A screenshot of a GitHub page titled 'SSH keys / Add new'. It shows a 'Title' input field containing 'Personnal Windows' and a 'Key' input field containing a long SSH key. Below the key input is an 'Add SSH key' button.

```
ssh-rsa AAAAB3NzaC1yc2EAAAQEAvgCekDEPhSaFvsl4vGzH+oAAuML+TSCVXWYf7efOrSiNM5qJq6SgC966 qvB4Cz1z/H5uzCp8L/q5x2WeOy+FRpcasIB+6qZrYdCINEpx43A9xEN5E7pc60efu9g3c3iRM7kQC5OgtaPe AD/IGDWquaXctKzxNDz1FavM5STIBA7KJr+VRfopzC4z7gr1sh8RvPsC400VMK7my8bzSja0qhqlEx73xh9fnBERHC g0G2lMahn0DX4okyOhP0k2dYRogWVGobCKWOz2ADA2XSK8FaPIW2mn3MUH31Q2zf1pRK0iwOZp3Y9daXD0XI CyqAXztISU3bmbmgU9+i6yjVTuP+ +xZig6TwQQp5vxXTUAdwrcmA1e1Jv0dgrak3HHbm9Yrgsmzp2w24jEn8xS WWaKuJHQzNmZ30ckMMcasqiz9mkVKWhymhtXtzVHyMz3MkvkjTY84ua6VzMCkkluwiynvspUlnlG6d/TICf2/k OTRYpo7UCro3K+FvnZ4bbBDm2SwggUNJvB5fzoN8EUvBOaS7wOM8k0owMln9tbz8yk9Q0Req5DMd/ZQheTsU Wam1HZNaMTSyOmdpsumNCrX7fK0qD4lkwmv+l2v+MYA0IT2h76FdKVu9QRsuTyhISNI5nf1AD+LxFZeQUBU4E
```

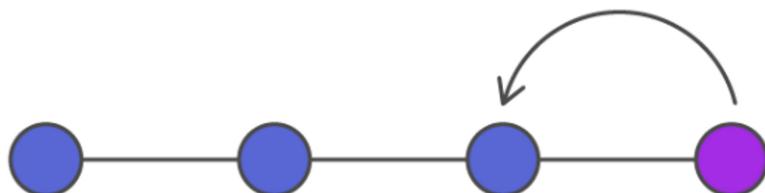
Vous devrez ensuite confirmer votre mot de passe. Votre clé SSH sera alors ajoutée à votre compte GitHub !

En résumé :

- **git revert HEAD<sup>n</sup>** permet d'**annuler un commit** en créant un nouveau commit.
- La commande **ssh-keygen** permet de **générer un duo de clés SSH**.
- Vous pouvez **configurer une nouvelle clé SSH sur GitHub**.

## Utilisez git reset

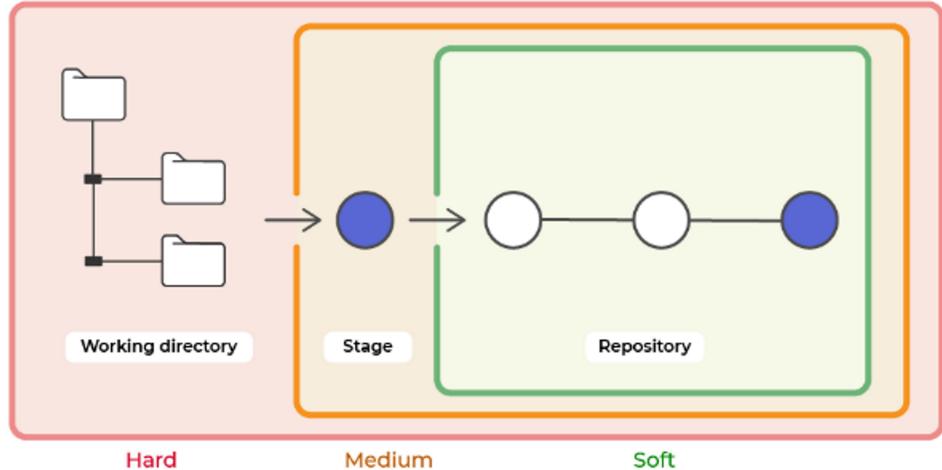
*Si vous développpez une nouvelle fonctionnalité mais que votre client vous explique qu'il n'a plus besoin de cette dite fonctionnalité, vous pouvez utiliser la commande **git reset** pour revenir en arrière*



Revenez en arrière grâce à git reset

## Les 3 types de réinitialisation de GIT

La commande `git reset` est un **outil complexe et polyvalent** pour **annuler les changements**. Elle peut être **appelée de trois façons différentes**, qui correspondent aux arguments de ligne de commande `--soft`, `--mixed` et `--hard`.



Les 3 types de réinitialisation de Git reset

#### La commande `git reset --hard`

Cette commande permet de **revenir à n'importe quel commit mais en oubliant absolument tout ce qu'il s'est passé après** ! Quand je dis **tout, c'est TOUT** ! Que vous ayez fait des modifications après ou d'autres commits, tout sera effacé ! C'est pourquoi il est extrêmement important de revérifier plusieurs fois avant de la lancer, vous pourriez perdre toutes vos modifications si elle est mal faite.

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git reset 3443bc6541 --hard
HEAD is now at 3443bc6 rajout du 1er fichier texte
```

Cette utilisation de `git reset` constitue une manière simple d'**annuler des changements qui n'ont pas encore été partagés**. Cette commande est incontournable lorsque vous commencez à travailler sur une fonctionnalité, que vous vous êtes trompé et que vous voulez **recommencer de zéro**.

#### La commande `git reset --mixed`

La commande `git reset --mixed` va permettre de **revenir juste après votre dernier commit ou le commit spécifié, sans supprimer vos modifications en cours**. Il permet aussi, dans le cas de fichiers indexés mais pas encore commités, de désindexer les fichiers.

Si rien n'est spécifié après `git reset`, par défaut il exécutera un `git reset --mixed HEAD~`.

```
git reset HEAD~
```

Le **HEAD** est un **pointeur**, une **référence sur votre position actuelle dans votre répertoire de travail Git**. Par défaut, **HEAD pointe sur la branche courante, main/master**, et peut être déplacé vers une autre branche ou un autre commit.

#### La commande `git reset --soft`

Nous avons enfin `git reset --soft`. Cette commande permet de **se placer sur un commit spécifique afin de voir le code à un instant donné, ou de créer une branche partant d'un ancien commit**. Elle ne supprime aucun fichier, aucun commit, et ne crée pas de HEAD détaché.

#### Le cas des conflits

*Nous avons vu dans la deuxième partie de ce cours comment fusionner des branches, en utilisant un exemple assez simple où tout s'est bien terminé. Malheureusement, il arrive parfois, même souvent, que cela ne se passe pas aussi bien, et que des conflits apparaissent.*

Si nous reprenons notre exemple de début de cours, **vous avez travaillé sur la branche "ameliorationCagnotte" alors que des fichiers correspondant à l'amélioration de la fonctionnalité "cagnotte" existent déjà dans la branche principale, et que vous modifiez des lignes déjà en place.**

**Vous avez modifié du code** pour afficher le message "Une super cagnotte !" alors qu'il était déjà en place le message "Une cagnotte". Lorsque vous allez fusionner les deux branches, les choses ne vont donc pas très bien se passer :

```
git checkout main git merge ameliorationCagnotte Auto-merging cagnotte.php CONFLICT  
(content): merge conflict in cagnotte.php Automatic merge failed; fix conflicts and  
then commit the result
```

Git va voir que **sur la même ligne on essaie de fusionner deux choses différentes**. Il ne va pas pouvoir deviner laquelle prendre, la ligne "Une cagnotte", ou bien "Une super cagnotte" ?

Git va donc afficher un conflit sur le fichier cagnotte.php et arrêtera le processus de fusion ou merge. Ce conflit, vous allez devoir le résoudre en ouvrant le fichier avec votre éditeur habituel :

```
<<<<< HEAD Une cagnotte ====== Une super cagnotte ! >>>>> ameliorationCagnotte
```

Maintenant, réglez les conflits en **comparant les deux lignes et en choisissant quelle modification vous voulez garder**. Ici, il faut garder "Une super cagnotte !", **on va donc supprimer les autres lignes** et ne garder que celle-ci :

**Une super cagnotte !**

Maintenant que vous avez résolu le conflit, il vous reste à le dire à Git :

- git add cagnotte.php git commit

Git va détecter que vous avez résolu les conflits et va vous proposer un message de commit.

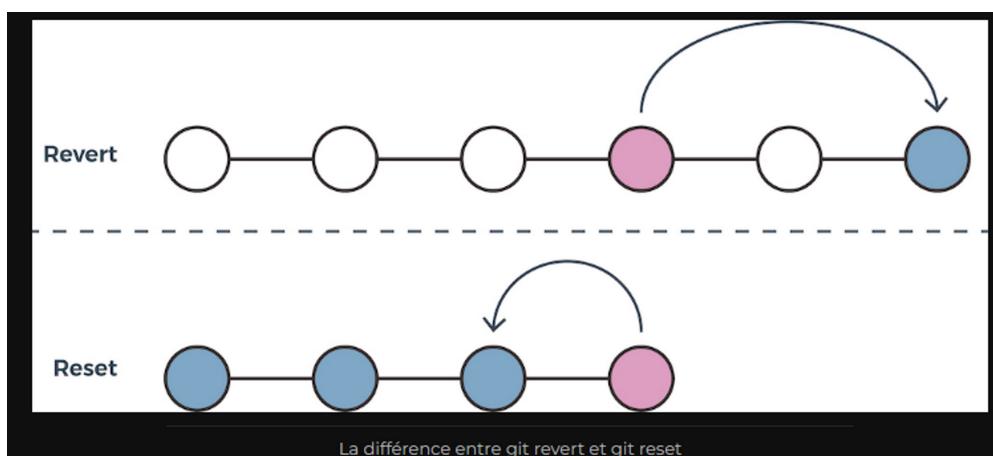
### Rajout du mauvais fichier au commit

En cas de fichier non désiré rajouté au commit, il est possible d'utiliser la commande **git revert**

La commande **git revert** vous permet de revenir à l'état précédent, tout en faisant un deuxième commit. Au lieu de supprimer le commit de l'historique du projet, elle détermine comment annuler les changements introduits par le commit et ajoute un nouveau commit avec le contenu ainsi obtenu. Vous allez donc revenir à l'état précédent mais avec un nouveau commit. Ainsi, **Git ne perd pas l'historique**, lequel est important pour l'intégrité de votre historique de révision et pour une collaboration fiable.

Quelle est la différence entre **git reset** et **git revert** ?

- **git reset** va revenir à l'état précédent sans créer un nouveau commit.
- **git revert** va créer un nouveau commit.



Essayons cette commande en faisant un **premier commit** que nous allons finalement **ne plus vouloir**. Une fois votre commit fait, écrivez la commande suivante :

- `git revert HEAD`

Une fois votre commit "annulé", vous pouvez enlever votre fichier et réaliser de nouveau votre commit.

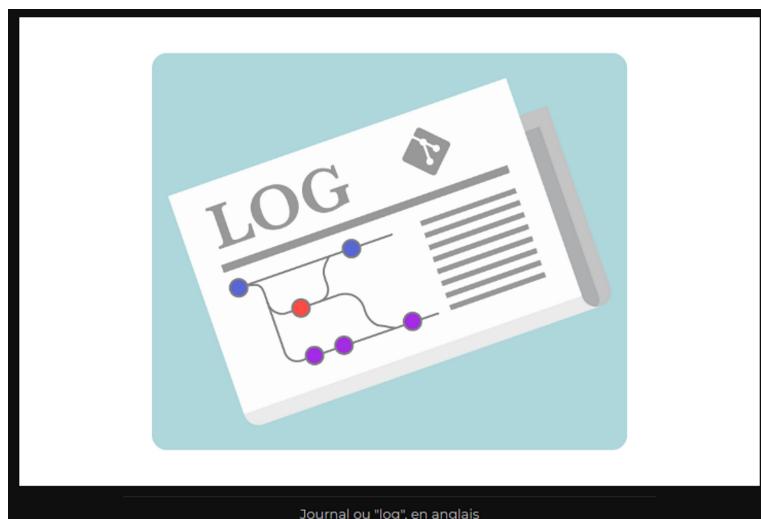
En résumé :

- `git reset` est une commande puissante. Elle peut être appliquée de **3 façons** différentes (`--soft`; `--mixed`; `--hard`).
- La commande `git merge` produit un **conflit** si une même **ligne** a été modifiée plusieurs fois. Dans ce cas, il faut indiquer à Git quelle ligne conserver.
- `git reset` permet de revenir à l'état précédent sans créer un nouveau commit.
- `git revert` permet de revenir à l'état précédent en créant un nouveau commit.

## Corriger un commit raté

Pendant un projet, il peut-être nécessaire de revenir à certaines versions ou encore de savoir qui a fait le commit : Git est un outils possédant des techniques de **journalisation** pour répondre à ce type de demande :

- La **journalisation**, ou **history** en anglais, désigne l'enregistrement dans un fichier ou une base de données de **tous les événements affectant une application**. Le **journal** (en anglais **log file** ou plus simplement **log**), désigne alors le fichier contenant ces enregistrements.



## Git reflog

L'objectif d'un gestionnaire de versions est d'**enregistrer les changements apportés à votre code**. Il vous permet de **consulter l'historique de votre projet** pour :

- savoir qui a contribué à quoi
- déterminer où des bugs ont été introduits
- annuler les changements problématiques.

Par défaut, `git log` énumère en **ordre chronologique inversé** les commits réalisés. Cela signifie que les **commits les plus récents apparaissent en premier**. Cette commande affiche chaque commit avec son **identifiant SHA**, l'auteur du commit, la date et le message du commit.

Le **SHA** ou **Secure Hash Algorithm** est un **identifiant**. C'est ce grand code incompréhensible qui nous permettra de **revenir en arrière si besoin, à un commit exact**.

Git dispose d'un outil encore plus puissant, poussant le journal de logs à l'extrême :

- `git reflog`

**git reflog** va loguer les commits ainsi que toutes les autres actions que vous avez pu faire en local : vos modifications de messages, vos merges, vos resets, enfin tout, quoi . Comme `git log` , `git reflog` affiche un **identifiant SHA-1** pour chaque action. Il est donc très facile de revenir à une action donnée grâce au SHA. Cette commande, c'est votre joker, elle assure votre survie en cas d'erreur :

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
$ git reflog
8b595fe (HEAD -> brancheCommit) HEAD@{0}: commit: Le test
1082739 HEAD@{1}: commit: je déteste les gens
33d650e HEAD@{2}: commit: apparemment j'aime les lalafell
3443bc6 (main) HEAD@{3}: reset: moving to 3443bc6541
7089418 HEAD@{4}: commit (amend): Je m'excuse auprès de la Lynn Corporation
e1353d6 HEAD@{5}: commit (amend): Je m'excuse auprès de la Lynn Corporation
ebd0552 HEAD@{6}: commit (amend): Je m'excuse auprès de la Lynn Corporation
```

Pour revenir à une action donnée, on prend les **8 premiers caractères de son SHA** et on fait : `git checkout` numéro du SHA

```
aresv@MSI MINGW64 ~/Documents/test (brancheCommit)
git checkout 1082739
Note: switching to '1082739'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 1082739 je déteste les gens
```

## Git blame

La commande `git blame` permet d'examiner le contenu d'un fichier ligne par ligne et de déterminer la date à laquelle chaque ligne a été modifiée, et le nom de l'auteur des modifications

- `git blame monFichier.php`

`git blame` va afficher pour chaque ligne modifiée :

- son ID
- l'auteur
- l'horodatage
- le numéro de la ligne
- le contenu de la ligne

## Git cherry-pick

Lorsque vous travaillez avec une équipe de développeurs sur un projet de moyenne à grande taille, la gestion des modifications entre plusieurs branches de Git peut devenir une tâche complexe. Parfois, vous ne voulez pas fusionner une branche entière dans une autre et vous n'avez besoin que de choisir **un ou deux commits spécifiques**. Ce processus s'appelle **cherry-pick** !

Cette commande va permettre de sélectionner **un ou plusieurs** commits grâce à leur SHA et de les **migrer sur la branche principale**, sans pour autant **fusionner toute la branche "Mes évolutions"**.

- `Git cherry-pick numéro de sha de la modification à apporter`

```
aresv@MSI MINGW64 ~/Documents/test (main)
$ git cherry-pick 3443bc6541732fa7c
```

Ici, nous prenons les deux commits ayant pour SHA d356940 et de966d4, et nous les **ajoutons à la branche principale** sans pour autant les enlever de votre branche actuelle. **Nous les dupliquons !**

En résumé :

- **git log** affiche l'**historique des commits réalisés sur la branche courante**.
- **git reflog** est identique à **git log**. Cette commande affiche également toutes les **actions réalisées en local**.
- **git checkout un\_identifiant\_SHA-1** permet de **revenir à une action donnée**.
- **git blame** permet de **savoir qui a réalisé telle modification dans un fichier**, à quelle date, ligne par ligne.
- **git cherry-pick un\_identifiant\_SHA-1 un\_autre\_identifiant\_SHA-1** permet de **sélectionner un commit et de l'appliquer sur la branche actuelle**.