Formation Git : Manipulations pratiques

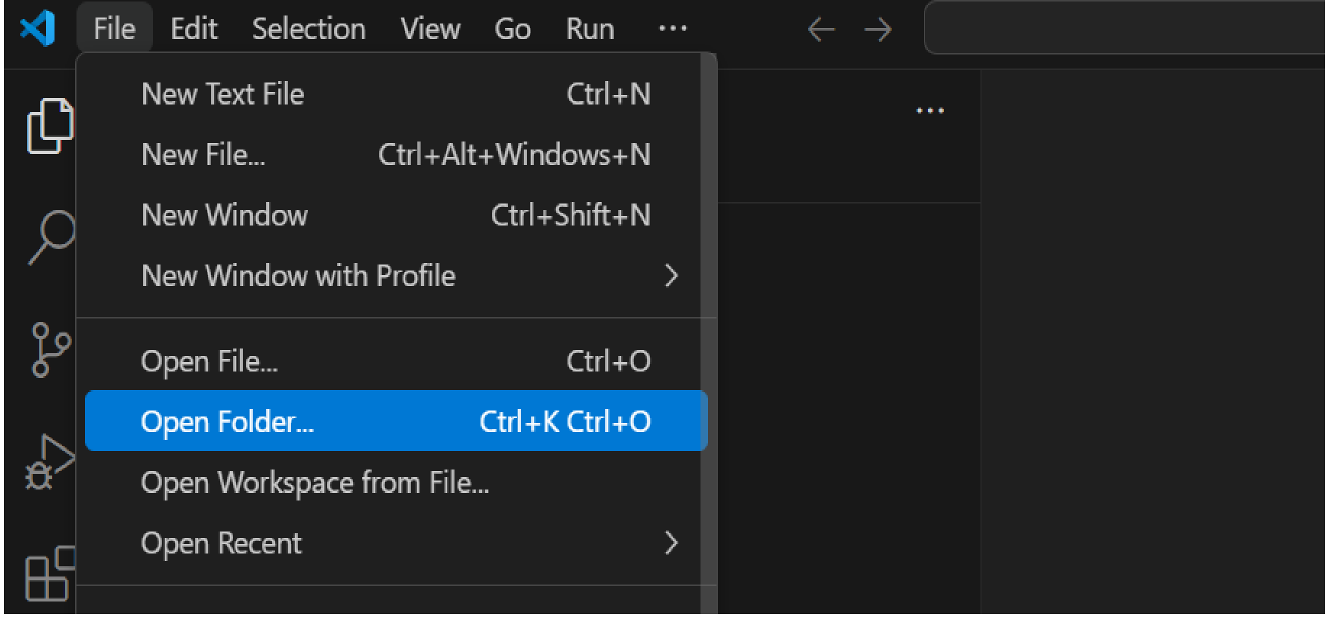
El Hadji Gaye

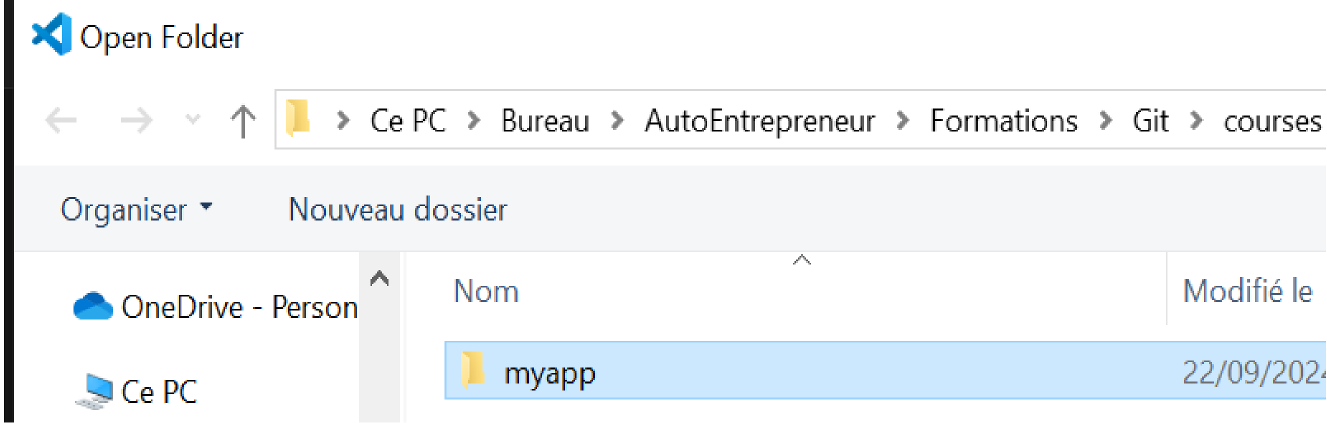
Auteur	El Hadji Gaye
Pour	Formation
Date	30/10/2024
Objet	Formation Git : Manipulations pratiques

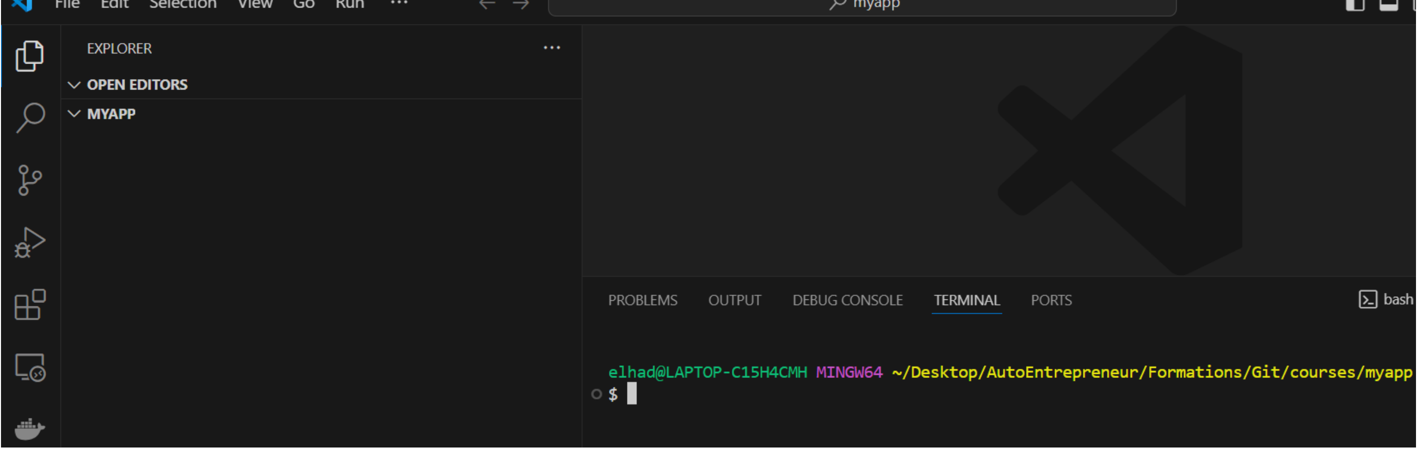
I) Configuration initiale de Git 3
II) Fonctionnement de Git 8
1. Comment est réalisé le suivi des fichiers dans Git 8
2. Le Hashage SHA-1 13
3. Etat d'un fichier et index avec Git 15
4. Comment ignorer des fichiers et des dossiers 21
5. Voir les differences entre repertoire 25
6. Gestion d'un commit Git 27
7. Suppression de fichiers avec Git 35
8. Historique avec git log 38
9. Historique avec git blame 41
10. Branche master et HEAD 44
11. Commande git checkout 50
12. Commande git clean 58
13. Commande git revert 61
14. Commande git reset 66
III) Les branches avec Git 79
1. Introduction 79
2. Lister et créer des branches 86
3. Basculement de branche avec git checkout 96
4. Fusionner des branches avec git merge 107
5. Conflit entre branches Git 118
6. Rectifier un commit 122
7. La commande git rebase 125
IV) Les répertoires distants avec Gitlab 134
1. Introduction 134
2. Pourquoi choisir de Gitlab 135
3. Création d'un compte Gitlab 136
4. Cloner un repertoire distant 144
5. Mise à jours pointeurs distants avec git fetch 154
6. La commande git pull 156
7. La commande git push 159
8. Protocole ssh avec Git et Gitlab 160
V) Git-Flow : workflow d'entreprise 167
1. Introduction 167
2. Les branches éternelles 168
3. Les branches éphémères 169

I) Configuration initiale de Git

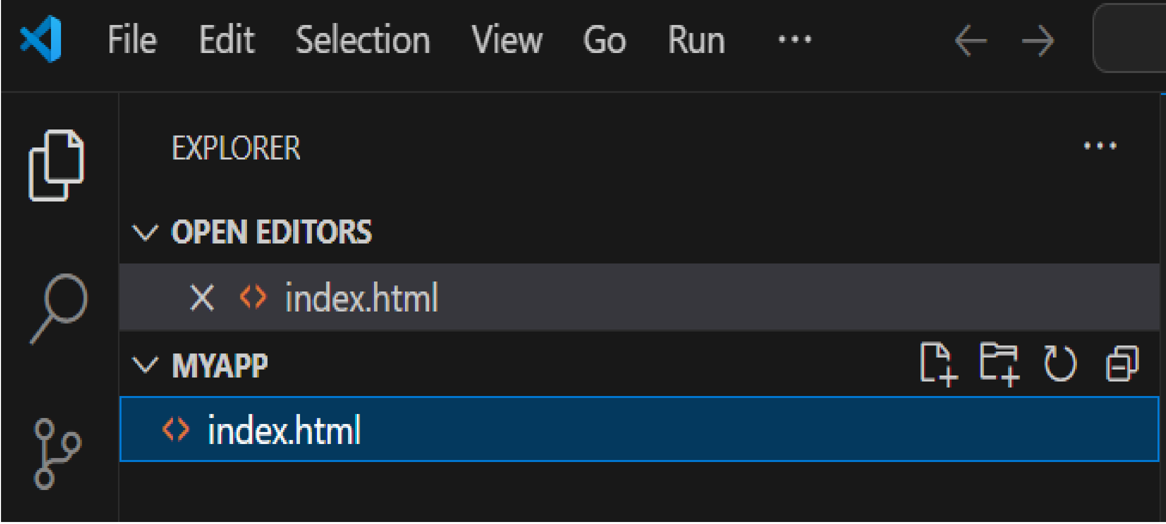
Créer le dossier **Git/courses/myapp** puis positinner votre visual studio dessus :

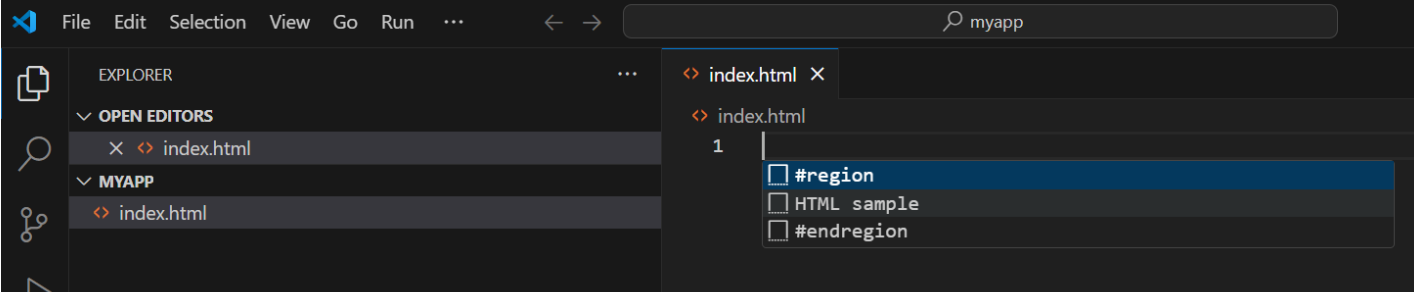






Créer le fichier **Git/courses/myapp/index.html**





Le fichier **index.html** peut avoir comme contenu :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Page Title</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
</head>
<body>

</body>
</html>
```

Se placer dans le repertoire **Git/courses/myapp**, puis lancer les commandes :

ls
ll

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS                    bash

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp
$ ls
index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp
$ ls -l
total 1
-rw-r--r-- 1 elhad            15:44 index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp
$ ll
total 1
-rw-r--r-- 1 elhad            15:44 index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp
```

Lancer la commande ci-dessous :

git init

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    GITLENS        bash

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp
$ git init
Initialized empty Git repository in C:/Users/elhad/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.gi
t/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

Puis lancer la commande :

ls -a

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ ls -a
./  ../  .git/  index.html
```

On voit bien la création du fichier **.git**.

cd .git

Puis

ll

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ ll
total 7
-rw-r--r-- 1 elhad                    config
-rw-r--r-- 1 elhad                    description
-rw-r--r-- 1 elhad                    HEAD
drwxr-xr-x 1 elhad                    hooks/
drwxr-xr-x 1 elhad                    info/
drwxr-xr-x 1 elhad                    objects/
drwxr-xr-x 1 elhad                    refs/
```

On va maintenant s'interessé au dossier **config** qui contient toute la configuration **Git** de notre repos **myapp**.

Affichons le contenu de la configuration avec :

cat config

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ cat config
[core]
        repositoryformatversion = 0
        filemode = false
        bare = false
        logallrefupdates = true
        symlinks = false
        ignorecase = true
```

Vous trouverez le fichier de configuration global de Git sur le dossier **C:/Users/myUser/.gitconfig**.

On peut aussi récupérer l'ensemble des configurations par l'intermédiaire de la commande :

git config --list

git config --global user.name ElHadji

En faisant la commande ci-dessous on peut voir

git config --global --list
git config --list

Renseigner maintenant l'email avec la commande :

git config --global user.email elhadji.gaye83@gmail.com

git config user.name

git config user.email

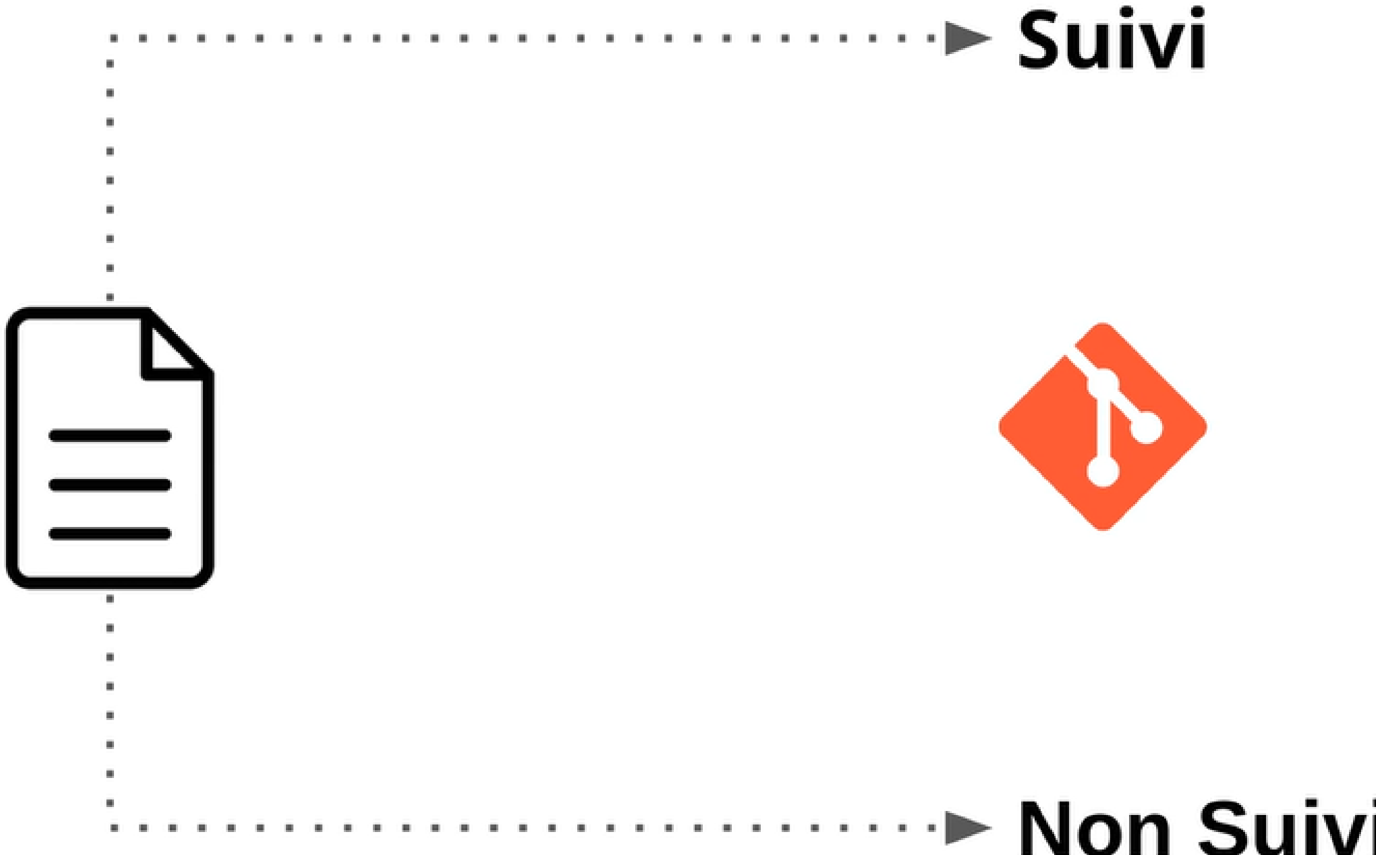
```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git config --global user.email elhadji.gaye83@gmail.com

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git config user.email
elhadji.gaye83@gmail.com

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git config user.name
ElHadji

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

II) Fonctionnement de Git

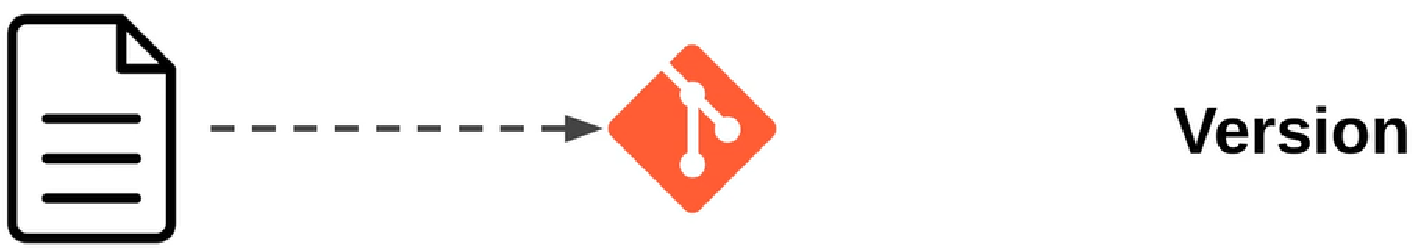
1. Comment est réalisé le suivi des fichiers dans Git

Lorsque vous créer un fichier dans votre repos Git par défaut il n'est pas suivi.

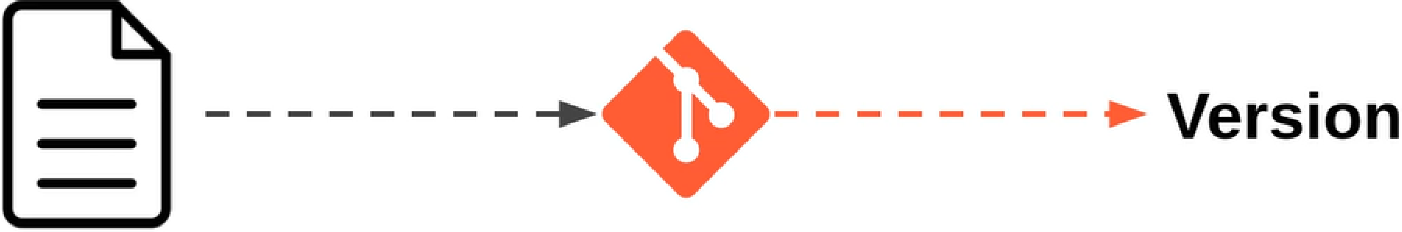
Dans Git vous pouvez avoir deux état pour un fichier **Suivi** ou **Non Suivi**.



Nous avons créer notre fichier index.html et nous voulons qu'il soit suivi :



Pour suivre notre fichier nous allons devoir créer une première version du fichier **index.html**.



Comment va-t-on maintenant passer d’un fichier Non Suivi à fichier Suivi ?

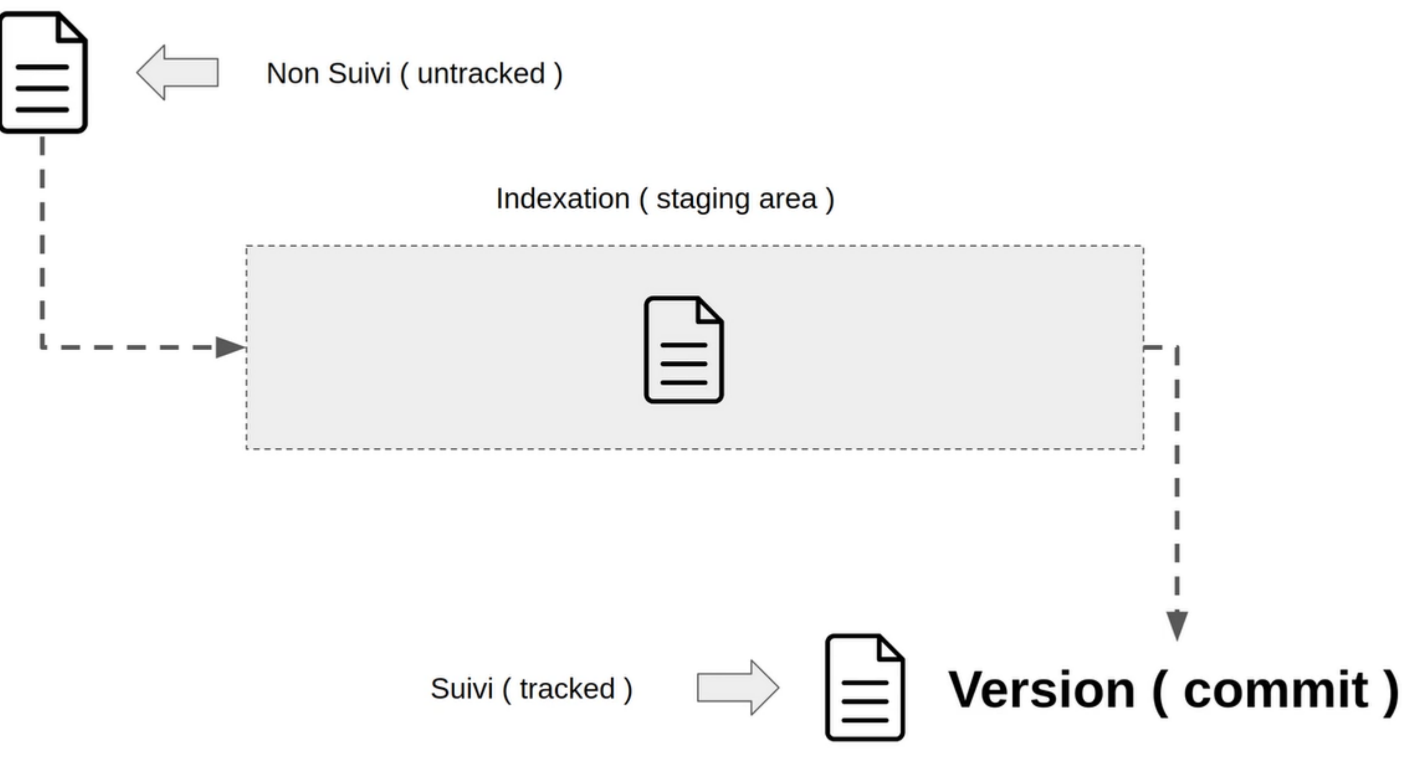
Cela necessite de passer par une étape d’indexation (staging area)

Non Suivi (untracked)

Indexation (staging area)

Non Suivi (untracked)

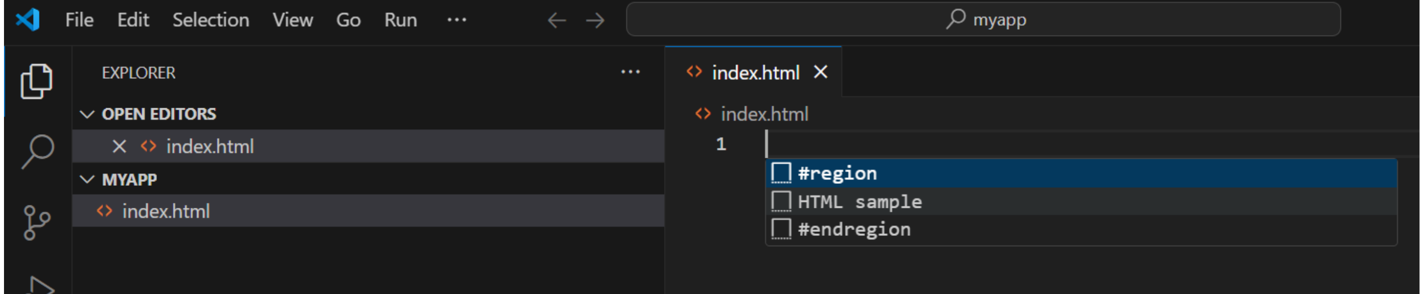
Indexation (staging area)



EL Hadji Gaye

Passons maintenant à la pratique.

Changer le contenu du fichier **Git/courses/myapp/index.html**



Le fichier **index.html** peut avoir comme contenu :

```
<!DOCTYPE html>
<html>
<head>
   <meta charset='utf-8'>
   <meta http-equiv='X-UA-Compatible' content='IE=edge'>
   <title>Page Title</title>
   <meta name='viewport' content='width=device-width, initial-scale=1'>
</head>
<body>

</body>
</html>
```

Se placer dans le repertoire **Git/courses/myapp**

Lancer la commande :

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (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)

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

Pour ajouter le fichier index.html il suffit de faire :

git add index.html

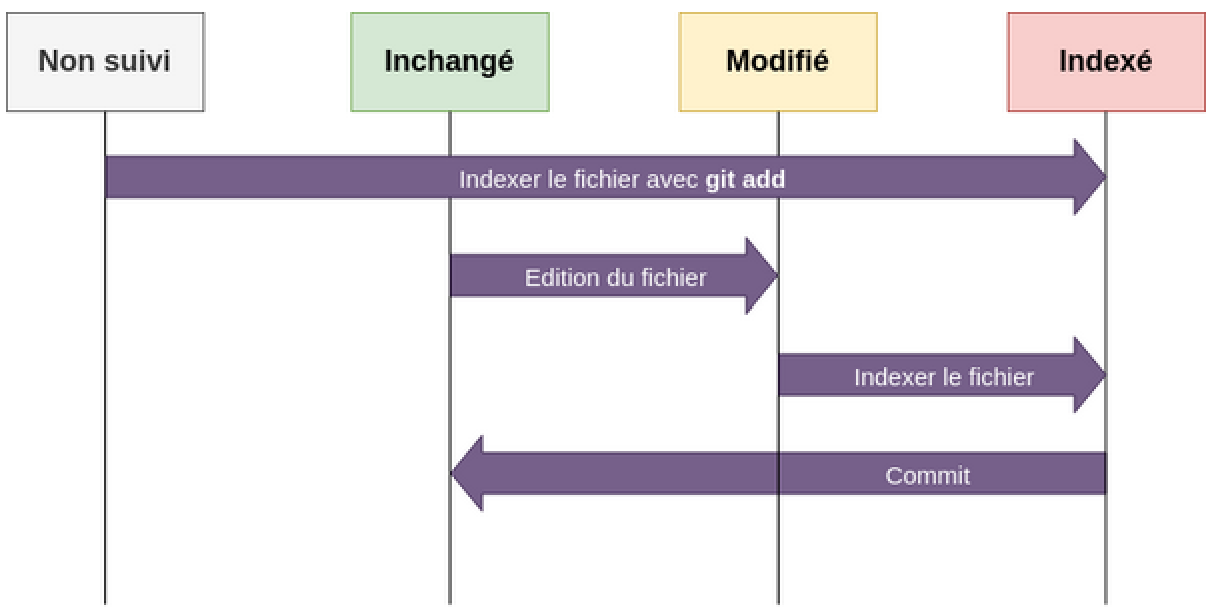
ou encore

git add .

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master

No commits yet

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

git commit -m "first commit"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "first commit"
[master (root-commit) 1104560] first commit
 1 file changed, 12 insertions(+)
 create mode 100644 index.html
```



2. *Le Hashage SHA-1*

Une histoire de hash

Un hash (qui peut également être appelé un condensat ou une signature) est une valeur calculée à partir d'une autre valeur. Dans la grande majorité des cas, cette valeur est représentée sous forme d'une chaîne de caractères hexadécimaux. Le calcul du hash fait appel à un algorithme complexe.

Voici deux exemples de hash calculés avec l'algorithme SHA-1 :

Valeur	Hash
Git	5819778898df55e3a762f0c5728b457970d72cae
git	46f1a0bd5592a2f9244ca321b129902a06b53e03
Je veux une phrase assez longue, au moins plus que le hash en tous cas	7a5a57cde20a9bbda76b70e9223292ce7f8472f9

Dans cet exemple, nous remarquons deux choses :

- Un changement mineur dans le contenu change totalement le hash. Nous remarquons cela en comparant les hash de « Git » et « git ».

- Un hash fait toujours la même taille : 40 caractères (ce qui équivaut à 160 bits).

Il est impossible de retrouver le contenu original à partir du hash. Au mieux on peut essayer de le deviner, mais on ne peut avoir aucune certitude étant donné qu'un même hash peut correspondre à différentes chaînes. En effet, si on calcule le hash de (2 puissance 160)+1 chaînes différentes, on a forcément au moins une chaîne qui partage le hash d'une autre.

Les hashs sont souvent utilisés pour vérifier qu'un fichier n'est pas corrompu ou alors pour authentifier un utilisateur sans devoir stocker son mot de passe en clair.

Une identification par contenu

En interne, Git travaille sur un certain nombre d'objets (contenus dans le dossier .git/objects) : des fichiers, des dossiers, des commits, etc.

Tous les éléments que Git manipule sont en réalité rangés dans des dictionnaires de paires clé/valeur dont la clé est le hash calculé en fonction du contenu. En réalité, Git permet de stocker des informations (comme le contenu d'un fichier) et nous donne un identificateur (le hash) nous permettant de récupérer ces données.

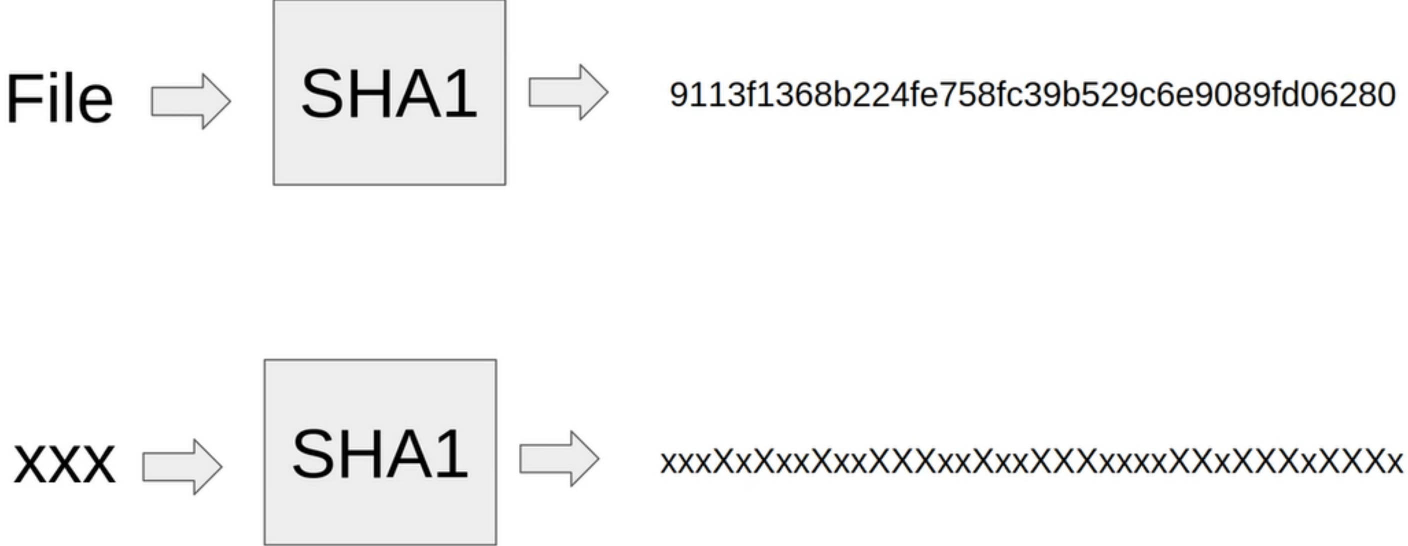
C'est-à-dire que Git ne va pas identifier un fichier **index.html** en fonction de son nom, mais plutôt à partir du hash généré à partir de son contenu. Cette méthode permet à Git de détecter facilement la moindre modification dans un fichier.

Risque de collision

De nombreux développeurs qui débutent avec Git se disent "Si un hash est noté avec 160 bits quel que soit le contenu en entrée alors il y a un risque de tomber sur un doublon". D'ailleurs avec toutes les versions différentes de fichiers et tous les commits, cela peut faire un grand nombre d'objets Git. Ce risque est réel mais extrêmement minime. Tellement minime qu'il est réellement négligeable même pour les projets colossaux.

3. *Etat d'un fichier et index avec Git*

Grâce à un Algorithme de Hashage git va pouvoir affecté pour chacune de ces fichiers une clés de Hashage :



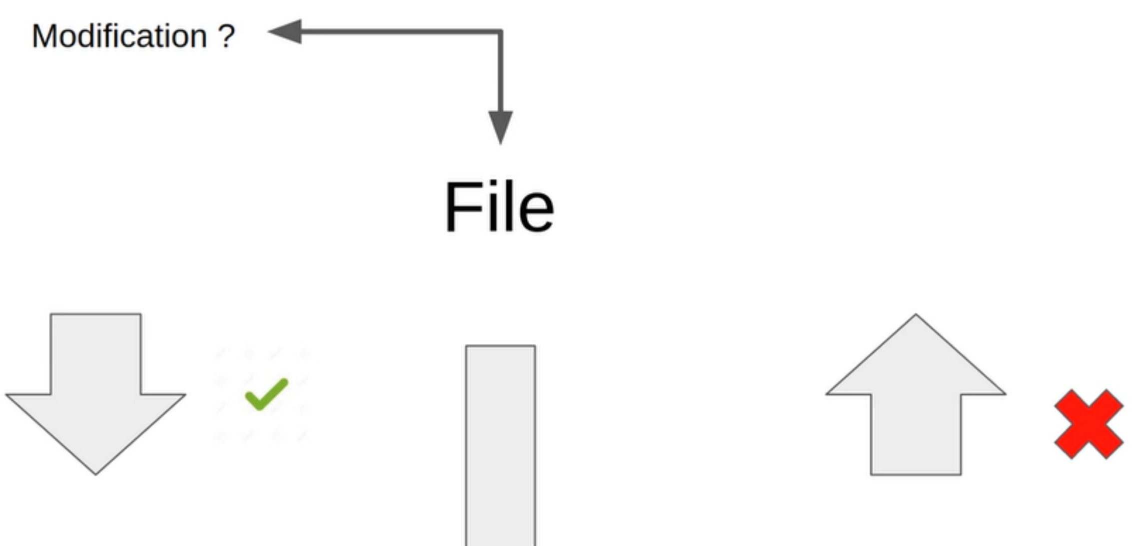
Avec Git chaque version d'un fichier aura une clés de Hashage :

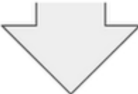
9113f1368b224fe758fc39b529c6e9089fd06280

Il faut noter cependant qu'il est impossible de reconstituer un fichier par l'intermediaire de son Hash.

9113f1368b224fe758fc39b529c6e9089fd06280

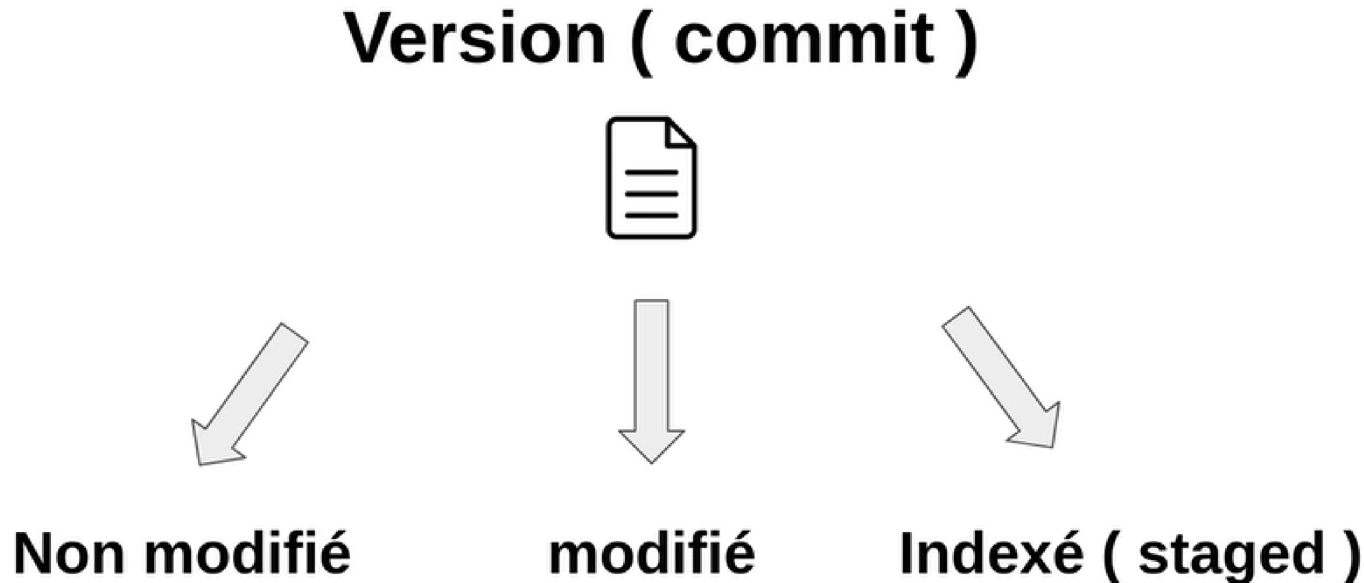
Comme à chaque modification le hash du fichier change alors Git va pouvoir detecter toutes les modifications dans notre repos.





3223f1368b224fe758fc39b529c6e9089fd06280

Nous allons maintenant parler des trois états d'un fichier après un commit :

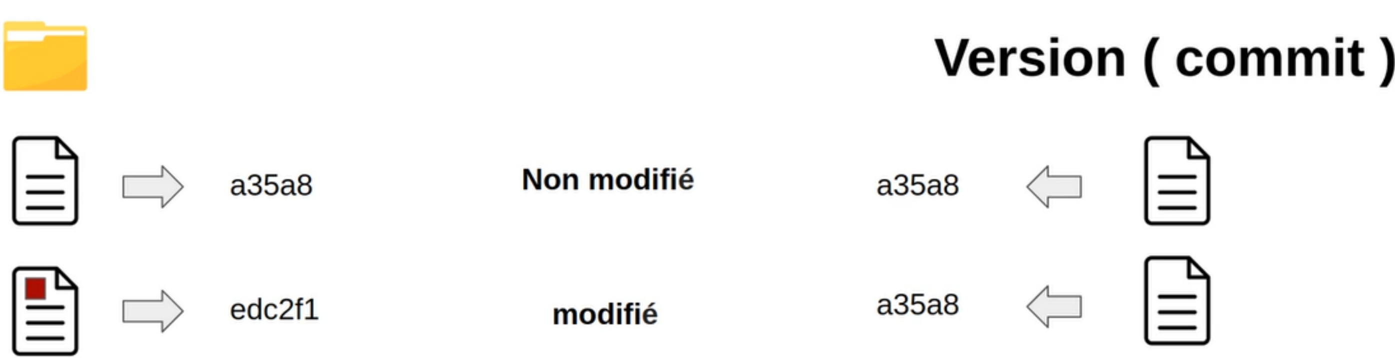


Revenons maintenant sur l'était de notre repos juste après notre dernier commit :

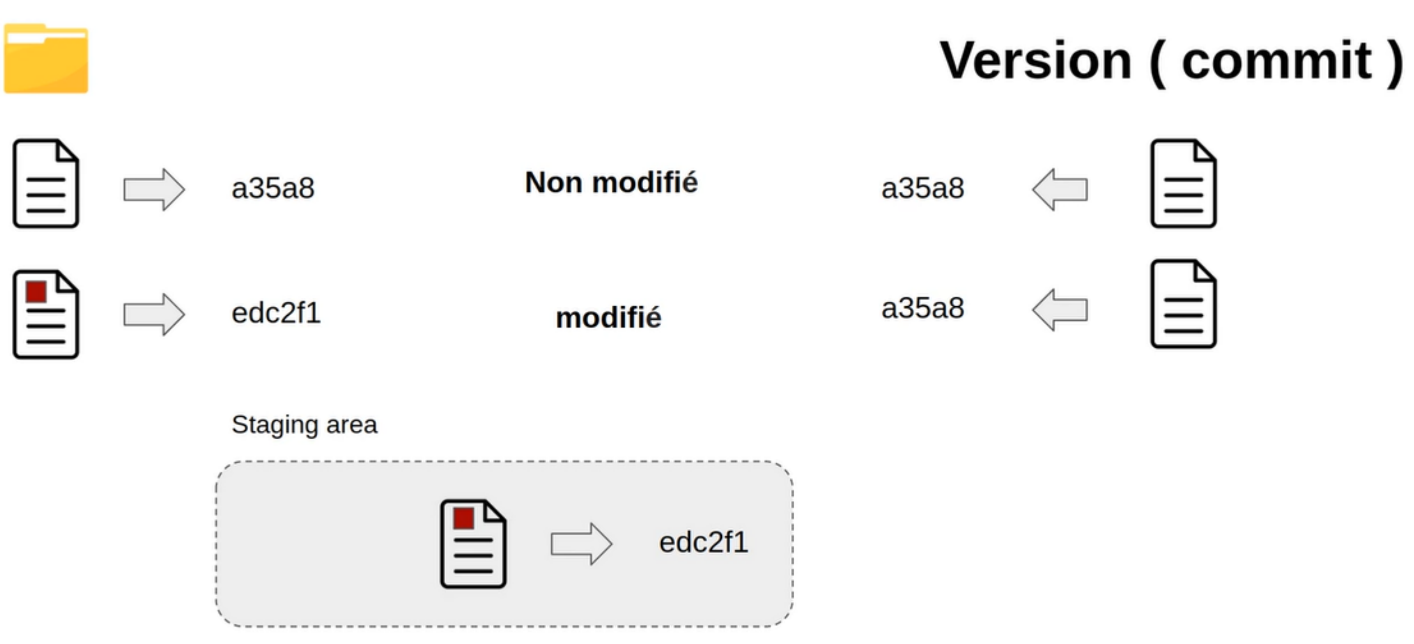
Version (commit)

Juste après le commit tous les fichiers de notre repos ont le même Hash.

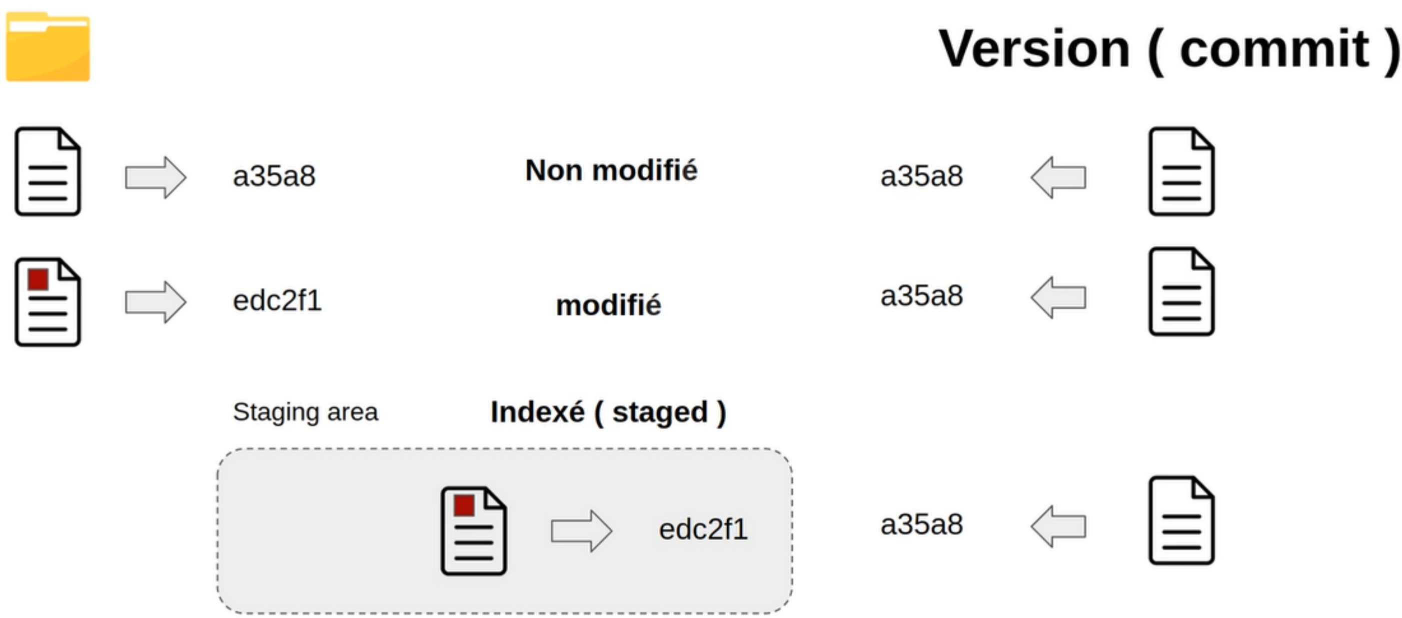
Juste après une modification du fichier le Hash change et git sais le fichier a été modifié :



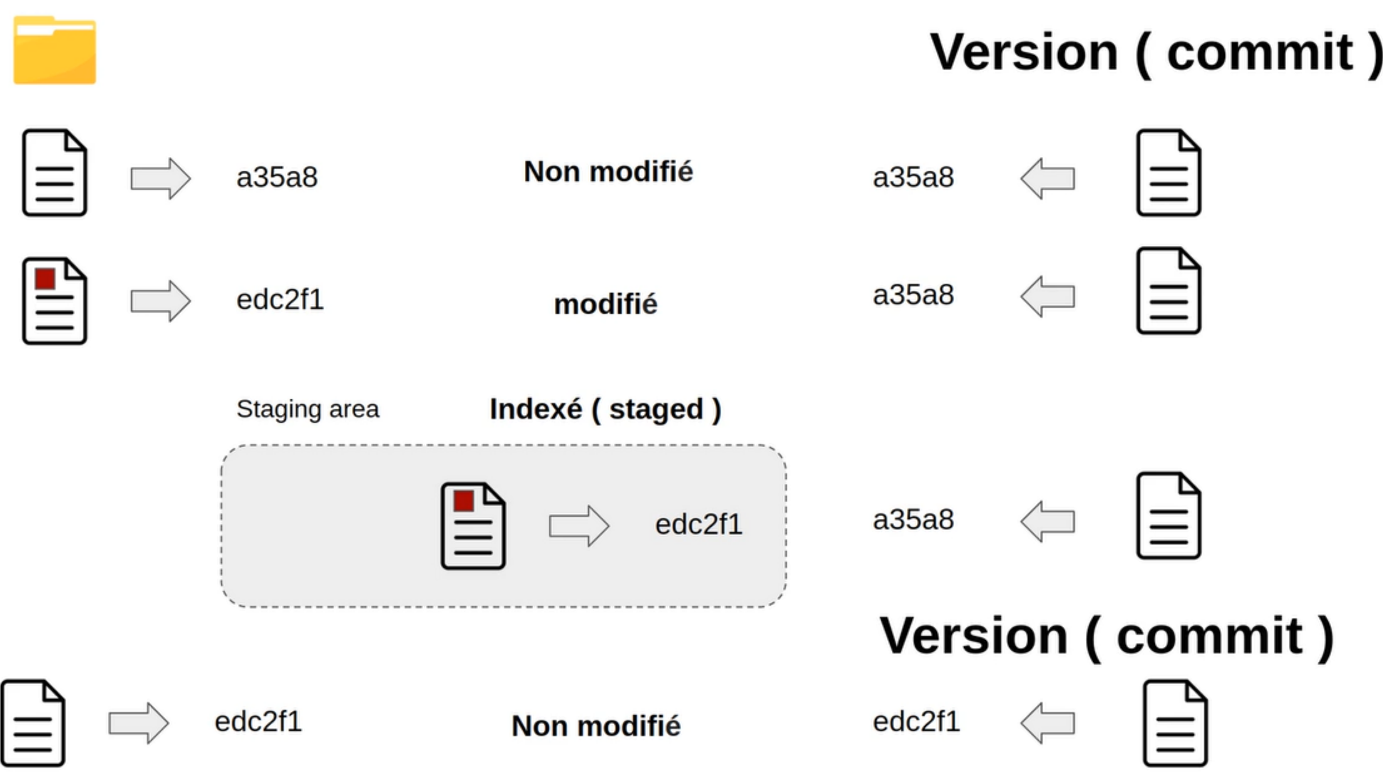
Par l'intermediaire de la commande git add nous allons ajouter ce fichier modifié dans le staging area :



Le fichier est maintenant indexé :



Apres le commit on obtient :



Passons maintenant à la pratique avec notre projet :

cd Git/courses/myapp

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

Modifions le fichier **index.html** par exemple juste par son titre qui devient :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Modified Page Title</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
  </head>
  <body>

  </body>
</html>
```

Refaire un autre status :

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (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

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

Index.html est en rouge car il n'a pas été ajouté dans la zone de staging.

git add index.html
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
```

Enfin le commit :

git commit -m "second commit"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "second commit"
[master c8d70e1] second commit
 1 file changed, 1 insertion(+), 1 deletion(-)

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

4. Comment ignorer des fichiers et des dossiers

Il existe plusieurs types de fichiers qui ne doivent pas être commités. Voici une liste non exhaustive des fichiers qui ne doivent pas être versionnés :

- les fichiers de logs,
- les fichiers résultants d'une compilation,
- les fichiers des bibliothèques
- les fichiers de configuration (surtout lorsqu'ils contiennent des données sensibles).

Ces fichiers seront spécifiés dans le fichier .gitignore qui sera stocké à la racine du dépôt. Les règles de syntaxe du fichier .gitignore sont les suivantes :

- Les lignes vides sont ignorées. Elles peuvent donc servir de séparateur pour aérer le fichier.
- Les lignes débutant par un dièse # sont considérées comme des commentaires et n'ont aucune incidence sur les fichiers ignorés.
- Une ligne contenant un chemin complet suivi du nom d'un fichier exclut uniquement le fichier ciblé.
- Il est possible d'utiliser un ou plusieurs astérisques pour spécifier plusieurs noms de fichier à l'instar de ce qui est possible avec une interface en ligne de commande.
- Une ligne composée uniquement d'un nom de fichier ou de dossier exclut les fichiers portant ce nom quel que soit le dossier dans lequel ils se trouvent. Pour spécifier un fichier se situant à la racine du dépôt, il faut le préfixer par un slash comme si le chemin absolu du fichier débutait au niveau du dépôt.
- Un point d'exclamation en début de ligne signifie que les fichiers ciblés ne seront pas ignorés.

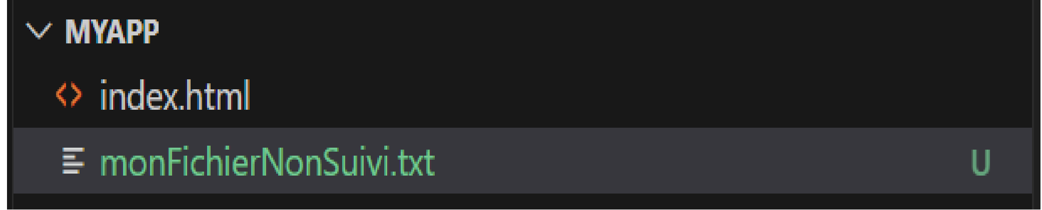
En JAVA par exemple il n'est pas nécessaire de sauvegarder les fichiers d'extension .class qui sont généré à la compilation à partir de fichier .java.

Pour empêcher la sauvegarde de ce type de fichier, nous allons le spécifier dans le fichier gitignore. Ci-dessous un exemple de fichier **.gitignore** simple :

```
# Ignorer tous les dossiers lib (qui contiennent des bibliothèques)
lib/
# Autoriser le versionnement des bibliothèques internes
!lib/interne/
# Ignorer tous les exports de documentation, mais pas les originaux
/documentation/*
!/documentation/*.md
```

Le fichier **.gitignore** doit être enregistré à la racine du dépôt pour être interprété par Git. Ce fichier sera versionné, car il correspond à toutes les exclusions qui doivent être faites pour le projet.

Créer le fichier **monFichierNonSuivi.txt** qui sera un fichier de notre repos Git mais qu'on ne veut pas suivre.



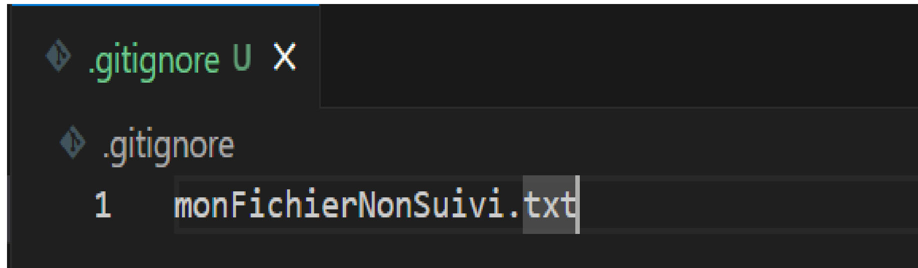
touch .gitignore

Faire un git status pour faire un peu l'état des lieux :

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        monFichierNonSuivi.txt

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

Modifions le le fichier .gitignore pour mettre notre fichier **monFichierNonSuivi.txt à ignorer :**

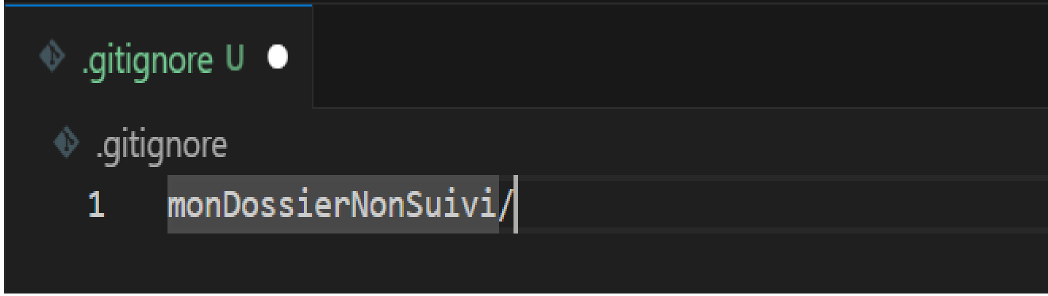


Sauvegarder puis refaire un git status

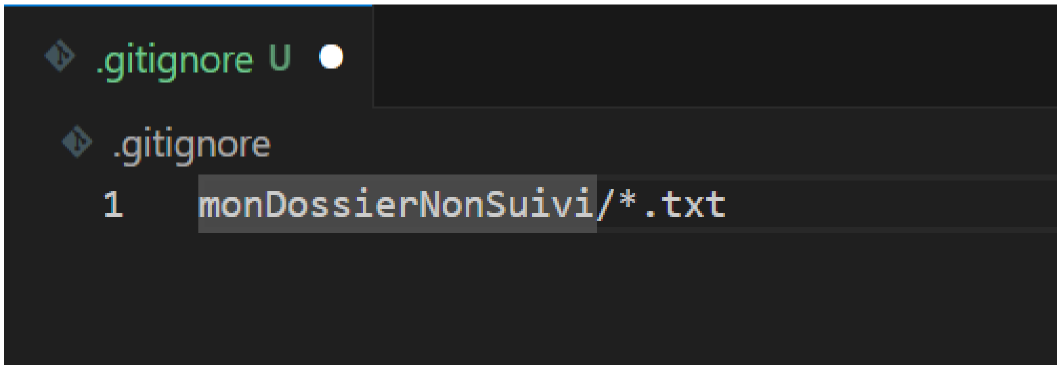
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

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

Dans le fichier .gitignore on mettre un dossier entier à exclure :



On peut aussi utiliser des **wildcard**



On peut aussi utilisé un **wildcard** recursif :

```
.gitignore U
.gitignore
1   monDossierNonSuivi/**/*.txt
```

git commit -m "commit gitignore"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "commit gitignore"
[master 40606b1] commit gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Le site internet https://www.toptal.com/developers/gitignore permet de générer un fichier gitignore en fonction du langage de programmation.

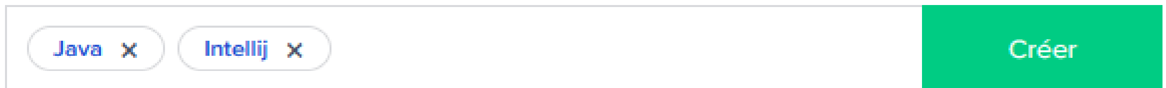
Créez des fichiers .gitignore utiles à votre projet

Recherchez des Systèmes d'Exploitation, IDEs ou Langages de Programmatic | Créer

Code Source | Documentation

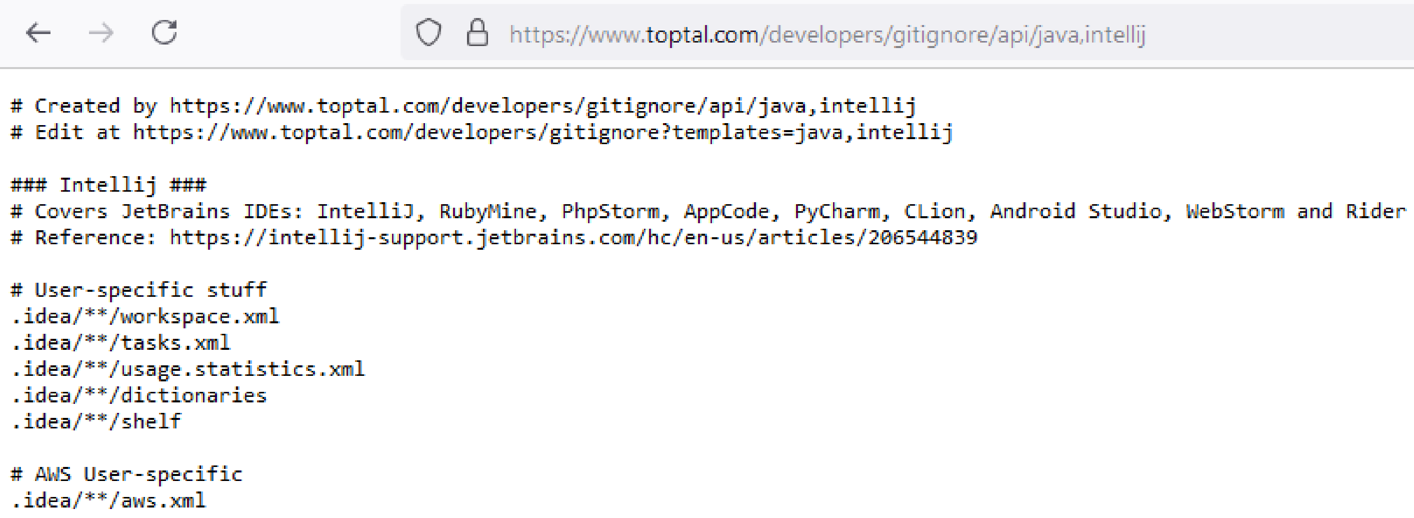
gitignore.io

Créez des fichiers .gitignore utiles à votre projet



Code Source | Documentation

Le clic sur « **Créer** » donne :




```
# Created by https://www.toptal.com/developers/gitignore/api/java,intellij
# Edit at https://www.toptal.com/developers/gitignore?templates=java,intellij

### Intellij ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio, WebStorm and Rider
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

# User-specific stuff
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/**/usage.statistics.xml
.idea/**/dictionaries
.idea/**/shelf

# AWS User-specific
.idea/**/aws.xml
```

5. *Voir les differences entre repertoire*

Nous avons vu que la commande git status nous donnait certaines informations sur les fichiers indexés.
Nous allons voir qu'il existe une autre commande permettant d'obtenir plus d'informations sur les fichiers indexés en zone de transit (staging area) : git diff.
La commande git diff sans aucune option permet d'afficher les modifications des fichiers dans le répertoire de travail qui ont été modifiés mais non indexés.
En fait, elle permet de comparer les fichiers du répertoire de travail avec la zone de transit, c'est-à-dire avec les fichiers indexés, et de n'afficher que les fichiers qui ont donc des modifications non indexées.

Lancer la commande **git diff**

git diff

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git diff

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

Modifier le title du fichier **index.html** qui devient :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Modified Page Title</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
  </head>
  <body>

  </body>
</html>
```

Lancer à nouveau la commande **git diff**

git diff

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git diff
diff --git a/index.html b/index.html
index e714a02..6cea71a 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@
     <head>
         <meta charset='utf-8'>
         <meta http-equiv='X-UA-Compatible' content='IE=edge'>
-        <title>Modified Page Title</title>
+        <title>Page Title</title>
         <meta name='viewport' content='width=device-width, initial-scale=1'>
     </head>
     <body>
```

Lancer la commande

git add index.html

git diff

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git diff

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

git diff --staged

```
$ git diff --staged
diff --git a/index.html b/index.html
index e714a02..6cea71a 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@
     <head>
         <meta charset='utf-8'>
         <meta http-equiv='X-UA-Compatible' content='IE=edge'>
-        <title>Modified Page Title</title>
+        <title>Page Title</title>
         <meta name='viewport' content='width=device-width, initial-scale=1'>
     </head>
     <body>
```

6. Gestion d'un commit Git

En Git, nous parlons de validation des modifications pour une sauvegarde.
Une sauvegarde s'appelle un commit, cela signifie littéralement que nous engageons les modifications indexées.
Un commit n'est donc pas à prendre à la légère ! C'est une sauvegarde qui sera à tout jamais dans le dépôt Git.

La commande git commit

Pour faire un commit, il suffit de taper :

git commit

commit

Nom (hash)

Auteur

Date

Message

Liste de hash de tous les fichiers

Un commit est à l'echelle Git est juste un fichier dans lequel Git va écrire des informations. Le nom de ce commit va être representé par un Hash.

Vous trouverez dans l'image ci-dessous les informations du commit comme le message du commit.

commit

Fichier

3c45e95e98a3205b150b974b5eadece5fb413324

```
tree 152d25cabcf6f30eb48b24701133494b501e625f
parent cd4d859ea42c5802041c3fa5a045cb3e9f58fb33
author jean <jean@gmail.com> 1583662192 +0100
committer jean <jean@gmail.com> 1583662192 +0100

second commit
```

Un **Tree** est aussi un fichier Git dont le nom est représenté par un Hash avec une liste des fichiers du commit.

Tree

Fichier

3c45e95e98a3205b150b974b5eadece5fb413324

```
100644 blob 7b202d2832fcb346742764fbc746029704660cd7    .gitignore
100644 blob d91cc4da4f8764b2658991a4798f1941f6c2a40c    index.html
```

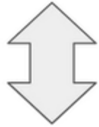
Que cela soit le commit, le Tree ou les fichiers du projet, ils sont tous stocké dans le fichier Objects.

Commit

Tree

Fichier du projet

Objects

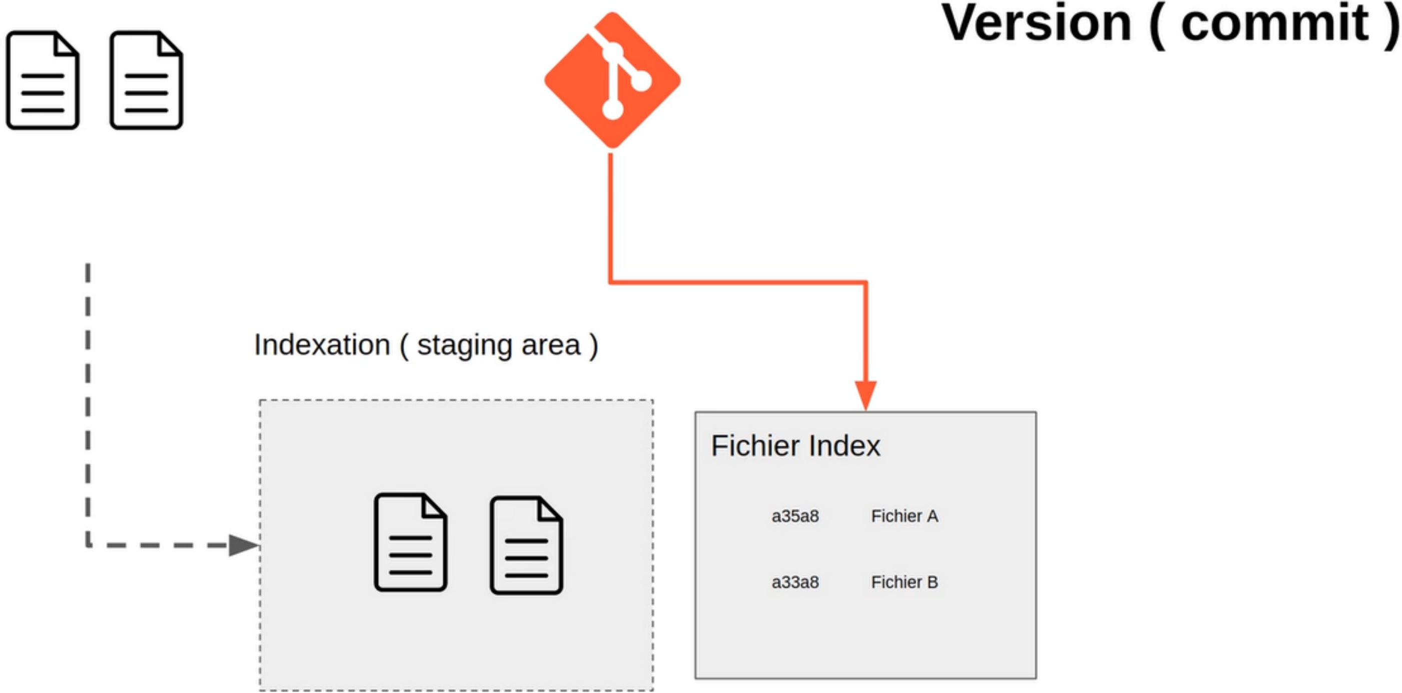


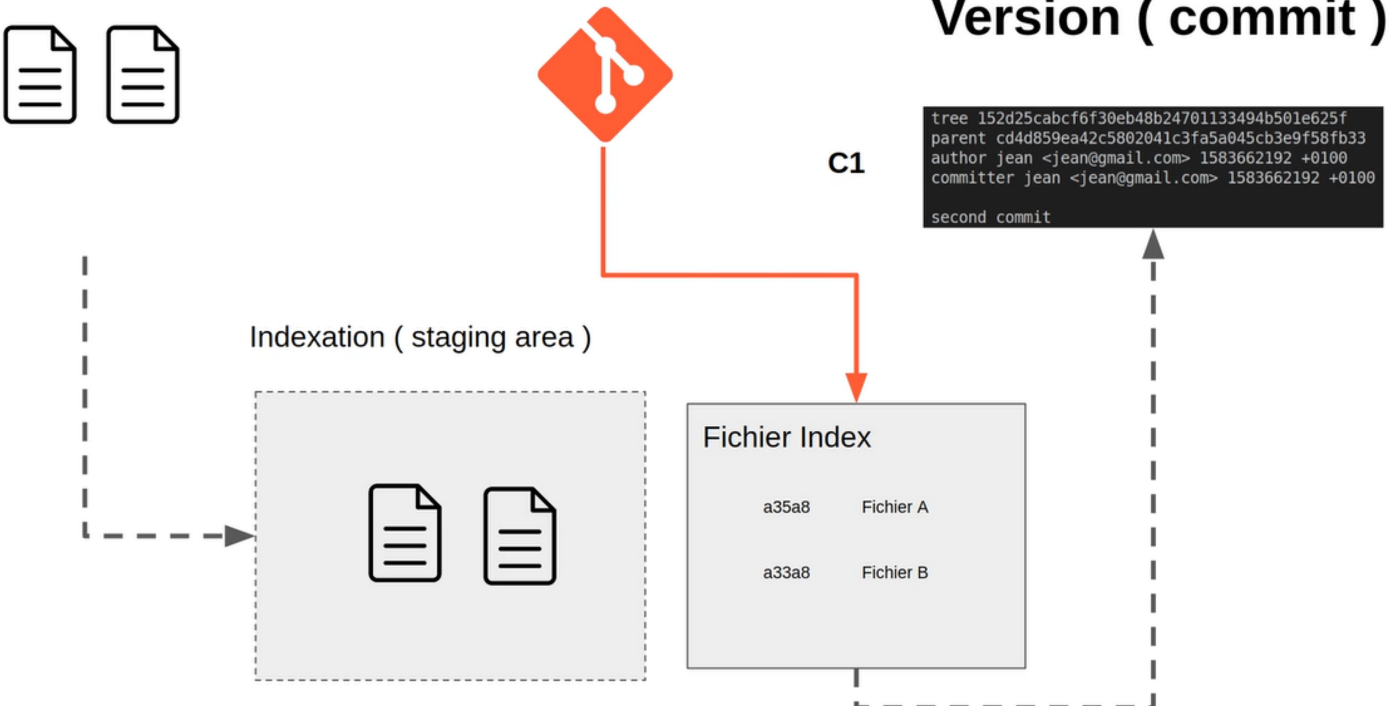
.git

Nous allons expliqué en detail le processus de commit :
Supposons qu'on se retrouve dans la situation où on doit commité deux fichiers.

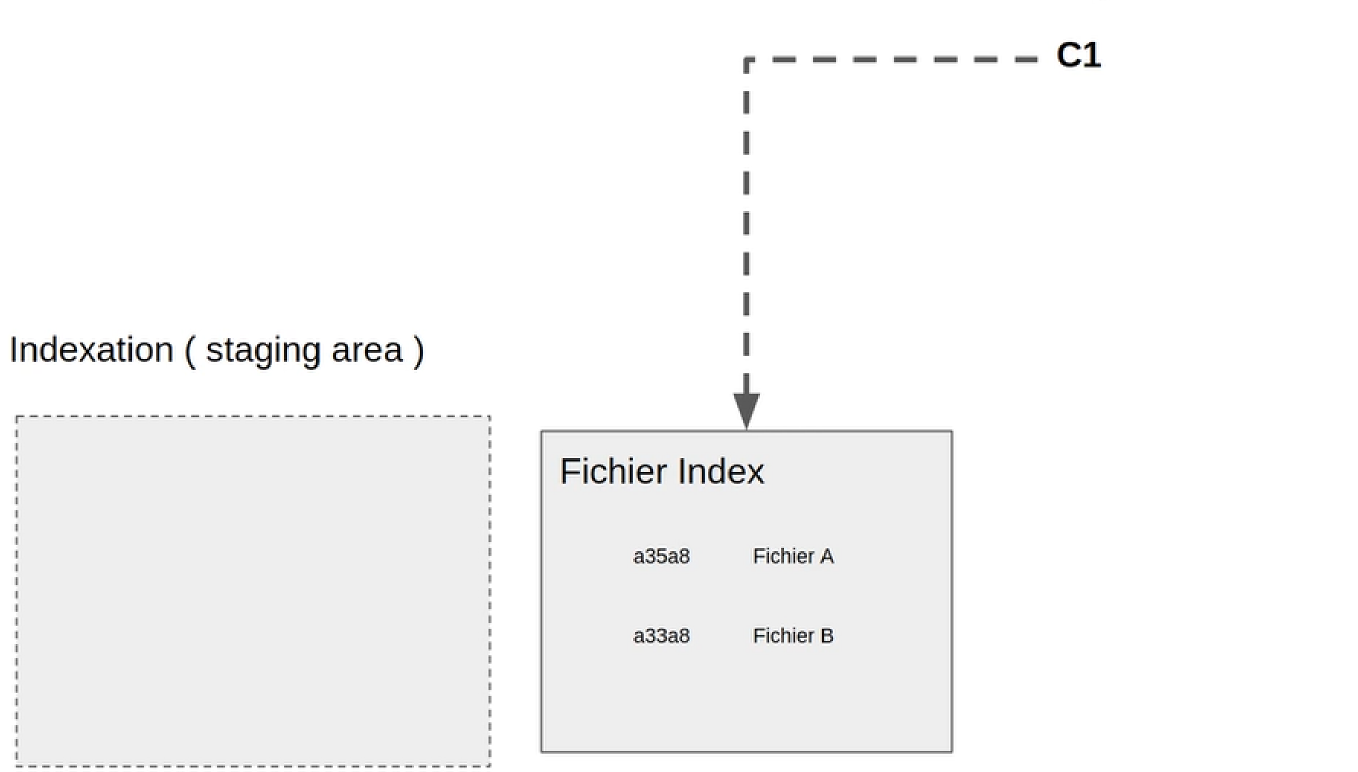
Version (commit)

Après la commande git add on ajoute les fichiers A et B dans le « staging area ». Dans l'index de Git les deux Hash correspondant aux deux fichiers A et B.

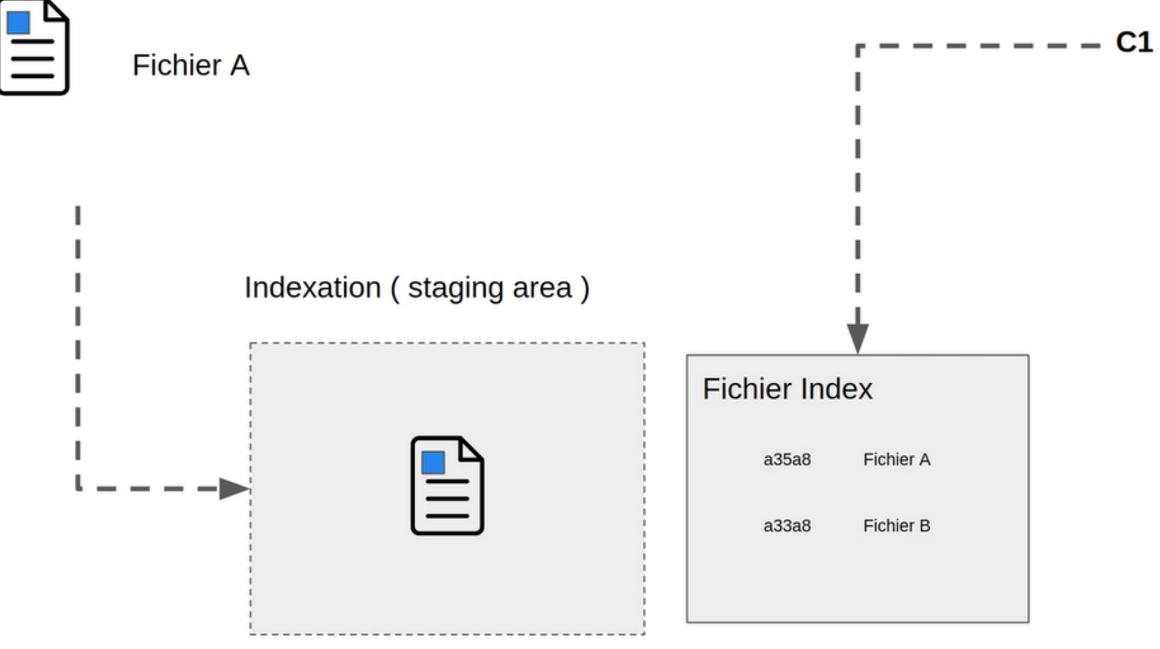




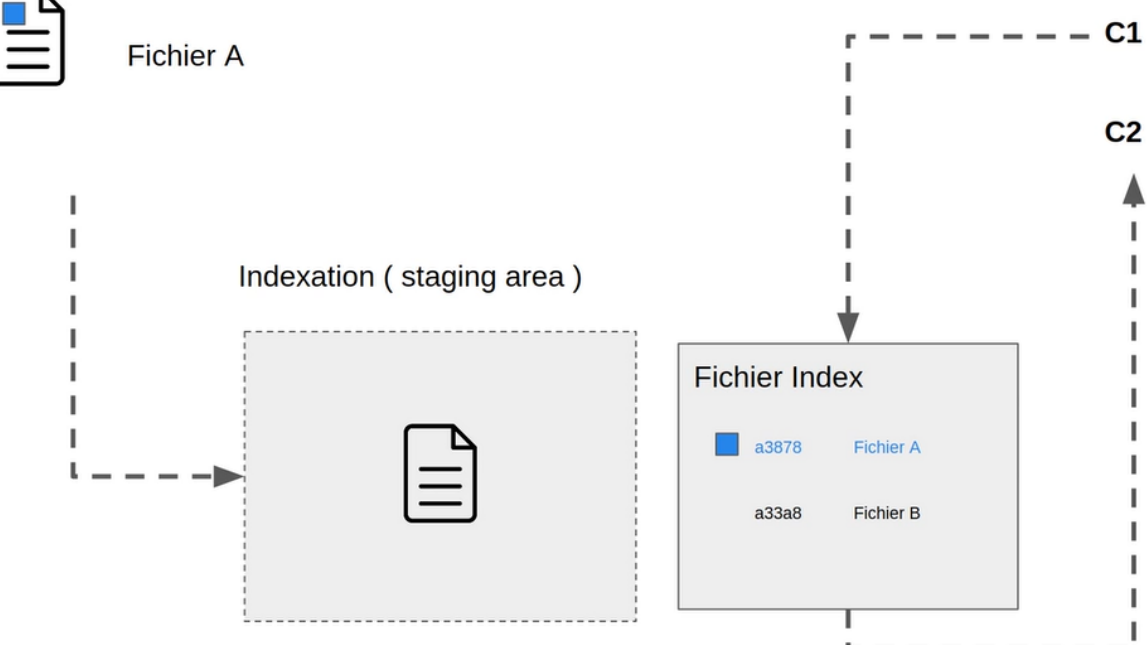
Après le commit **C1** Git va recréer l'index.

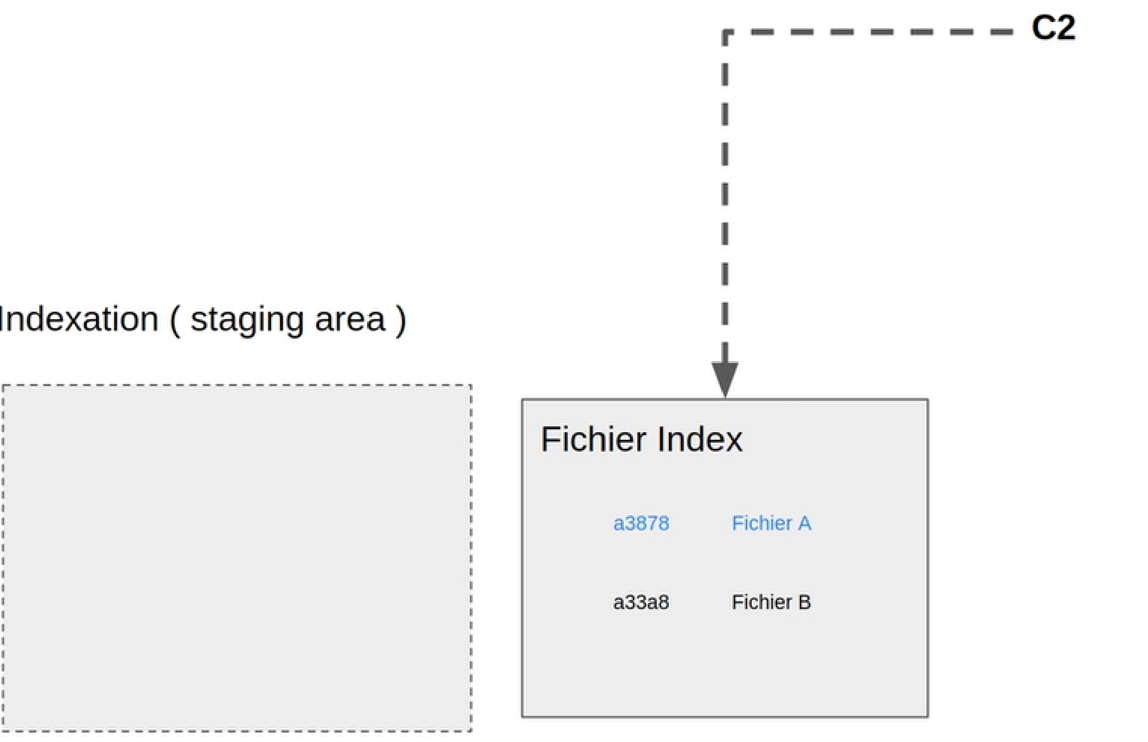


Supposons qu'on ajoute de nouvelles modification au fichier A.



Git génére un nouveau Hash.





Pratiquons maintenant :

Faire une autre modification dans le fichier **index.html** :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Seconde modification Page Title</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
  </head>
  <body>

  </body>
</html>
```

Lancer les commandes ci-dessous

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (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

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

git add index.html
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
```

git commit

```
1
2   # Please enter the commit message for your changes. Lines starting
3   # with '#' will be ignored, and an empty message aborts the commit.
4   #
5   # On branch master
6   # Changes to be committed:
7   #   modified:   index.html
8   #
```

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit
Aborting commit due to empty commit message.

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

En fermant l'éditeur cela annule tout mais on va recommencer pour mettre un bon message de commit.

git commit

Mettre sur la première ligne du fichier : Ceci est mon commit via Notepad

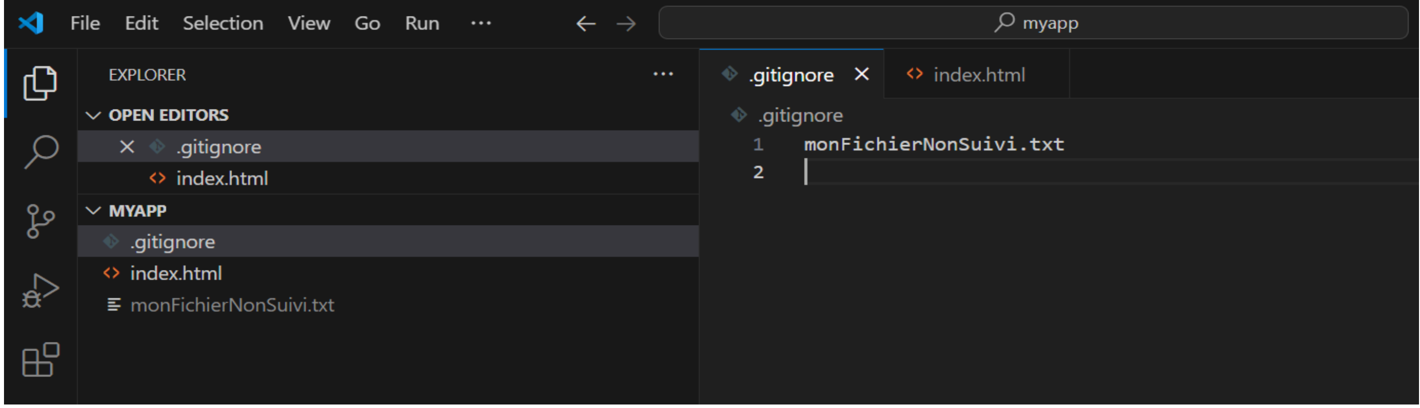
```
1
2  # Please enter the commit message for your changes. Lines starting
3  # with '#' will be ignored, and an empty message aborts the commit.
4  #
5  # On branch master
6  # Changes to be committed:
7  #   modified:   index.html
8  #
```

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit
[master 7927c82] Ceci est mon comimit via Notepad
 1 file changed, 1 insertion(+), 1 deletion(-)
```

7. *Suppression de fichiers avec Git*

La commande **git rm** peut être utilisée pour supprimer des fichiers individuels ou une série de fichiers. **git rm** a pour fonction principale de supprimer les fichiers suivis de l'index Git. En outre, la commande **git rm** permet de supprimer des fichiers de l'index de staging et du répertoire de travail.

Pour tester en pratique supposons qu'on veuille supprimer le fichier **.gitignore**.



Faire une sauvegarde du fichier **.gitignore** puis lancer les commandes :

Suppression Numéro 1

rm .gitignore
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ rm .gitignore
```

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    .gitignore

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

git add .gitignore
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add .gitignore

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    .gitignore
```

Suppression Numéro 2

Remettre le fichier **.gitignore** dans le repos puis lancer les commandes :

git status
git rm .gitignore
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git rm .gitignore
rm '.gitignore'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    .gitignore
```

On peut eventuellement utilisé la commande git restore --staged .gitignore pour enlever .gitignore dans le staging.

Effacer des fichiers uniquement de l'index

Vous pouvez utiliser l'option **--cached** avec git rm :

git rm --cached .gitignore

Dans ce cas le fichier (ou le dossier) sera supprimé de l'index (unstaged), mais restera dans le répertoire de travail.

8. *Historique avec git log*

Lorsque vous aurez plein de commits, ou lorsque vous arrivez sur un projet, il peut être utile de visualiser rapidement l'historique des commits.

Pour ce faire il suffit d'utiliser la commande :

git log

Passons maintenant à la pratique :

git log

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ceci est mon comimit via Notepad

commit 40606b1e8e887c4944146396b108e41e18dcf83f
Author: ElHadji <elhadji.gaye83@gmail.com>

    commit gitignore

commit c8d70e143acbe329abe27089fc8ab75a048ee5b0
Author: ElHadji <elhadji.gaye83@gmail.com>

    second commit

commit 11045600de263d4442592607f055905790f1ac18
Author: ElHadji <elhadji.gaye83@gmail.com>

    first commit
```

git log --oneline

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log --oneline
7927c82 (HEAD -> master) Ceci est mon comimit via Notepad
40606b1 commit gitignore
c8d70e1 second commit
1104560 first commit
```

On peut afficher les 2 dernier commit avec

git log -2

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log -2
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ceci est mon comimit via Notepad

commit 40606b1e8e887c4944146396b108e41e18dcf83f
Author: ElHadji <elhadji.gaye83@gmail.com>

    commit gitignore
```

Pour avoir plus de détails dans les commit on peut ajouter l'option –stat :
git log --stat

```
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ceci est mon comimit via Notepad

 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 40606b1e8e887c4944146396b108e41e18dcf83f
Author: ElHadji <elhadji.gaye83@gmail.com>

    commit gitignore

 .gitignore | 1 +
 1 file changed, 1 insertion(+)

commit c8d70e143acbe329abe27089fc8ab75a048ee5b0
Author: ElHadji <elhadji.gaye83@gmail.com>

    second commit

 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On peut aussi avoir les commit d'un autheur donnée :

git log --author="ElHadji"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log --author="ElHadji"
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ceci est mon comimit via Notepad

commit 40606b1e8e887c4944146396b108e41e18dcf83f
Author: ElHadji <elhadji.gaye83@gmail.com>

    commit gitignore

commit c8d70e143acbe329abe27089fc8ab75a048ee5b0
Author: ElHadji <elhadji.gaye83@gmail.com>

    second commit

commit 11045600de263d4442592607f055905790f1ac18
Author: ElHadji <elhadji.gaye83@gmail.com>

    first commit
```

git shortlog

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git shortlog
ElHadji (4):
      first commit
      second commit
      commit gitignore
      Ceci est mon comimit via Notepad
```

git log --grep="first"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log --grep="first"
commit 11045600de263d4442592607f055905790f1ac18
Author: ElHadji <elhadji.gaye83@gmail.com>

    first commit
```

9. *Historique avec git blame*

Git possède une commande qui permet de vous indiquer pour chaque ligne d'un fichier qui l'a modifié et quand : git blame.

On peut faire un git blame sur un fichier en particulier.

git blame myFile

Nous pouvons utiliser l'option -L pour spécifier un intervalle de lignes par exemple de 1 à 4 :

git blame -L 1,4 myFile

On peut aussi visualiser les mouvements de code

Git est tellement puissant qu'il peut repérer si une ligne de code a été déplacé depuis un autre fichier.

Mettons que vous refactorez votre code et que vous déplaciez trois lignes d'un fichier vers un autre fichier, grâce à git blame vous allez pouvoir le voir !

L'option -C permet de détecter les lignes déplacées ou copiées depuis d'autres fichiers dans le commit où elles ont été ajoutées au fichier.

Par défaut l'option va détecter les suites de 40 caractères qui ont été déplacés / copiés.

git blame -C -L 3,6 myFile

Passons maintenant à la pratique :

git blame index.html

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git blame index.html
^1104560 (ElHadji                                          1) <!DOCTYPE html>
^1104560 (ElHadji                                          2) <html>
^1104560 (ElHadji                                          3)     <head>
^1104560 (ElHadji                                          4)         <meta charset='utf-8'>
^1104560 (ElHadji                                          5)         <meta http-equiv='X-UA-Compatible' content='IE=edge
>
7927c825 (ElHadji                                          6)         <title>Seconde modification Page Title</title>
^1104560 (ElHadji                                          7)         <meta name='viewport' content='width=device-width,
nitial-scale=1'>
^1104560 (ElHadji                                          8)     </head>
^1104560 (ElHadji                                          9)     <body>
^1104560 (ElHadji                                         10)
^1104560 (ElHadji                                         11)     </body>
^1104560 (ElHadji                                         12) </html>
```

git blame index.html -b

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git blame index.html -b
         (ElHadji                            1) <!DOCTYPE html>
         (ElHadji                            2) <html>
         (ElHadji                            3)     <head>
         (ElHadji                            4)         <meta charset='utf-8'>
         (ElHadji                            5)         <meta http-equiv='X-UA-Compatible' content='IE=edge
'>
7927c825 (ElHadji                            6)         <title>Seconde modification Page Title</title>
         (ElHadji                            7)         <meta name='viewport' content='width=device-width, i
nitial-scale=1'>
         (ElHadji                            8)     </head>
         (ElHadji                            9)     <body>
         (ElHadji                           10)
         (ElHadji                           11)     </body>
         (ElHadji                           12) </html>
```

git blame index.html -f

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git blame index.html -f
^1104560 index.html (ElHadji                          1) <!DOCTYPE html>
^1104560 index.html (ElHadji                          2) <html>
^1104560 index.html (ElHadji                          3)     <head>
^1104560 index.html (ElHadji                          4)         <meta charset='utf-8'>
^1104560 index.html (ElHadji                          5)         <meta http-equiv='X-UA-Compatible' conten
t='IE=edge'>
7927c825 index.html (ElHadji                          6)         <title>Seconde modification Page Title</t
itle>
^1104560 index.html (ElHadji                          7)         <meta name='viewport' content='width=devi
ce-width, initial-scale=1'>
^1104560 index.html (ElHadji                          8)     </head>
^1104560 index.html (ElHadji                          9)     <body>
^1104560 index.html (ElHadji                         10)
^1104560 index.html (ElHadji                         11)     </body>
^1104560 index.html (ElHadji                         12) </html>
```

git blame index.html -e

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git blame index.html -e
^1104560 (<elhadji.gaye83@gmail.com>                          1) <!DOCTYPE html>
^1104560 (<elhadji.gaye83@gmail.com>                          2) <html>
^1104560 (<elhadji.gaye83@gmail.com>                          3)     <head>
^1104560 (<elhadji.gaye83@gmail.com>                          4)         <meta charset='utf-8'>
^1104560 (<elhadji.gaye83@gmail.com>                          5)         <meta http-equiv='X-UA-Compatible
' content='IE=edge'>
7927c825 (<elhadji.gaye83@gmail.com>                          6)         <title>Seconde modification Page
Title</title>
^1104560 (<elhadji.gaye83@gmail.com>                          7)         <meta name='viewport' content='wi
dth=device-width, initial-scale=1'>
^1104560 (<elhadji.gaye83@gmail.com>                          8)     </head>
^1104560 (<elhadji.gaye83@gmail.com>                          9)     <body>
^1104560 (<elhadji.gaye83@gmail.com>                         10)
^1104560 (<elhadji.gaye83@gmail.com>                         11)     </body>
^1104560 (<elhadji.gaye83@gmail.com>                         12) </html>
```

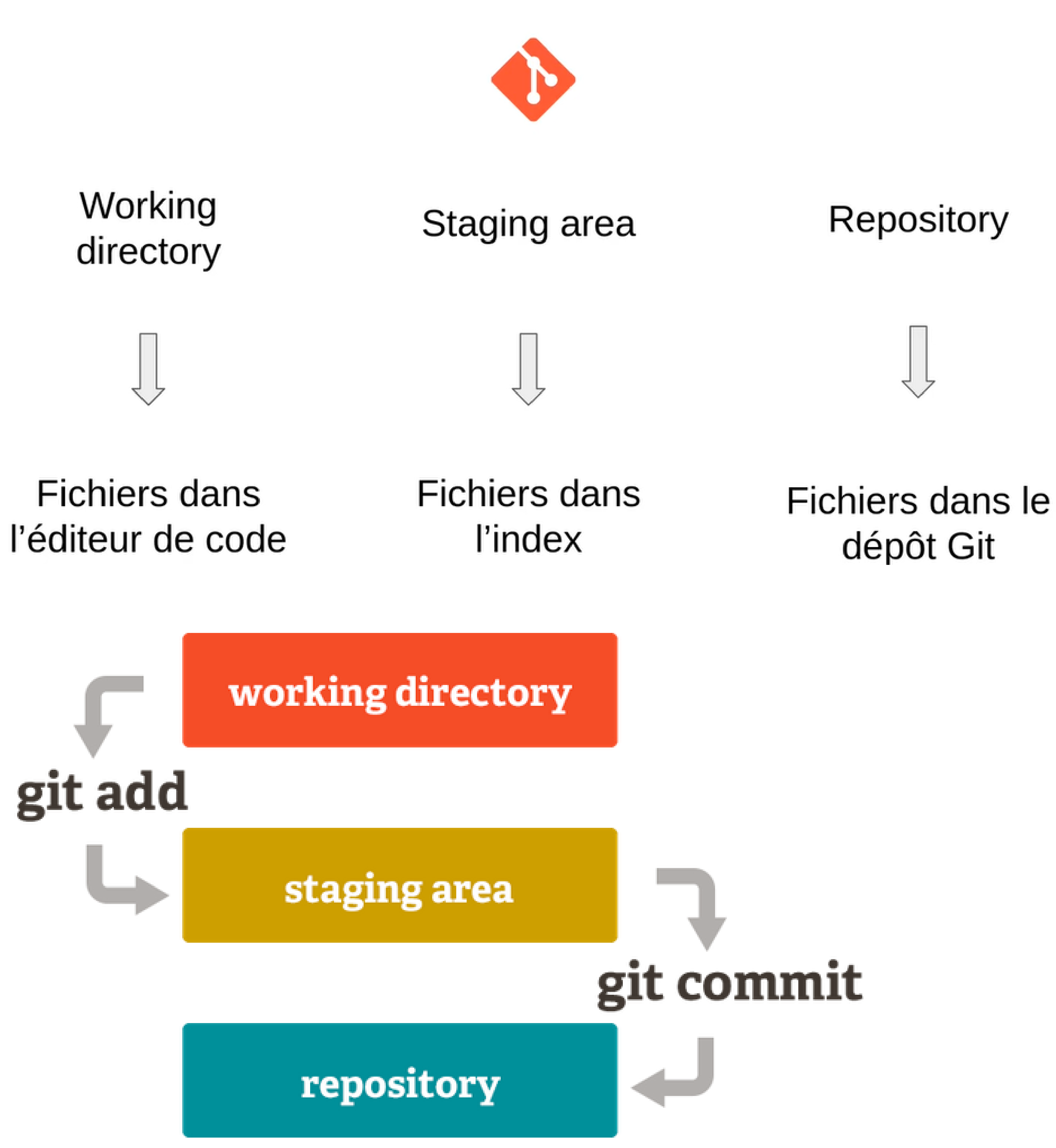
git blame index.html -L 1,3

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git blame index.html -L 1,3
^1104560 (ElHadji                          1) <!DOCTYPE html>
^1104560 (ElHadji                          2) <html>
^1104560 (ElHadji                          3)     <head>
```

git blame index.html > log.txt

10. Branche master et HEAD

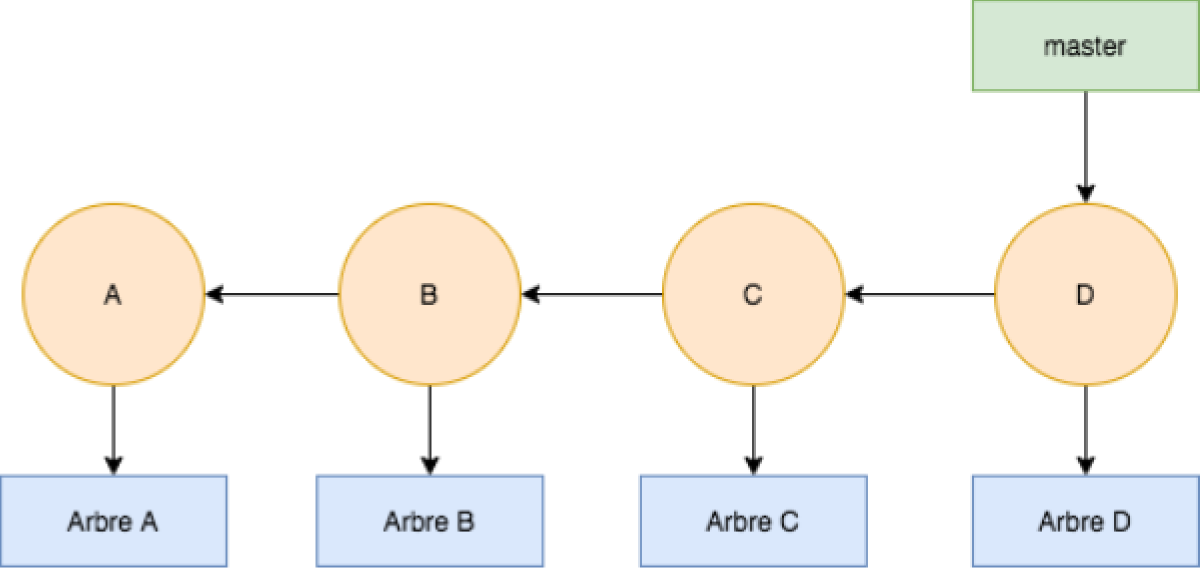
Nous avons vu que la version des fichiers tels qu'ils apparaissent dans l'éditeur de code est appelée repertoire de travail (ou working directory).
Nous avons vu que la version des fichiers qui a été ajoutée à l'index avec git add sont dits indexés ou en zone de transit (staging area).
Nous avons enfin vu que la version des fichiers qui était sauvegardée avec git commit depuis la zone de transit est ajoutée au répertoire Git (repository), appelé également dépôt.



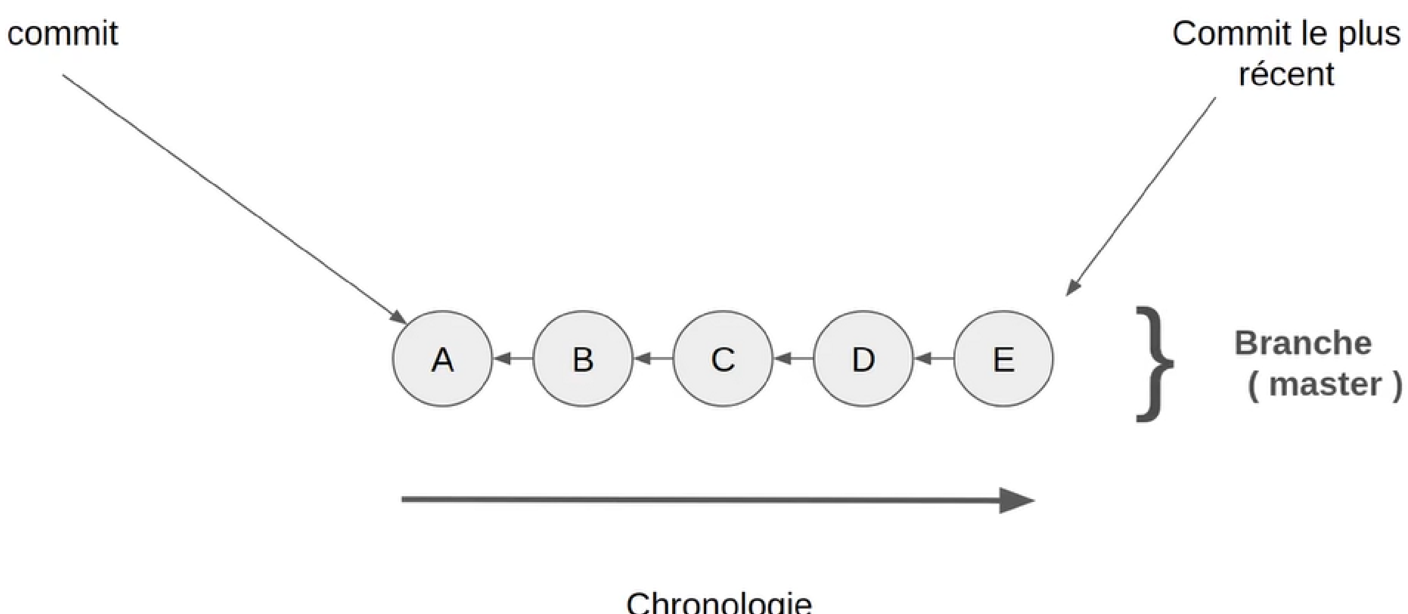
Qu'est-ce que la branche main ?

Les commits forment donc une chaîne de commits grâce à ces références.
Une branche est simplement un pointeur, également appelé référence, vers un commit.
main est la branche principale créée par défaut par Git lors du git init.
Au fur et à mesure que vous créez des commits sur une branche, la référence de la branche se déplace automatiquement au dernier commit.

Exemple :

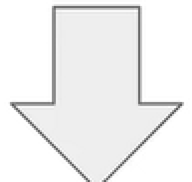


Au début du projet nous avons que le commit A, la branche main pointe sur celui-ci.
Ensuite, nous faisons un second commit B qui a pour parent le commit A, la branche main pointe alors automatiquement vers le commit B.
Ainsi de suite jusqu'au commit D qui est dans l'exemple le dernier commit réalisé.
Nous pouvons vérifier qu'une branche est simplement une référence en faisant :



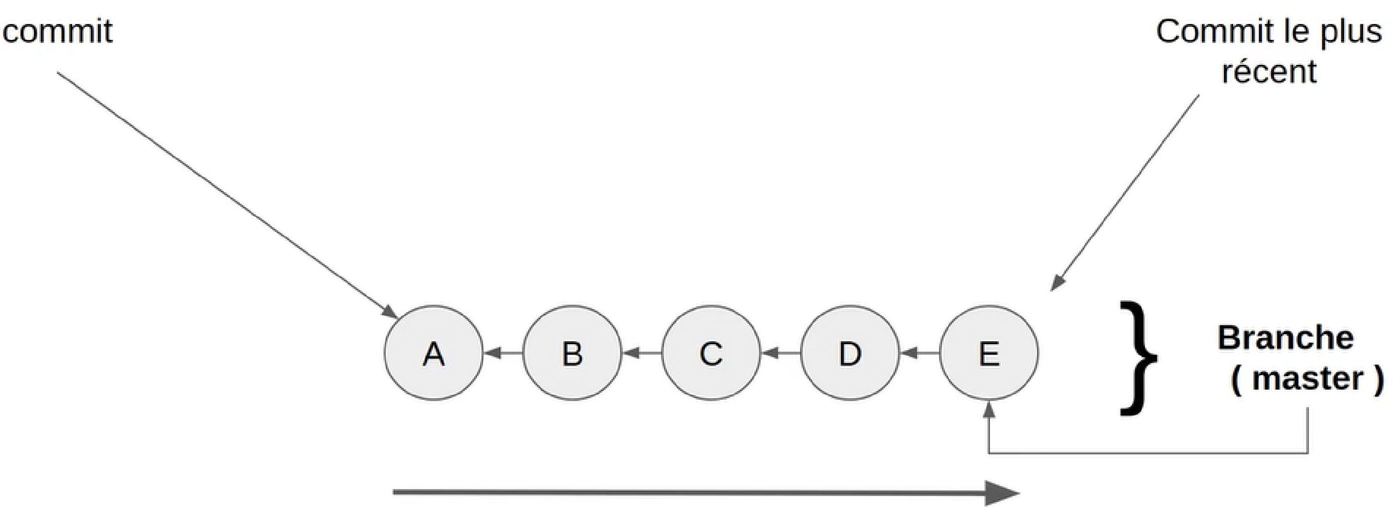
EL Hadji Gaye

MASTER



MAIN

Branche = référence d'un commit (E)

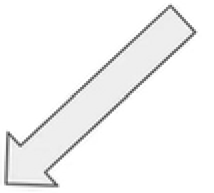


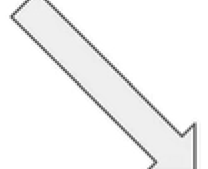
Chronologie

HEAD est une référence, ou pointeur, vers le commit, la branche, ou le tag sur laquelle vous vous trouvez actuellement. Par défaut, HEAD pointe vers la branche main qui pointe vers le dernier commit.

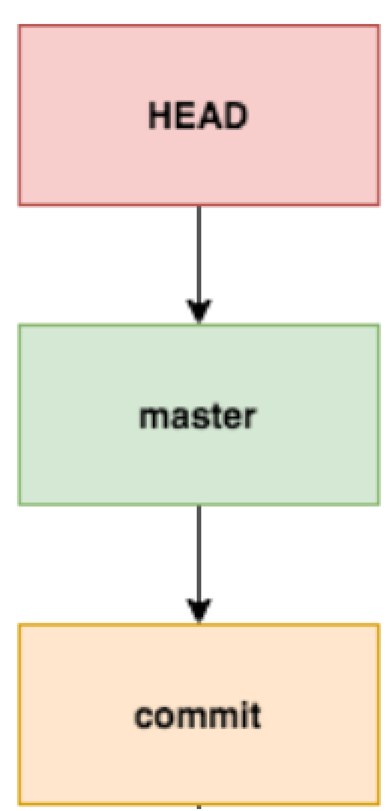
Le head va pointer soit sur une branche soit sur un commit :

HEAD

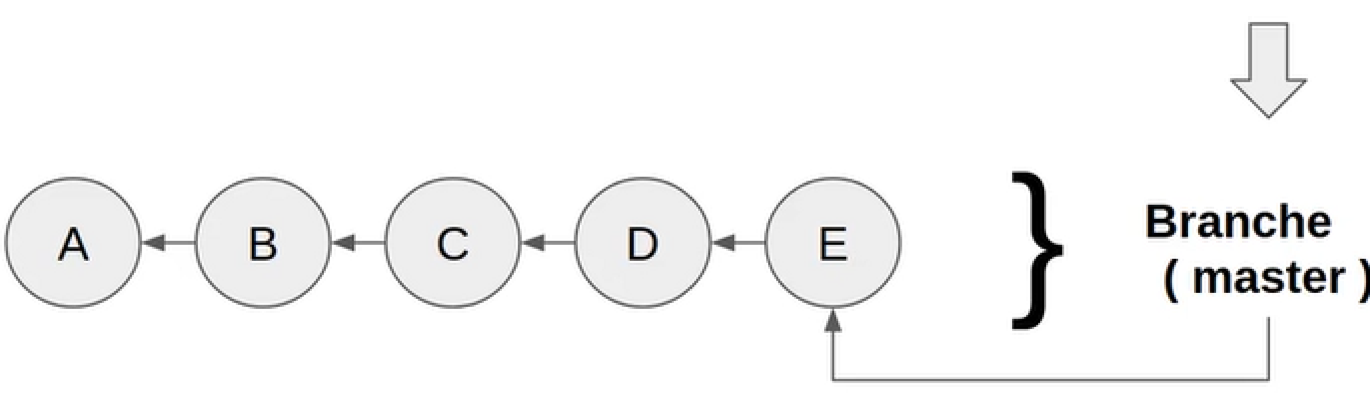




branche **commit**



HEAD



Le head est tout simplement le commit qui est active dans la branche.

Nous allons maintenant voir tout cela en pratique.

Lancer les commandes :

cd .git
ls
cat HEAD

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ cd .git

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ ls
COMMIT_EDITMSG  config  description  HEAD  hooks/  index  info/  logs/  objects/  refs/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ cat HEAD
ref: refs/heads/master
```

Ce qui veut dire effectivement que HEAD pointe vers master.
Allons maintenant voir ce qui se trouve dans notre master.

cd refs
ls
cd heads
ls
cat master

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ cd refs

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs (GIT_DIR!)
$ ls
heads/  tags/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs (GIT_DIR!)
$ cd heads

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs/heads (GIT_D
IR!)
$ ls
master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs/heads (GIT_D
IR!)
$ cat master
7927c8258ebd6d9954f1596e81a84faf3d253fa2
```

Lancer la commande **git log -1** pour verifier le dernier commit

git log -1

```
$ git log -1
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ceci est mon comimit via Notepad
```

Effectivement **HEAD** pointe vers le dernier commit.

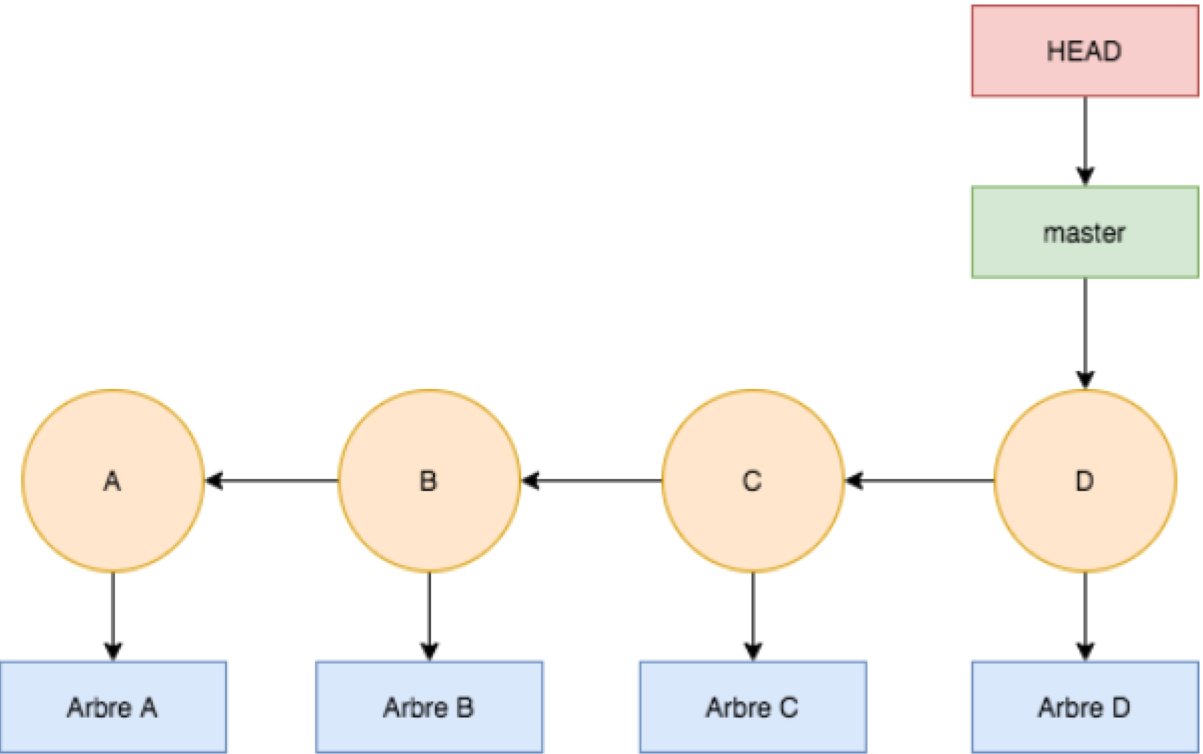
11. Commande git checkout

La commande git checkout est l'une des plus utilisées dans Git.
Elle permet de se déplacer vers une autre branche ou vers un tag (comme nous le verrons), vers un autre commit, ou de restaurer la version indexée.
Il faut vraiment voir la commande checkout comme une commande de navigation de HEAD.
Autrement dit elle permet de déplacer HEAD où vous souhaitez.

Comment naviguer entre les commits

La commande git checkout hashCommit permet de déplacer HEAD vers le commit spécifié et de mettre à jour le répertoire de travail et l'index à la version correspondant à ce commit.

Prenons la configuration suivante :

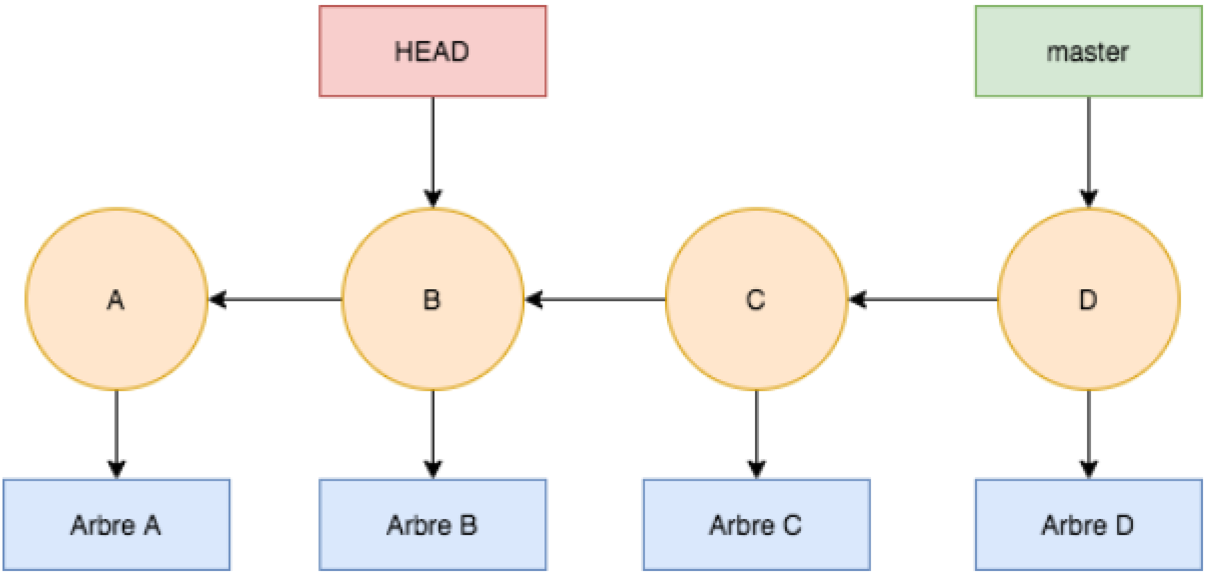


Par défaut, comme nous l'avons vu, HEAD pointe sur une branche, en l'occurrence la branche principale main créée par Git par défaut.

Mais si nous faisons :

git checkout hashCommitB

Que va t'il se passer ? Nous serons dans la situation suivante :



Nous avons déplacé HEAD vers le commit B.

Si nous faisons :

cd .git
cat HEAD

Nous avons maintenant directement le hash du commit B, par exemple :

9e633d56381d9f0335320f22ccf3f1562d1f80d8

Si nous faisons :

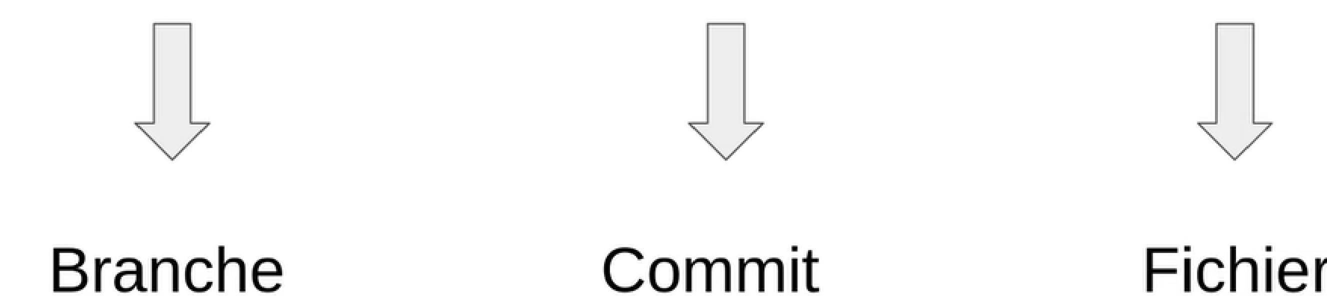
git status

Nous aurons en rouge :

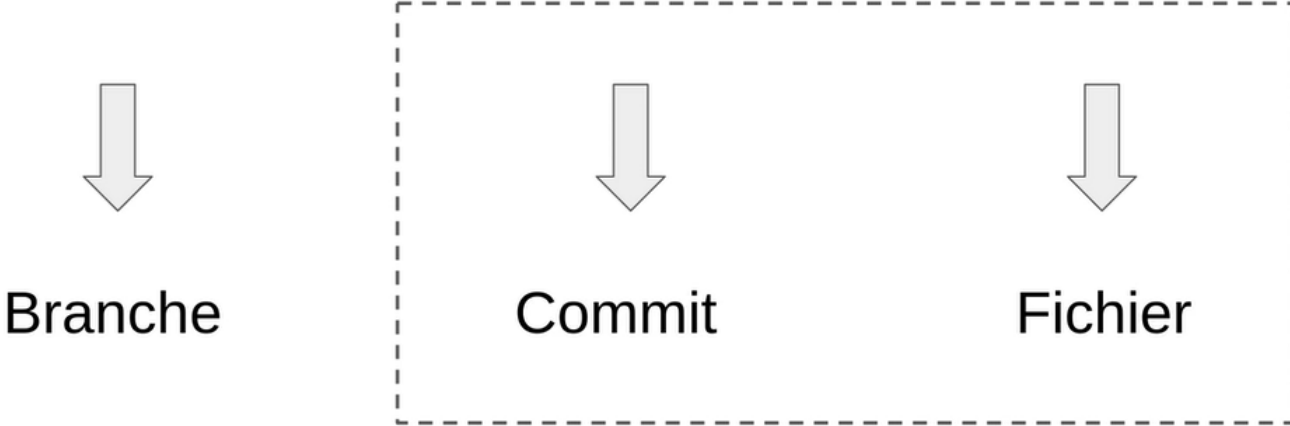
HEAD detached at 9e633d5
nothing to commit, working tree clean

git checkout par la pratique

On peut utilisé la commande git checkout dans 3 situations : **Branche**, **Commit** et **Fichier**.

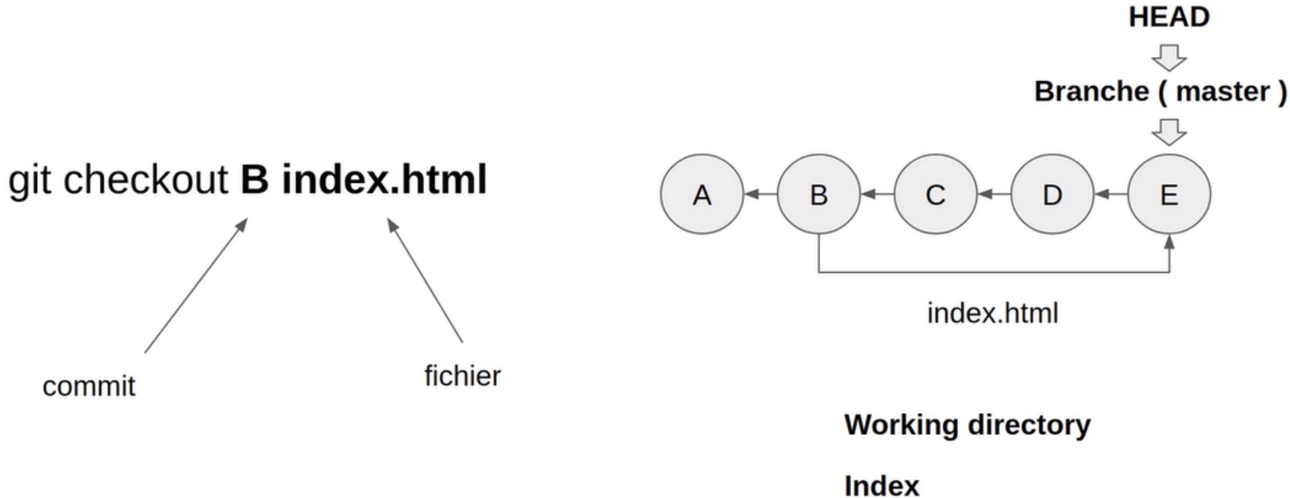


Dans cette partie de la formation nous allons nous limité aux cas : **Commit** et **Fichier**.



Le Git checkout sur un Fichier

Git checkout => fichier



En pratique nous avons :

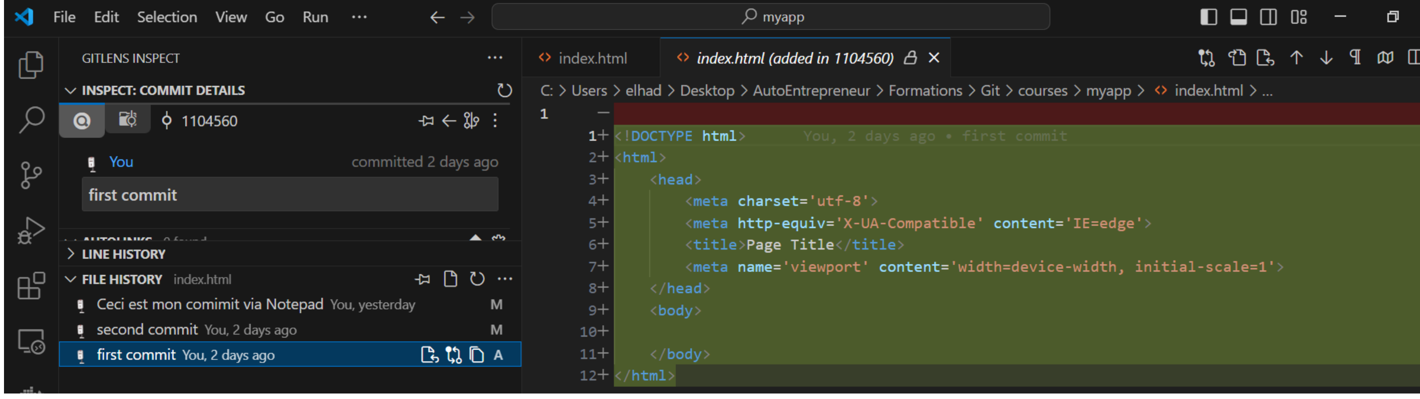
Supposons qu'on veuille revenir à la version initiale de notre fichier **index.html**.

Faisons d'abord un git status.

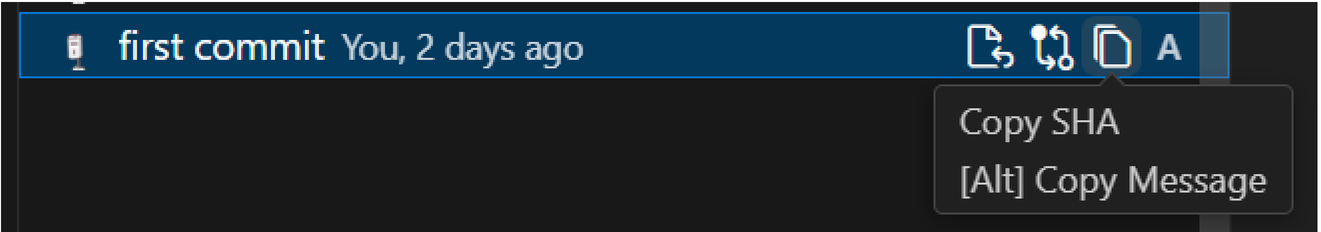
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Cliquer à gauche du **GitLens** :



Supposons que l’on soit interessé par « **first commit** » on peut retrouvé son hash facilement.



Copy SHA nous donne l’information **11045600de263d4442592607f055905790f1ac18**

Il faut noter qu’on aurait pu recuperer cet information par l’intermediaire d’un git log.

```
$ git log
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>


    Ceci est mon comimit via Notepad

commit 40606b1e8e887c4944146396b108e41e18dcf83f
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Mon Sep 23 05:12:27 2024 +0200

    commit gitignore

commit c8d70e143acbe329abe27089fc8ab75a048ee5b0
Author: ElHadji <elhadji.gaye83@gmail.com>


    second commit

commit 11045600de263d4442592607f055905790f1ac18
Author: ElHadji <elhadji.gaye83@gmail.com>


    first commit
```

Nous avons comme contenu du fichier **index.html**

```
index.html ×
index.html > html > head > meta
You, yesterday | 1 author (You)
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset='utf-8'>        You, 2 days ago • first commit
5           <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6           <title>Seconde modification Page Title</title>
7           <meta name='viewport' content='width=device-width, initial-scale=1'>
8       </head>
9       <body>
10
11      </body>
12  </html>
```

Chargons maintenant le contenu de index.html avec le commit **11045600de263d4442592607f055905790f1ac18**

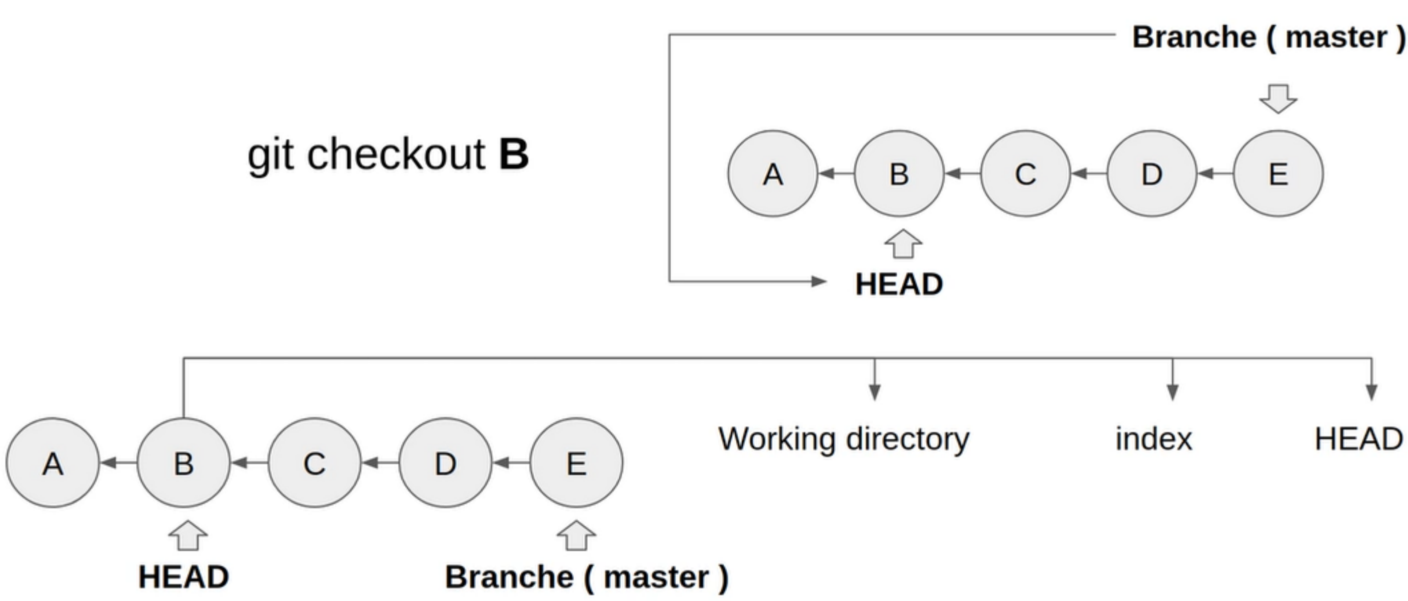
git checkout 11045600de263d4442592607f055905790f1ac18 index.html
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout 11045600de263d4442592607f055905790f1ac18 index.html
Updated 1 path from 4e5e489

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
```

On obtient le contenu suivant pour index.html :

```
index.html M ×
index.html > html
You, 56 seconds ago | 1 author (You)
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset='utf-8'>
5           <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6           <title>Page Title</title>
7           <meta name='viewport' content='width=device-width, initial-scale=1'>
8       </head>
9       <body>
10
11      </body>
12  </html>        You, 2 days ago • first commit
```

Le Git checkout sur un Commit



Si on decide de revenir au commit initiale sans préciser de fichier :

git checkout 11045600de263d4442592607f055905790f1ac18

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp ((7927c82...))
$ git checkout 11045600de263d4442592607f055905790f1ac18
M       index.html
Previous HEAD position was 7927c82 Ceci est mon comimit via Notepad
HEAD is now at 1104560 first commit
```

Comme on peut le voir le **HEAD** est maintenant sur 1104560
Ajoutons par exemple une balise **h2**.

```
index.html > html > body
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Seconde modification Page Title</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <h2>Bonjour</h2>
11     </body>        You, 2 days ago • first commit
12 </html>
```

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp ((1104560...))
$ git status
HEAD detached at 1104560
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

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

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp ((1104560...))
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp ((1104560...))
$ git commit -m "commit lost"
[detached HEAD 34b9e44] commit lost
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Retournons à notre branche **master** :

```
$ git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  34b9e44 commit lost

If you want to keep it by creating a new branch, this may be a good time
to do so with:

 git branch <new-branch-name> 34b9e44

Switched to branch 'master'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
```

Comme vous le voyez dans le message si vous ne créer pas de branche pour le Hash **34b9e44** alors les modifications seront perdus.

git checkout master pour retouner à un état stable.

12. Commande git clean

La commande **git clean** permet de supprimer tous les fichiers non suivis du répertoire de travail, sauf ceux contenus dans **.gitignore**.

Attention ! Les fichiers sont définitivement perdus après cette commande.

Les options de la commande :

L'option -n, pour dry-run, permet de savoir quels fichiers seront supprimés avant de procéder à leur suppression.

git clean -n

L'option -f, pour force, permet d'enclencher la suppression des fichiers. C'est une mesure de protection car les fichiers sont définitivement supprimés :

git clean -f

L'option -i, pour interactive permet de lancer le mode interactif de Git

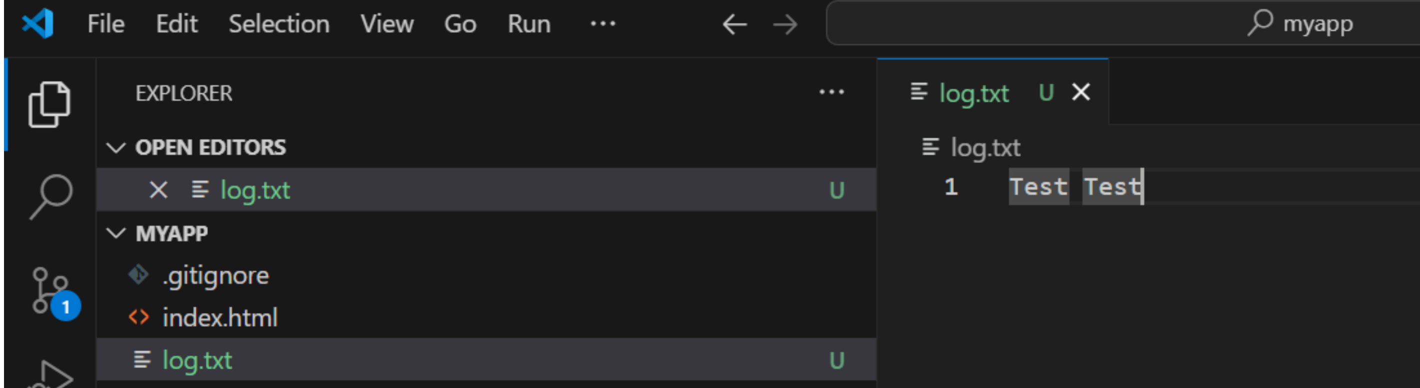
git clean -i

Passons maintenant à la pratique !

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Créer le fichier **log.txt** avec un contenu quelconque.



```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        log.txt

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

Lancer la commande suivante :

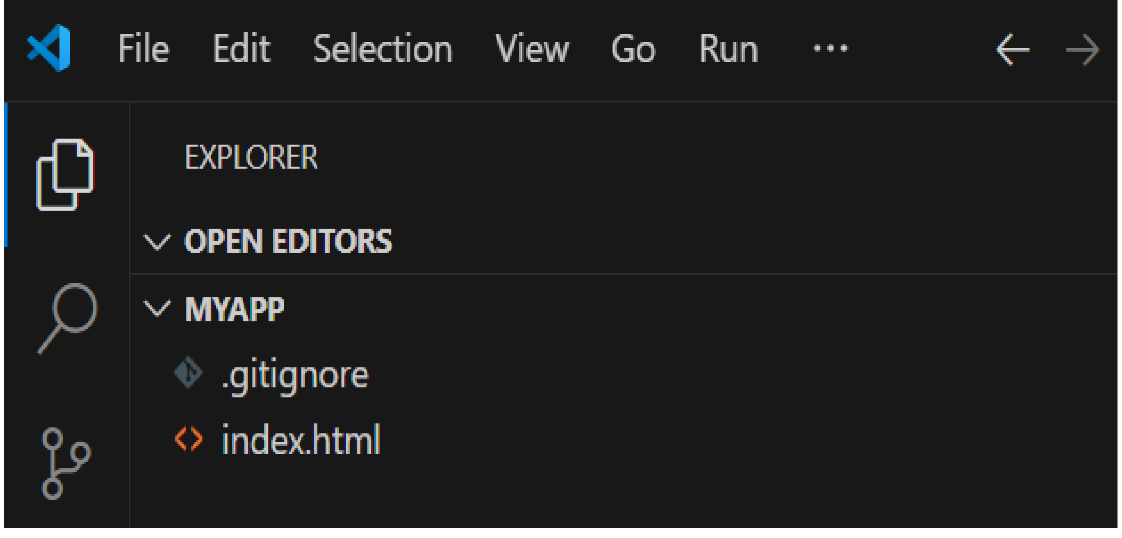
git clean

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git clean
fatal: clean.requireForce is true and -f not given: refusing to clean
```

git clean -i

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git clean -i
Would remove the following item:
  log.txt
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers
    4: ask each             5: quit                 6: help
What now> 1
Removing log.txt
```

Le fichier a été bien supprimé.



Faire un git status pour verifier qu'il n'y aucun fichier en cours.

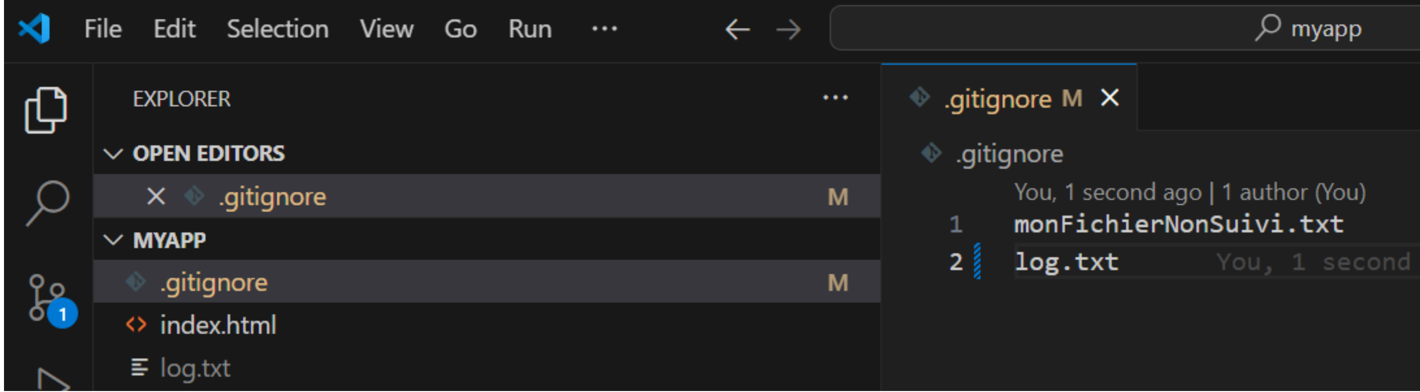
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

git clean -n
git clean -f

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git clean -n
Would remove log.txt

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git clean -f
Removing log.txt

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git clean -n
```

Que se passe t'il maintenant si on crée à nouveau le fichier log.txt qu'on l'ajoute au fichier .**gitignore**.



Lancer la commande **git clean -n** à nouveau :

git clean -n

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git clean -n
```

13. Commande git revert

Cette commande permet d’annuler les changements d'un commit.
La commande git revert permet de créer un nouveau commit qui va annuler tous les changements effectués par le commit spécifié.

Cette commande est très utile pour annuler un commit contenant des erreurs en conservant l'historique de toutes les modifications.

git revert commit

Où commit est le hash du commit dont vous voulez annuler les modifications.

Par défaut, l'éditeur par défaut sera ouvert pour spécifier le message de validation qui sera pré-rempli par Revert + le message de validation du commit que vous voulez annuler.

Si vous ne voulez pas avoir à modifier le message et que celui par défaut vous convient, vous pouvez faire :

git revert commit --no-edit

Dans ce cas le commit se fera immédiatement avec le message prérempli.

Comment annuler les changements d'un commit sans créer de nouveau commit

Parfois vous voudrez annuler les changements d'un commit et faire quelques modifications supplémentaires avant d'effectuer un nouveau commit.

Dans ce cas il faut utiliser git revert **--no-commit** :

git revert --no-commit commit

Où commit est le hash du commit dont vous voulez annuler les modifications.

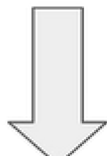
Cette commande va annuler tous les changements effectués par le commit cible, mais ne va pas créer de nouveau commit.

Elle va annuler les changements dans le répertoire de travail et dans l'index sans créer de commit.

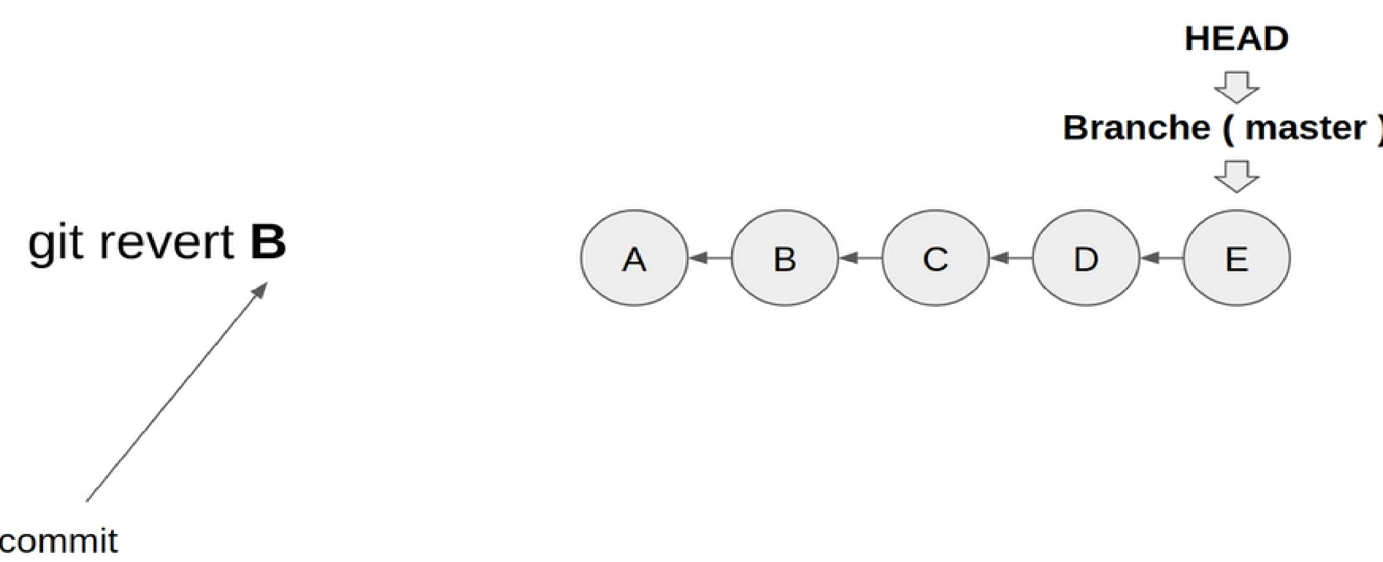
Pratiquons maintenant un peu !

Le Git revert est uniquement applicable à un commit.

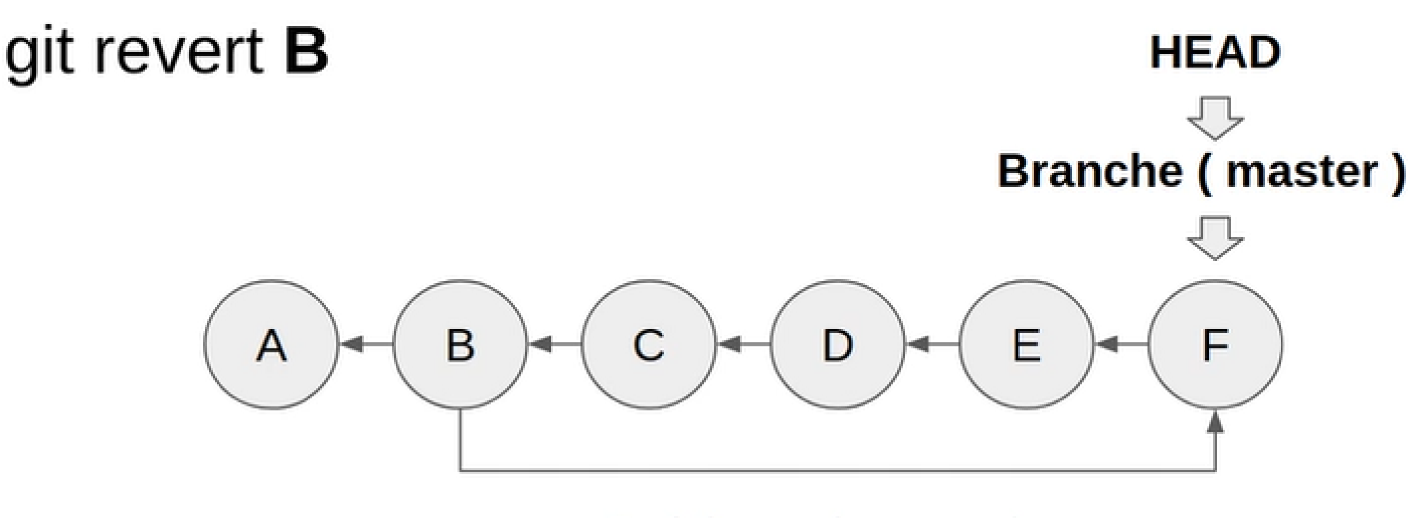
Git revert



Commit

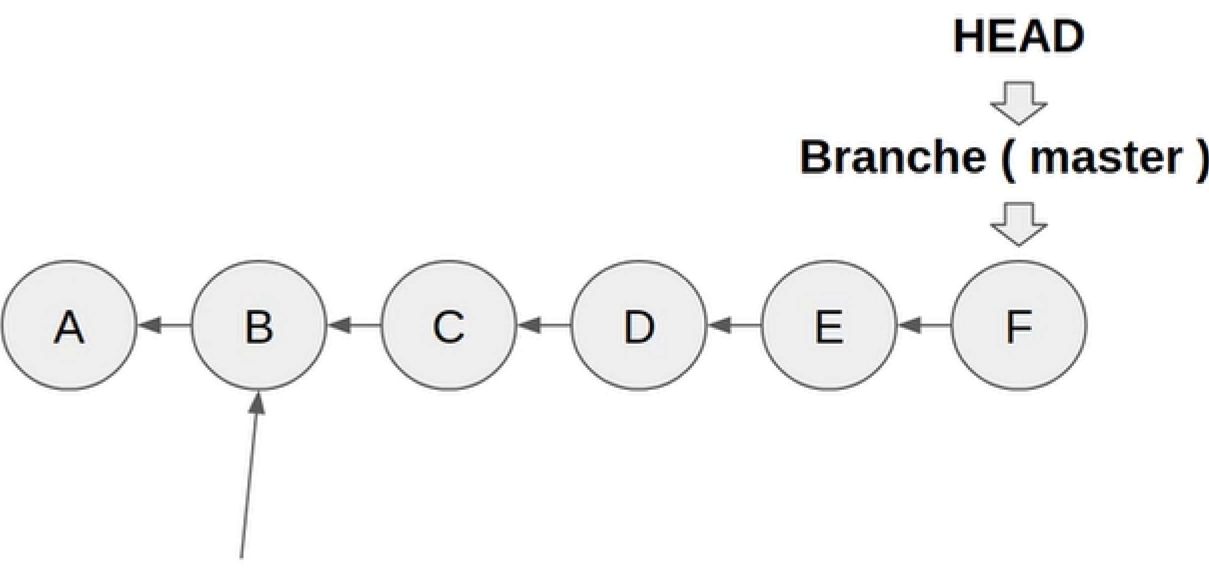


Le git revert B va créer le commit F.



Defait tout le commit **B**.

Il faut noter que le commit B existe toujours.



Le commit **F** repare le commit **B**.

Revenons à notre projet lançons un **git status** :

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Modifions le footer du fichier **index.html** :

```
index.html M ×

index.html > html
You, 29 seconds ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Seconde modification Page Title</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <footer>Ceci est le footer</footer>
11     </body>
12 </html>        You, 2 days ago • first commit
```

Lancer les commandes :

git add index.html

git commit -m "Ajout du footer"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "Ajout du footer"
[master d1ca0e1] Ajout du footer
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Faisons un revert sur le dernier commit :

git log -1

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log -1
commit d1ca0e115df634711a81337d2198eaa36e60f34f (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Tue Sep 24 15:08:41 2024 +0200

    Ajout du footer
```

git revert d1ca0e115df634711a81337d2198eaa36e60f34f

Après avoir fermer votre éditeur vous obtenez :

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git revert d1ca0e115df634711a81337d2198eaa36e60f34f
[master 247dacd] Revert "Ajout du footer"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

git log -3

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log -3
commit 247dacd93e9898e2b3f4a520e05667fbae5bd193 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>

    Revert "Ajout du footer"

    This reverts commit d1ca0e115df634711a81337d2198eaa36e60f34f.

commit d1ca0e115df634711a81337d2198eaa36e60f34f
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ajout du footer

commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2
Author: ElHadji <elhadji.gaye83@gmail.com>

    Ceci est mon comimit via Notepad
```

14. Commande git reset

La commande git reset est l'une des plus importantes de Git. Il est très important de bien la comprendre et bien la maîtriser.

Elle permet de faire beaucoup de choses, et peut être dangereuse dans certains cas car elle peut provoquer des pertes de données irréversibles.

- **Désindexer des fichiers ou des dossiers indexés**

La commande **git reset fichier** est l'inverse de **git add fichier** : elle permet de désindexer un fichier en conservant les changements dans le répertoire de travail.

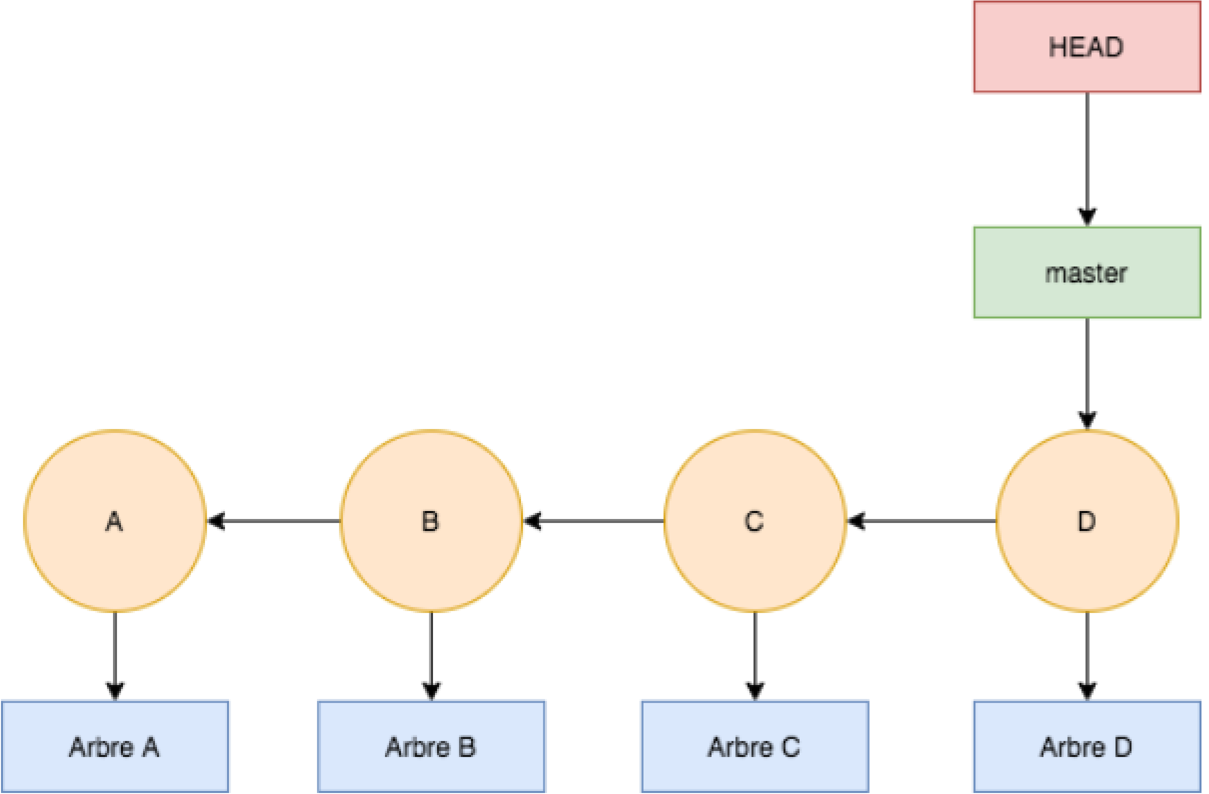
En fait, elle prend la version du fichier du dernier commit et la met dans l'index. Cela revient effectivement à désindexer un fichier.

Si vous spécifiez un dossier, tous les fichiers contenus dans ce dossier, et le dossier lui-même seront désindexés.

- **Déplacer la branche sur laquelle pointe HEAD**

La commande git reset commit permet de déplacer la branche sur laquelle pointe HEAD sur le commit sélectionné.
Nous avons vu que HEAD contient la référence vers la branche sélectionnée, par défaut main.
Par défaut git reset commit va donc déplacer la branche main sur le commit spécifié.

Prenons par exemple la situation suivante :

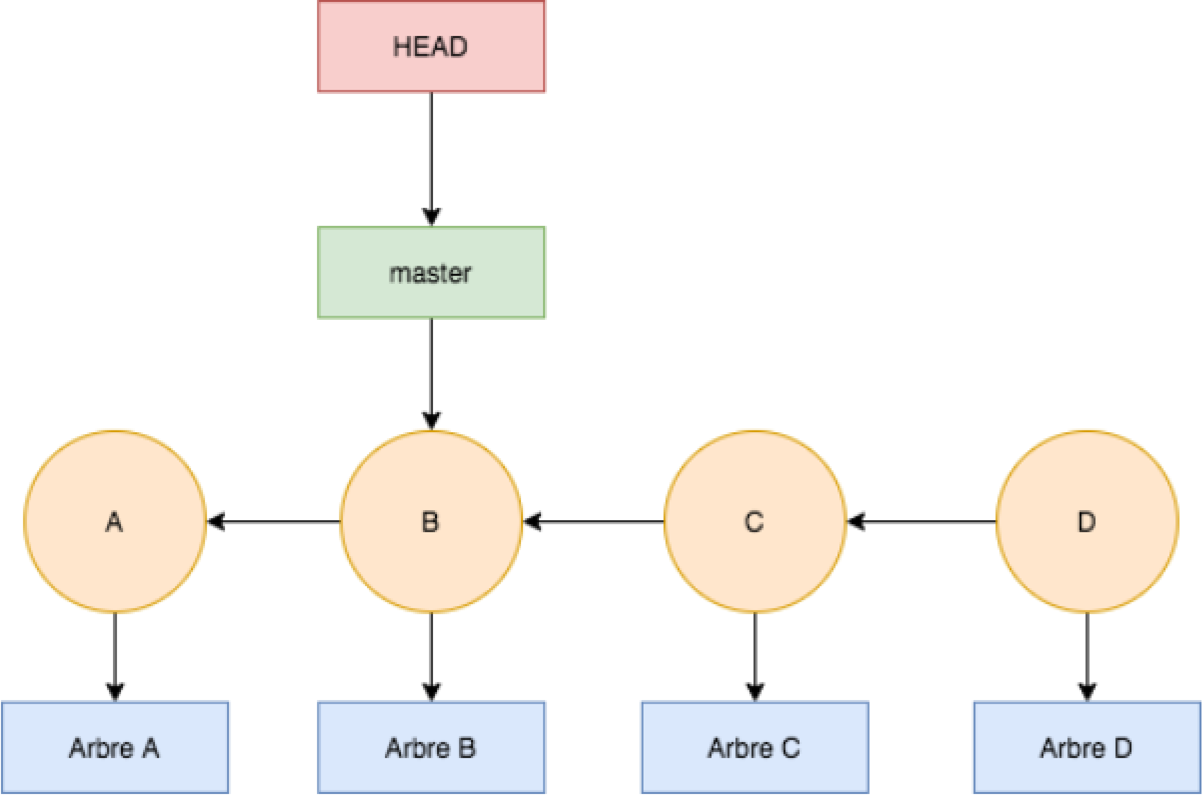


Que se passe t-il si nous faisons :

git reset B

Où B est le hash du commit B.

Voici le résultat :



HEAD pointe toujours sur main :

cat .git/HEAD

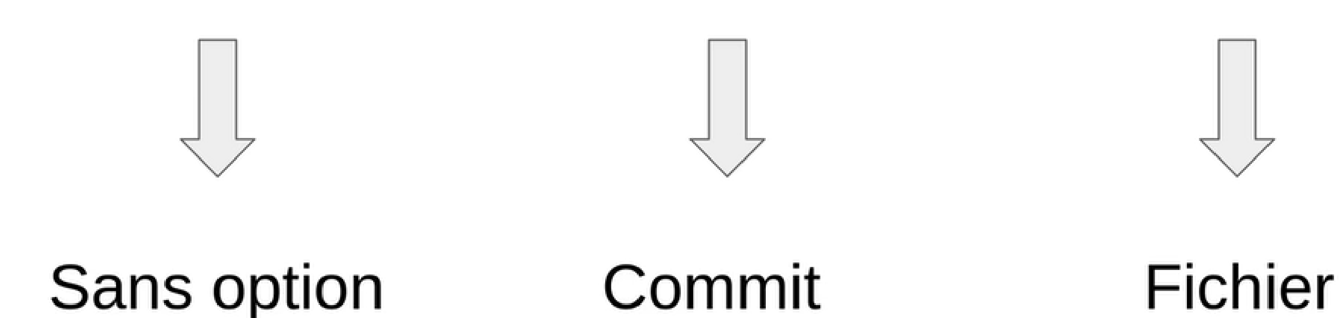
Donne toujours :

ref: refs/heads/main

Pratiquons maintenant :

Le git reset peut s’utiliser dans 3 options :

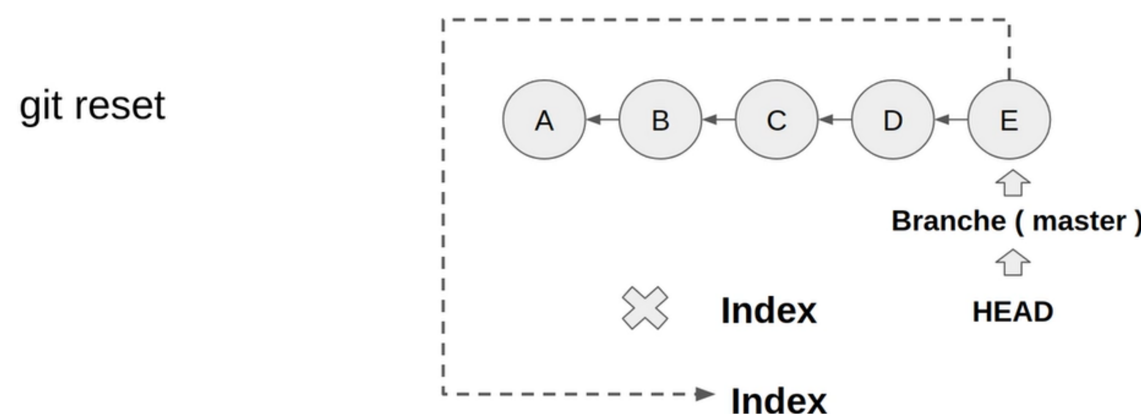
Git reset



Le git reset sans option :

A partir du moment qu’on a fait un git reset on va supprimer le contenu de l’index et le faire pointer sur le dernier commit donc tous les fichiers que vous avez mis dans le staging vont etre retirés :

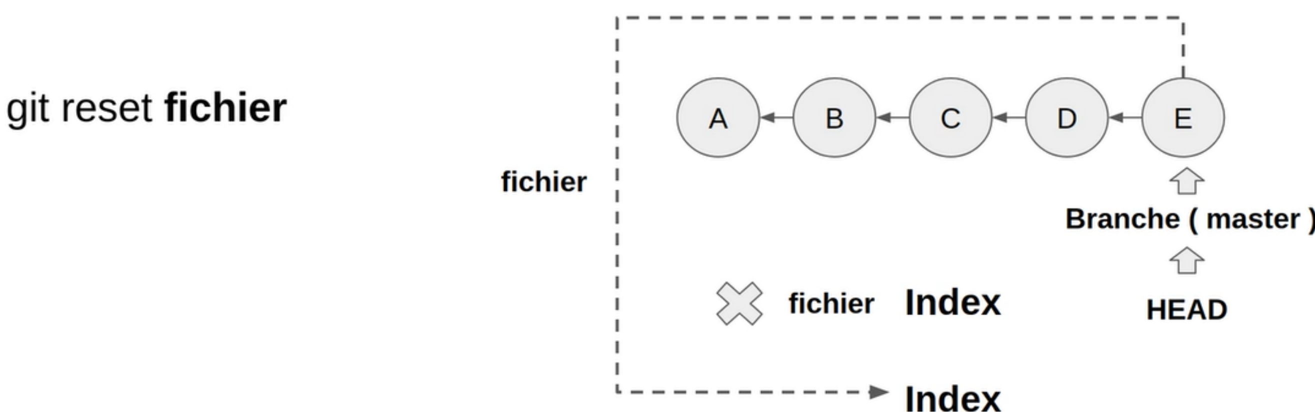
Git reset => sans option



Le git reset sur un fichier :

Il fait exactement la même chose que le reset sans option sauf que lui il remplace juste le fichier :

Git reset => avec un fichier



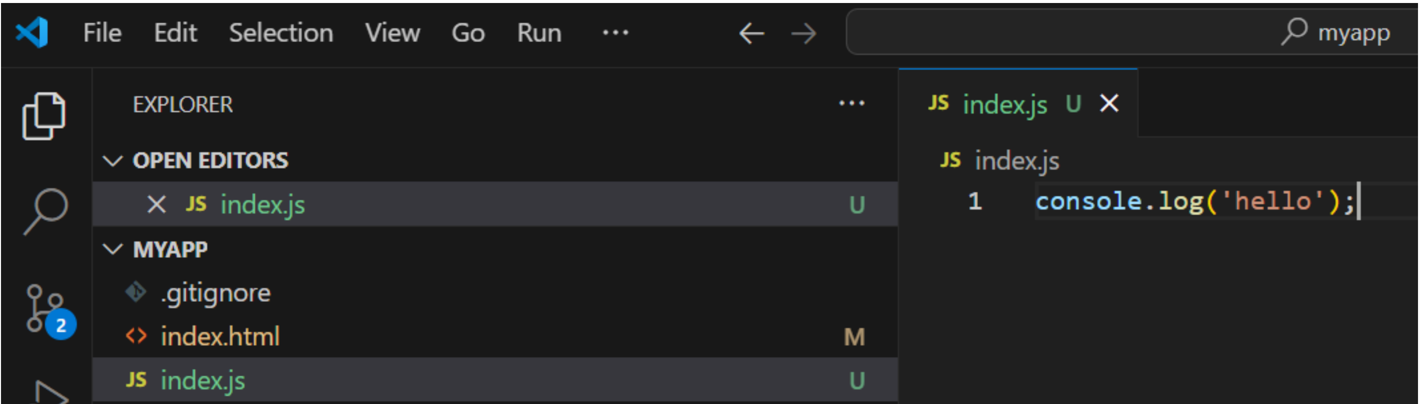
Donc au final la commande va unstage « **fichier** » et le remplacer par sa version lors du dernier commit.

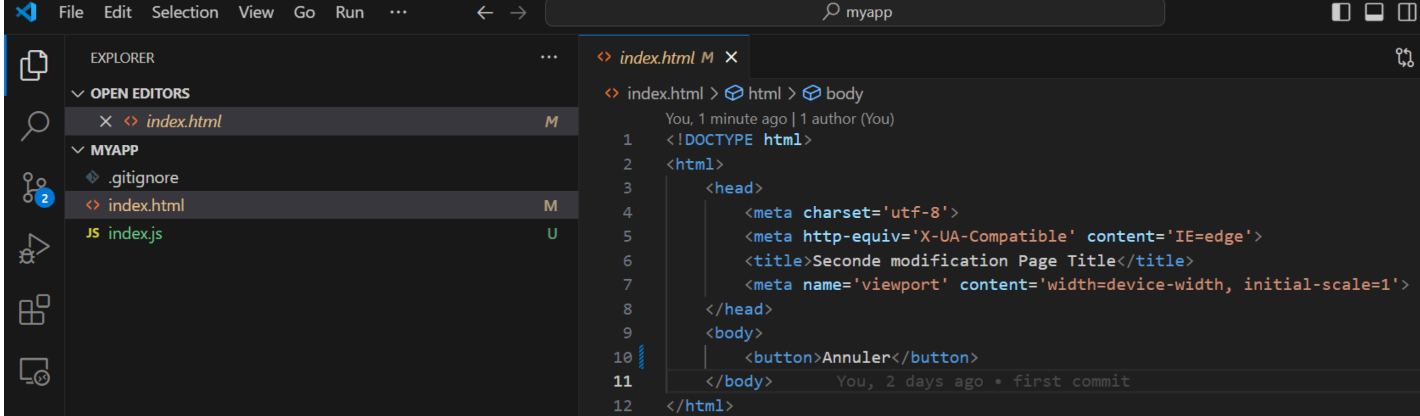
Revenons à notre projet.

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Ajouter le fichier **index.js** et modifier le fichier **index.html**.





git reset

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset
Unstaged changes after reset:
M       index.html
```

git add index.html index.js
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html index.js

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
        new file:   index.js
```

git reset
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset
Unstaged changes after reset:
M       index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (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)
        index.js
```

Remettons le projet dans son état initial :

git add index.html index.js
git status
git reset index.html

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html index.js

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
        new file:   index.js


elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset index.html
Unstaged changes after reset:
M       index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.js

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
```

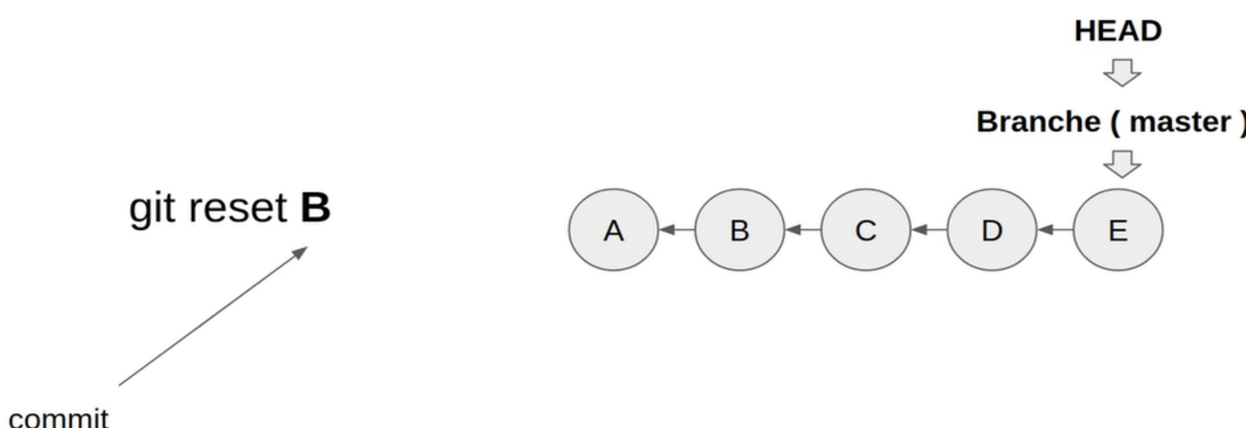
Si je veux remettre le fichier **index.html** à son contenu initial il suffit de faire :

git checkout index.html
git status

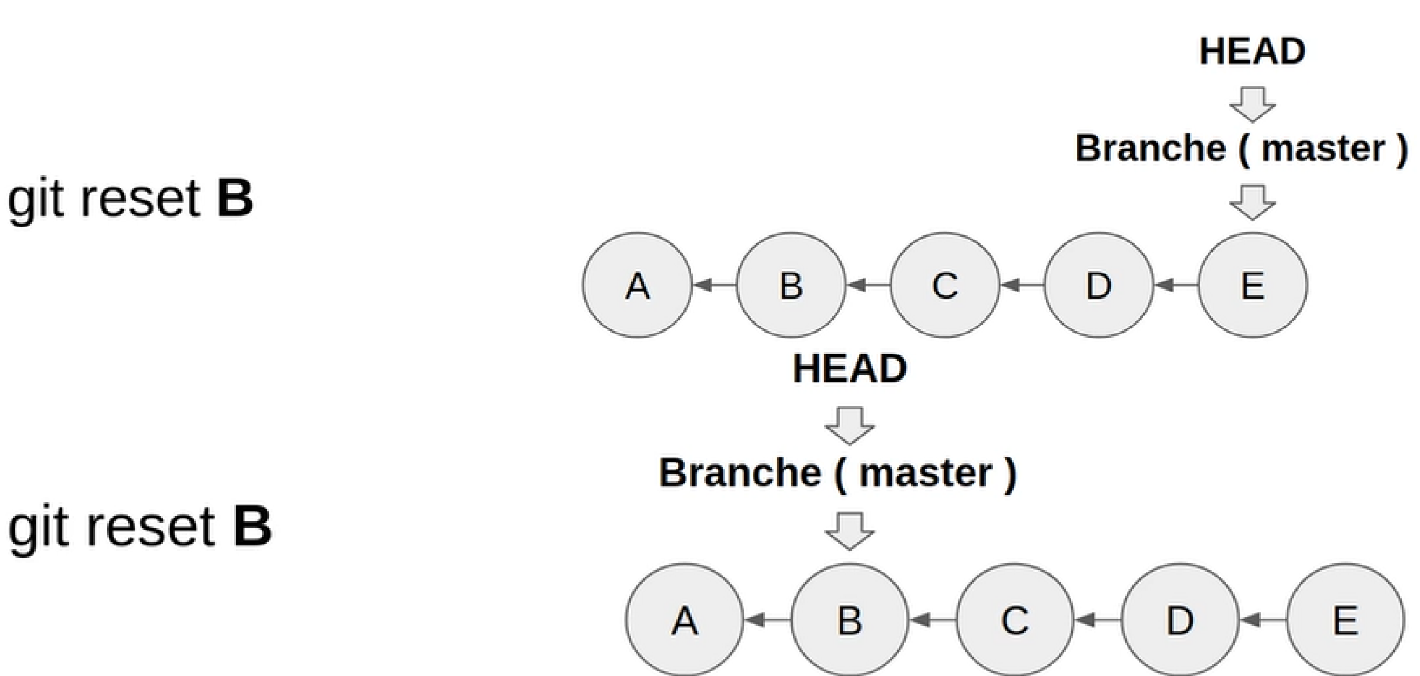
```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout index.html
Updated 1 path from the index

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.js
```

Regardons maintenant à propos di git reset sur un commit

Git reset => commit

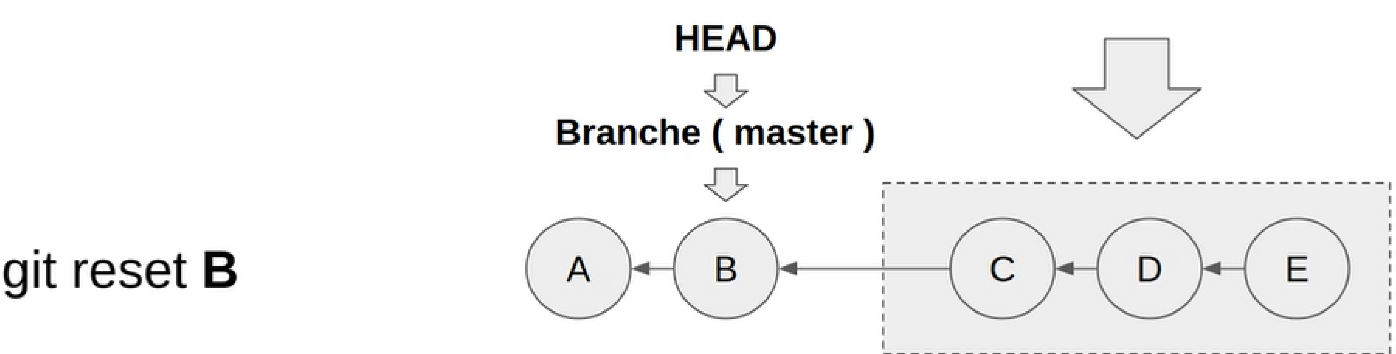


Contrairement à la commande git revert la commande git reset deplace le commit et le Head.

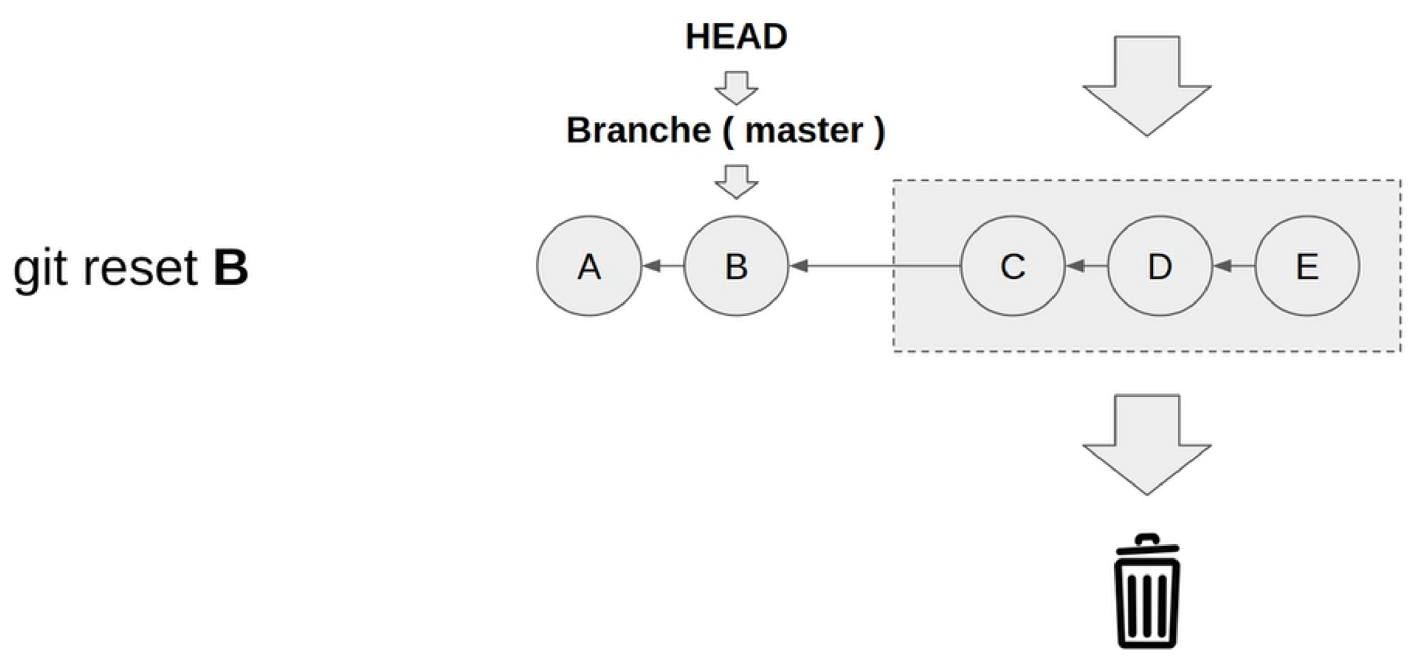


EL Hadji Gaye

On a plus aucun moyen d'acceder aux commit **C**,**D** et **E**. Ils deviennent des commit orphelins.



Ces commit orphelins seront prochainement supprimé par Git.

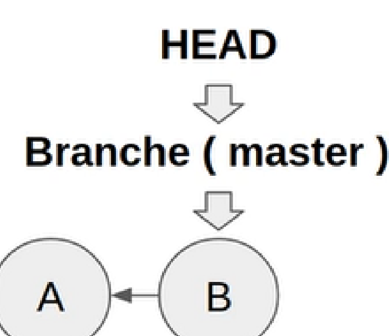


Quelques option avec la commande reset

git reset B –options

git reset B --soft
git reset B –mixed
git reset B --hard

git reset **B --options**



	Working directory	Index
--soft	✓	✓
--mixed	✓	✗
--hard	✗	✗

Par défaut c'est l'option **mixed** qui est appliqué.

		Working directory	Index
	--soft	✓	✓
défaut	--mixed	✓	✗
	--hard	✗	✗

Créer le fichier **index.js**, le mettre dans l'index et faire une modification dans **index.html**.

Retournons à notre projet et faisons un git status :

git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.js

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
```

git reset HEAD
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset HEAD
Unstaged changes after reset:
M       index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (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)
        index.js

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

git add index.js
git status

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.js

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.js

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
```

git reset HEAD --soft
git status

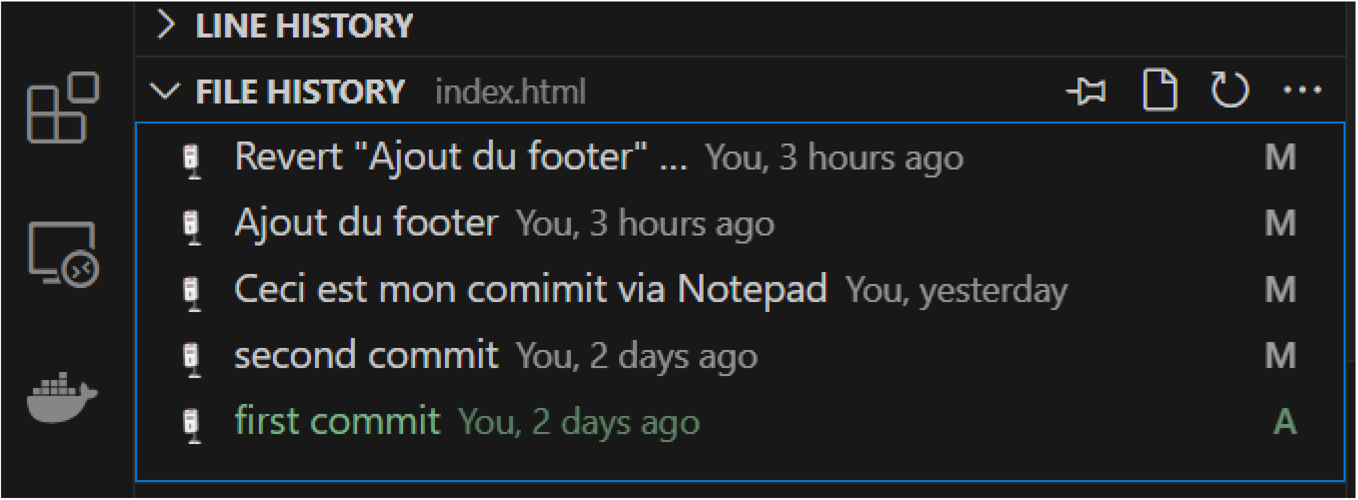
```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset HEAD --soft

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.js

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
```

Aucun changement ni dans le **working directory** ni dans l'index staged.

git reset HEAD --hard
git status

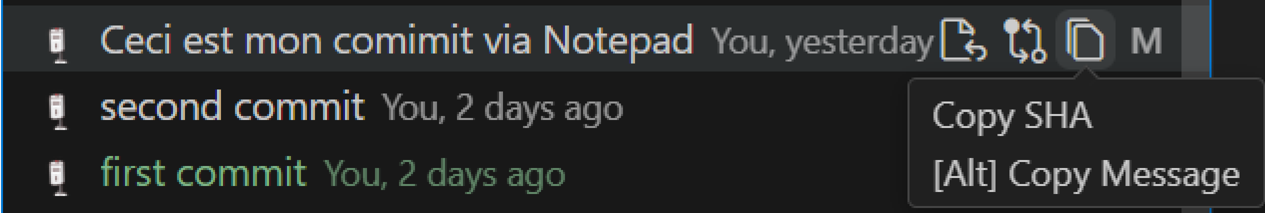
```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset HEAD --hard
HEAD is now at 247dacd Revert "Ajout du footer"
```

On pointe effectivement sur le dernier commit et notre fichier **index.js** a été supprimé.
On peut faire un reset sur un commit en particulier pour cela regardons notre historique GitLens.



Admettons qu'on veuille retourner au commit « Ceci est mon commit Notepad » ce qui veut dire nous allons supprimer les commit «Revert "Ajout du footer"» et «Ajout du footer».

Recupèrer le Hash du commit « Ceci est mon commit Notepad » :

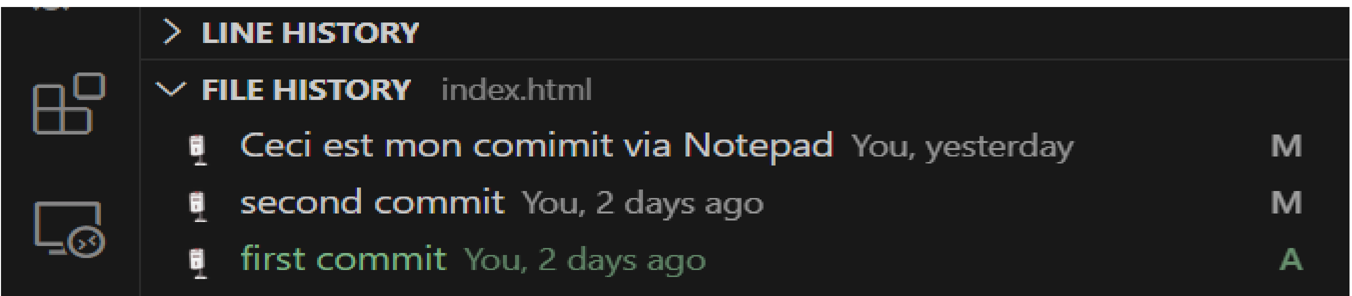


Le commit « Ceci est mon commit Notepad » a pour Hash
7927c8258ebd6d9954f1596e81a84faf3d253fa2

git reset 7927c8258ebd6d9954f1596e81a84faf3d253fa2 --hard

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git reset 7927c8258ebd6d9954f1596e81a84faf3d253fa2 --hard
HEAD is now at 7927c82 Ceci est mon comimit via Notepad
```

Au niveau de **GitLens** on obtient :



Vérifions avec un **git log** :

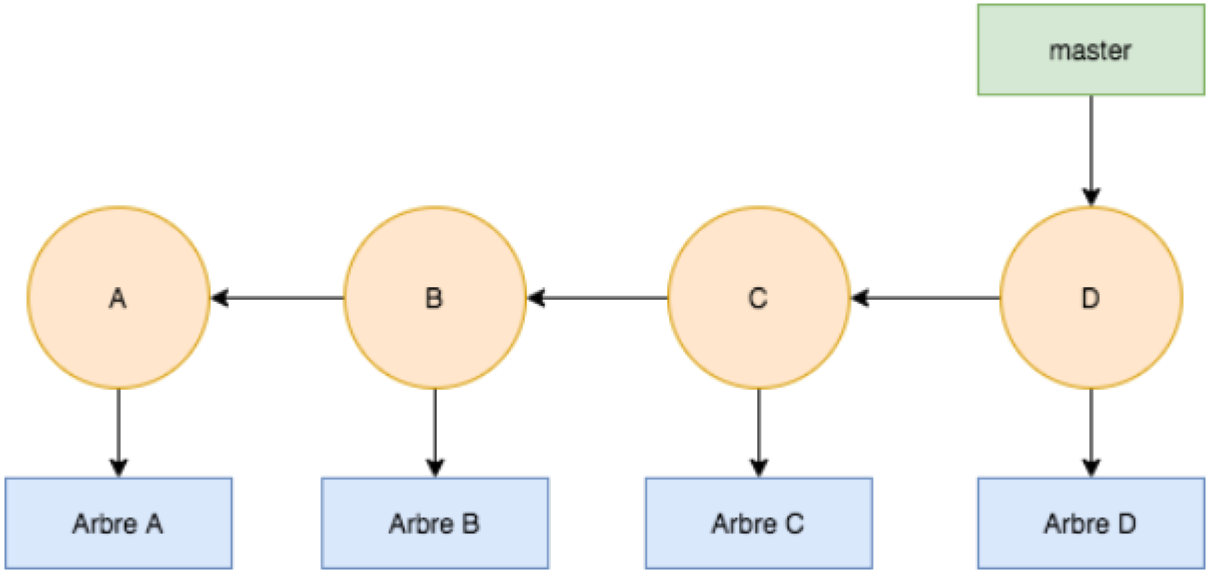
```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Mon Sep 23 06:42:53 2024 +0200

    Ceci est mon comimit via Notepad

commit 40606b1e8e887c4944146396b108e41e18dcf83f
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Mon Sep 23 05:12:27 2024 +0200

    commit gitignore

commit c8d70e143acbe329abe27089fc8ab75a048ee5b0
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Sun Sep 22 21:20:23 2024 +0200

    second commit
```

III) Les branches avec Git

1. *Introduction*

Qu'est-ce qu'une branche ?

Nous avons vu que chaque commit (sauf le premier, appelé commit racine) a une référence à son commit parent, formant ainsi une ligne ou une chaîne de commits :



Parfois vous avez cependant besoin de travailler sans impacter cette ligne principale de développement, appelée branche main.

Les cas d'utilisation des branches

Des cas d'utilisation typique des branches sont principalement :
Premièrement, le développement d'une nouvelle fonctionnalité.
Deuxièmement, effectuer des tests de fonctionnalité sans risquer d'interférer avec la version en production.
Grâce aux branches, la ligne principale des commits n'est pas du tout impactée, ce qui permet d'avoir une branche propre généralement réservée à la mise en production : main.

Nous verrons en détails dans un chapitre dédié quel est le processus de développement classique avec une équipe qui utilise Git comme système de contrôle de versions.

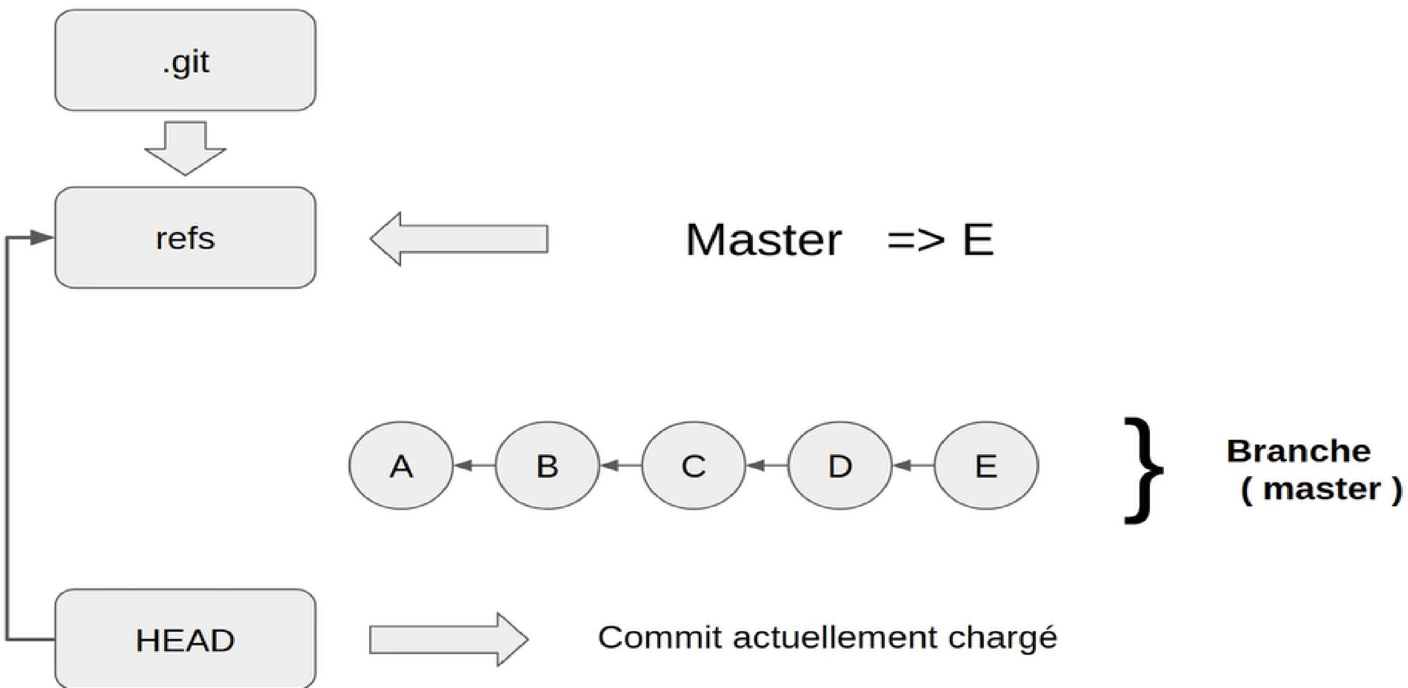
Recommandations sur les branches

Les branches sont une des fonctionnalités principales de Git, comme elles n'ont aucun coût d'espace mémoire (il s'agit simplement d'un pointeur sur un commit), elles sont très recommandées.
En effet, elles ne prennent que 41 caractères (40 caractères du hash et un retour à la ligne), on dit donc que les branches en Git sont "gratuites", c'est-à-dire qu'elles ne coûtent quasiment aucune mémoire.
Ce qui prend de la mémoire en Git, comme vous le savez, c'est de sauvegarder un instantané du projet, et donc un commit. Il vaut donc mieux éviter de commit trop fréquemment : les commits sont conçus pour être des sauvegardes propres.

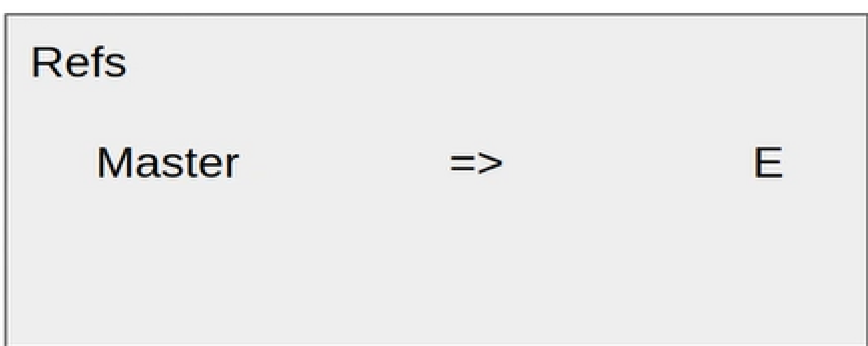
N'hésitez donc pas en revanche à vous servir de branches ! Sur un projet en équipe, il n'est pas rare d'avoir une dizaine de branches actives sur un projet.

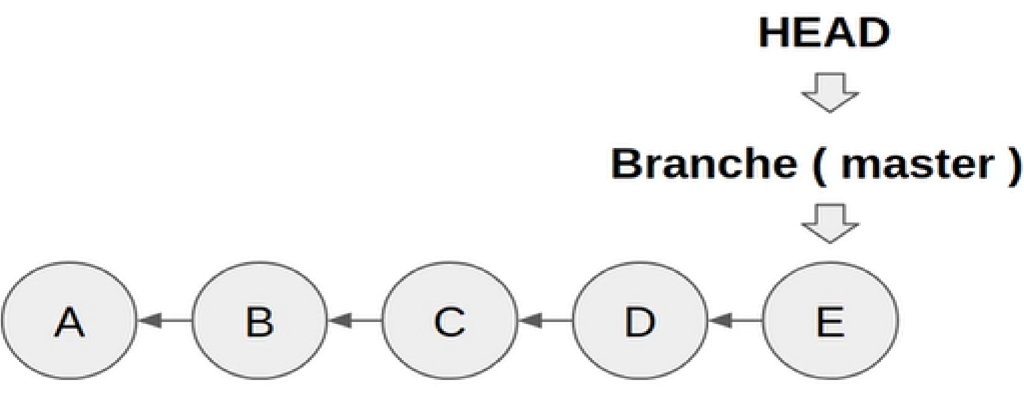
Pratiquons un peu maintenant !

Précédamment nous avions vu pour la branche master :

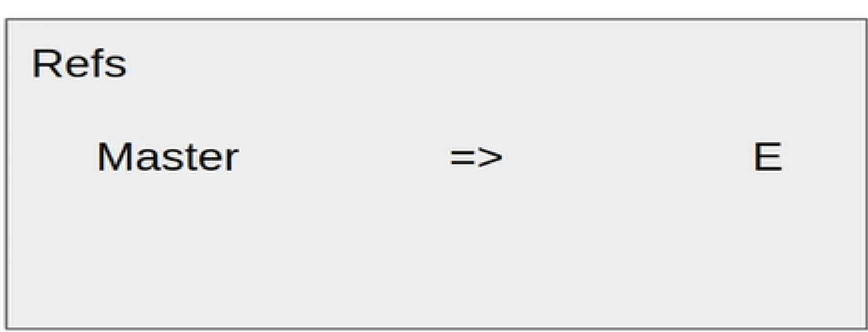


Supposons qu'on se retrouve dans une situation où le commit **E** est le dernier commit :





Le **Head** pointe vers la branche master qui pointe vers le commit **E**.
Lorsqu’un nouveau commit arrive on obtient :



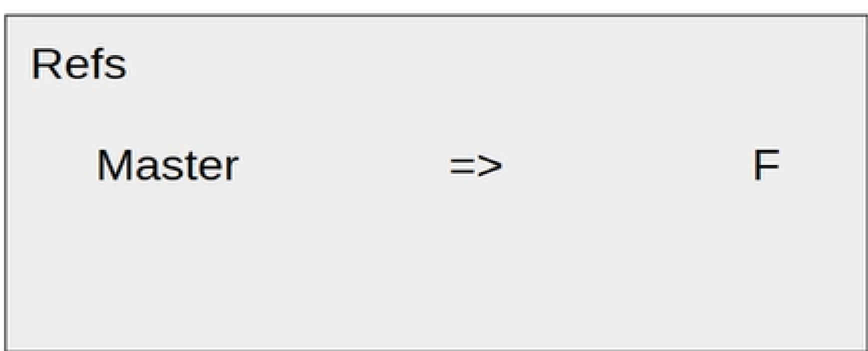
Git commit -m “new commit”

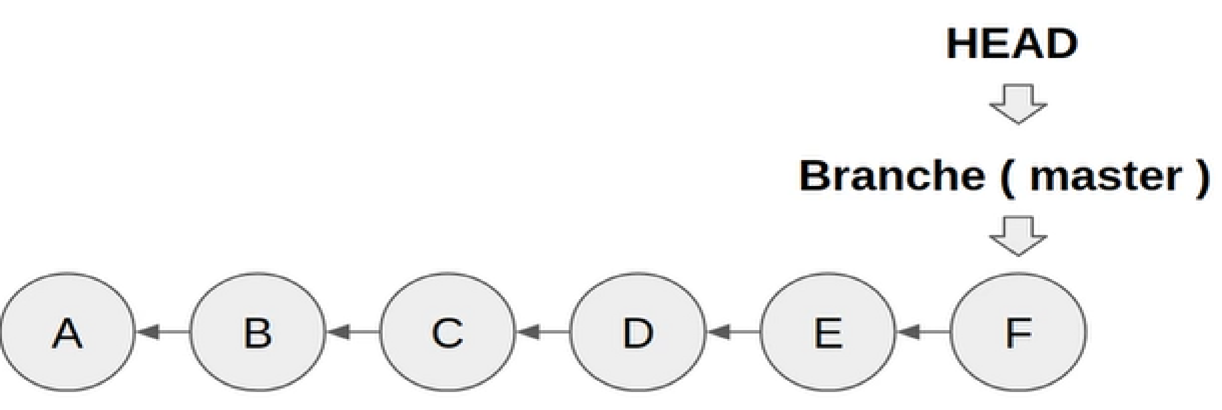
HEAD

Branche (master)

A B C D E F

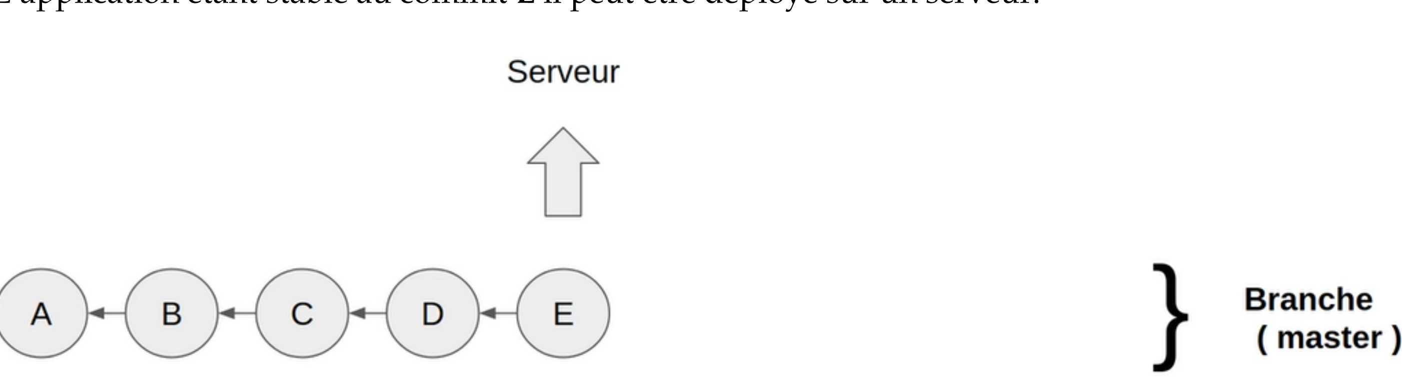
Le HEAD se deplace ensuite vers le commit F.



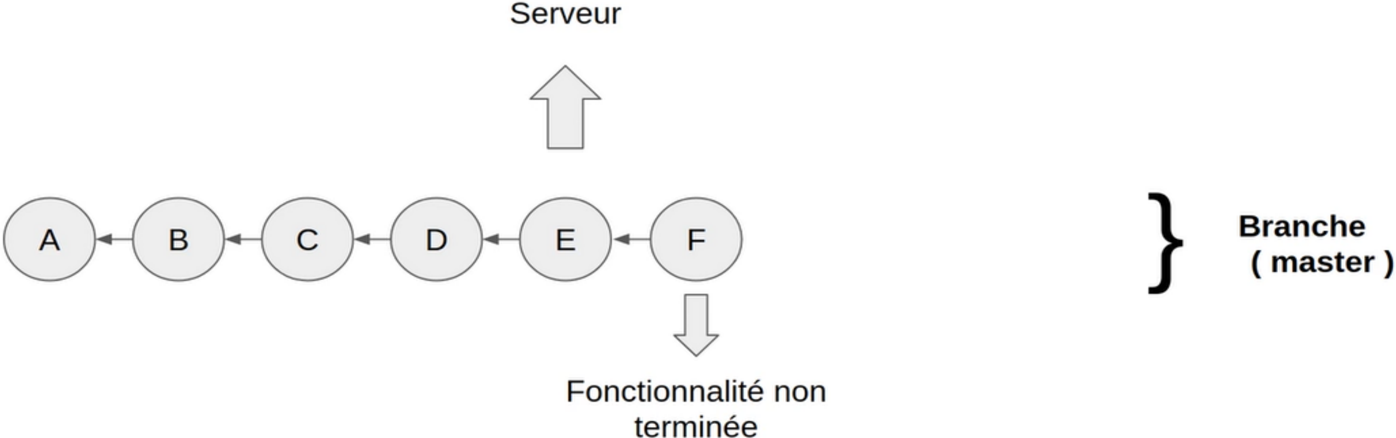


Une fois que toutes ces eclairecissements sont terminés revenons sur une hypothese d'une application qui a été développé à travers des commit A,B,C,D et E.

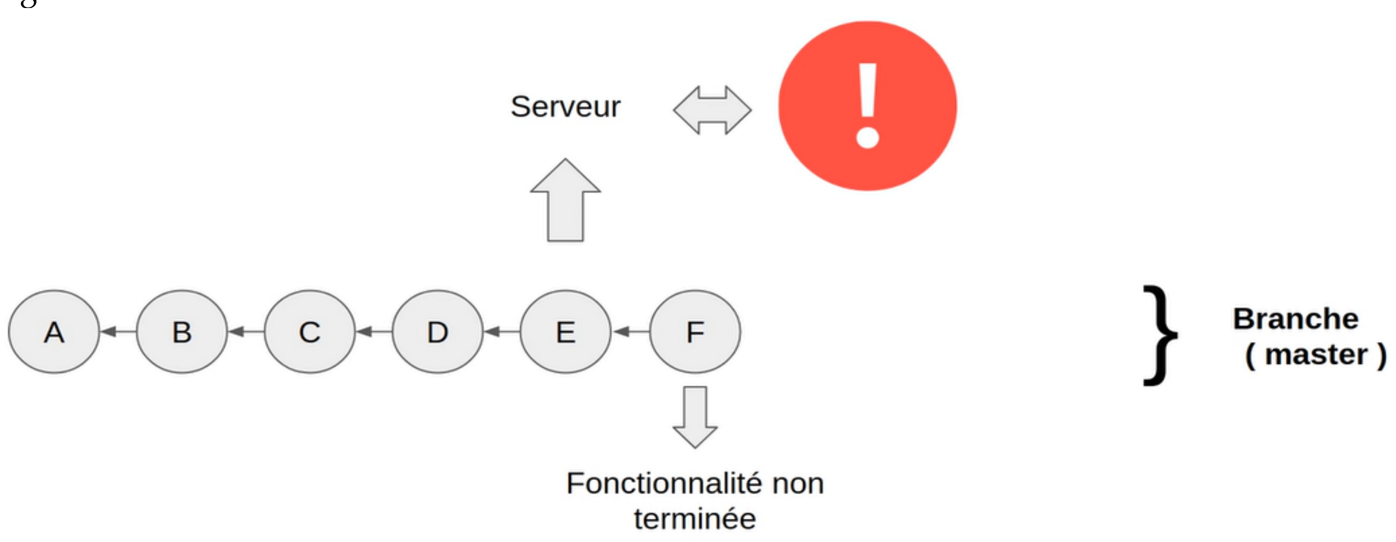
L'application etant stable au commit **E** il peut etre déployé sur un serveur.



Une fois que le code a été déployé on decide d'ajouter à notre code un feature assez conséquent.



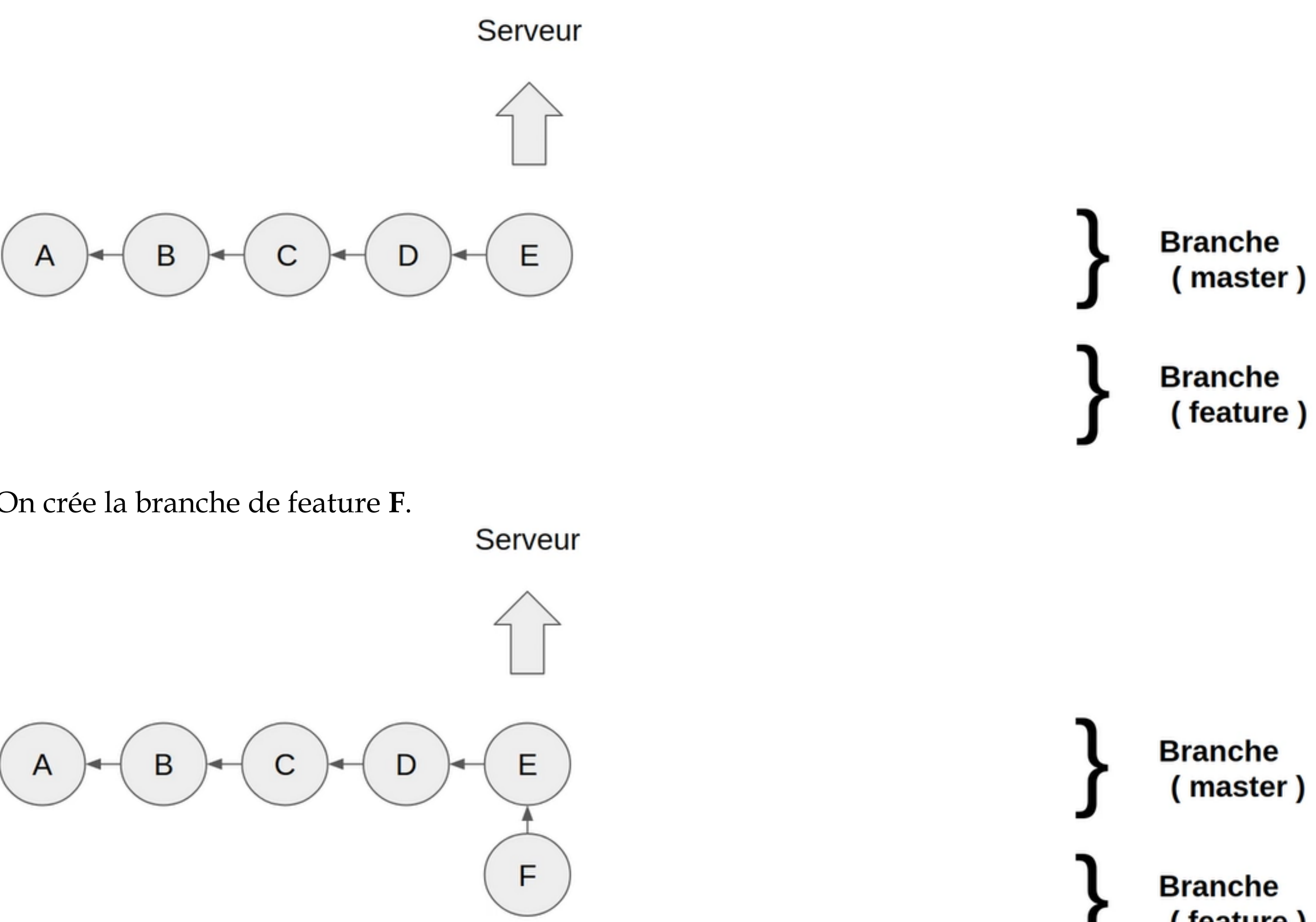
Malheureusement entre temps la situation s'empire car sur votre fonctionnalité en E contient un bug.



Pour cette problèmatique nous aurons deux issus :

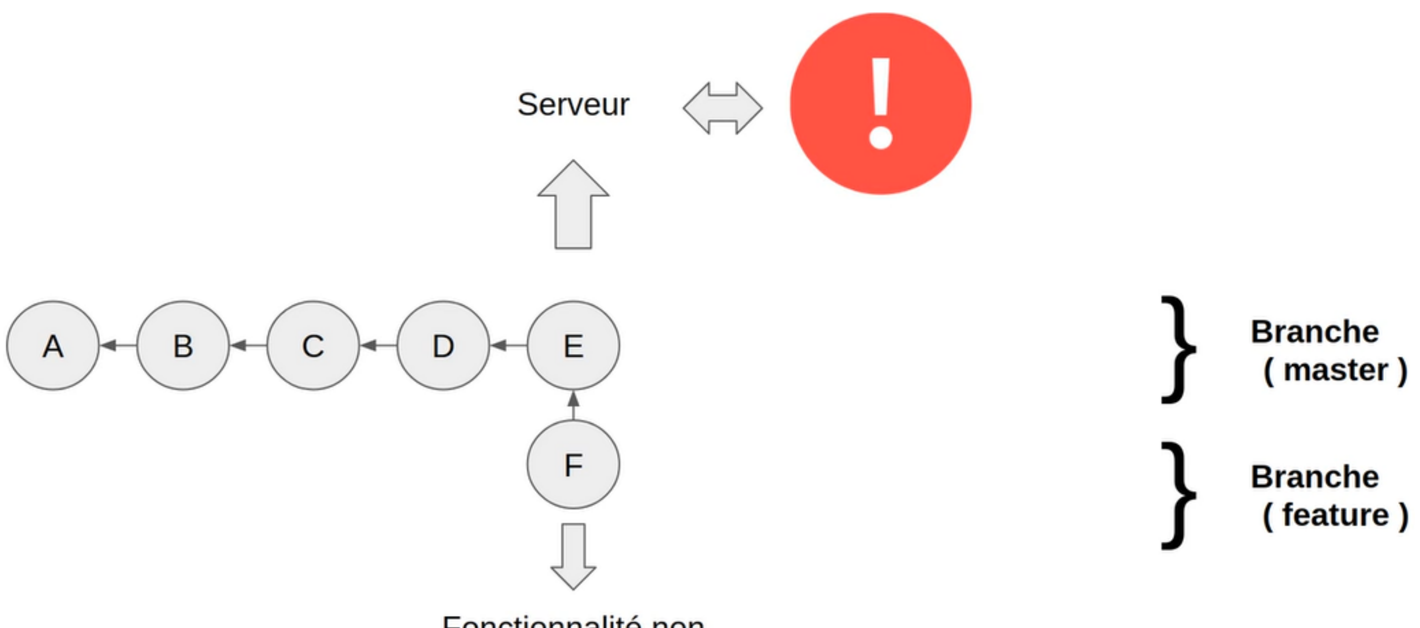
- Soit on retourne au commit E et on corrige et dans ce cas on perd ce qu'on avait fait sur F.
- Soit on accelere les developpement de F et on fait patienter les utilisateur pour les bugs.

Si on choisit d'utiliser des branches nous aurons :

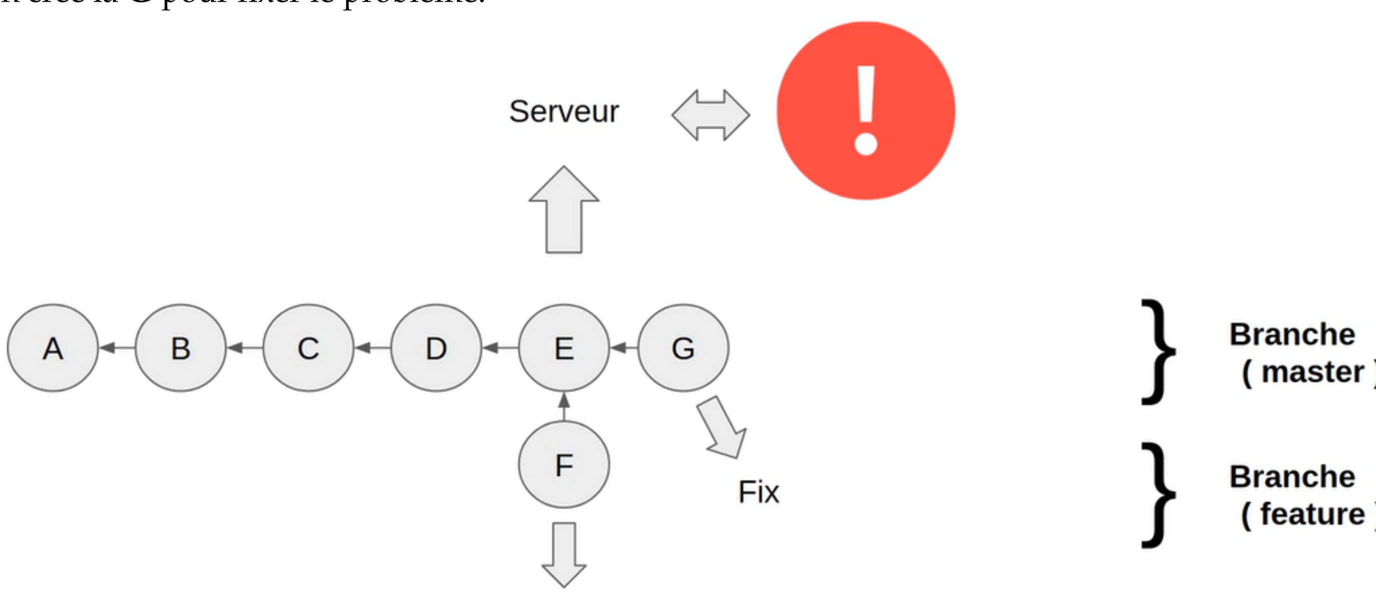


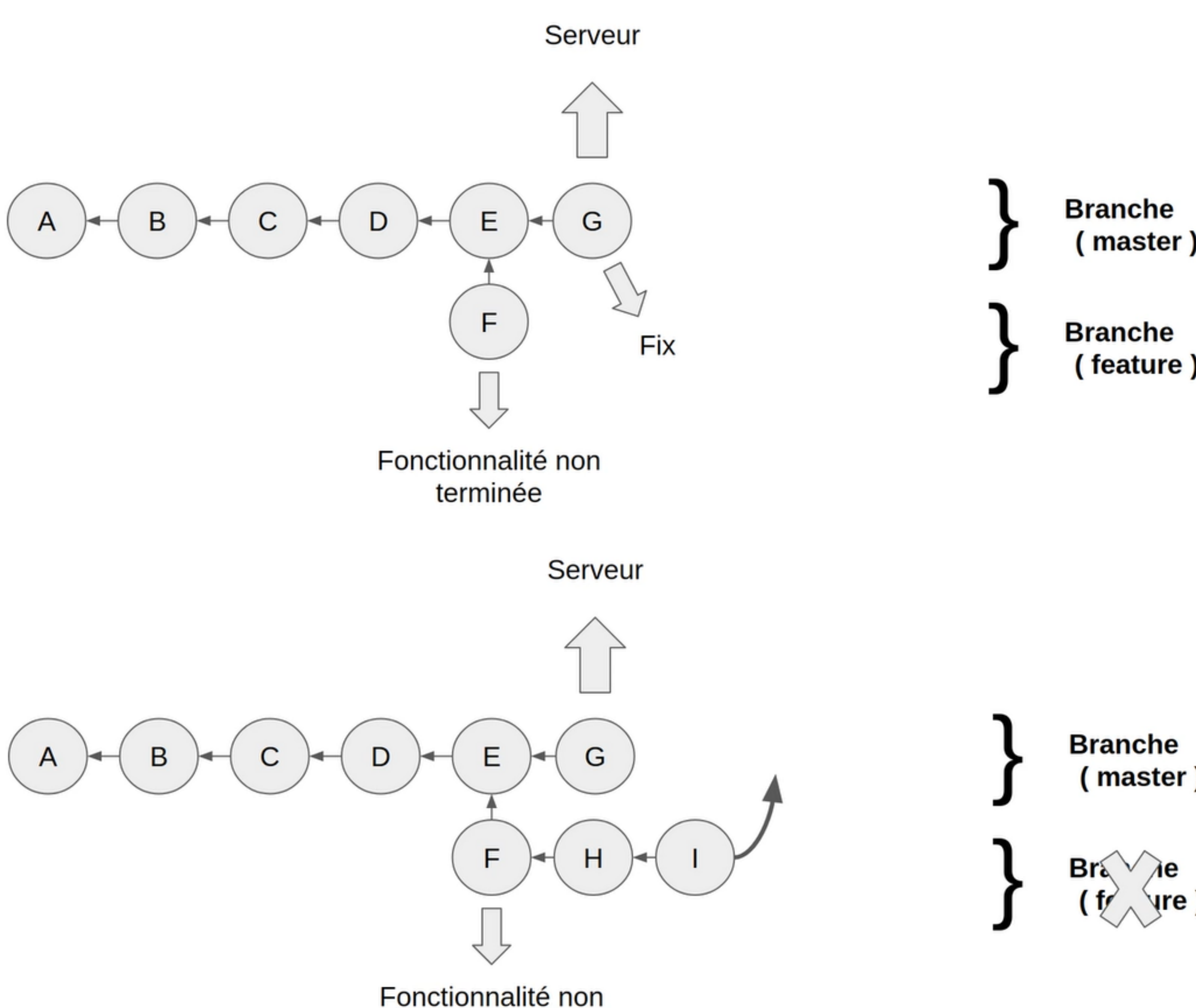
On crée la branche de feature **F**.

Toujours pareil pendant qu'on travaille dans notre feature on se retrouve avec les même bugs dans le serveur de production.



On crée la **G** pour fixer le problème.





Fix

Features

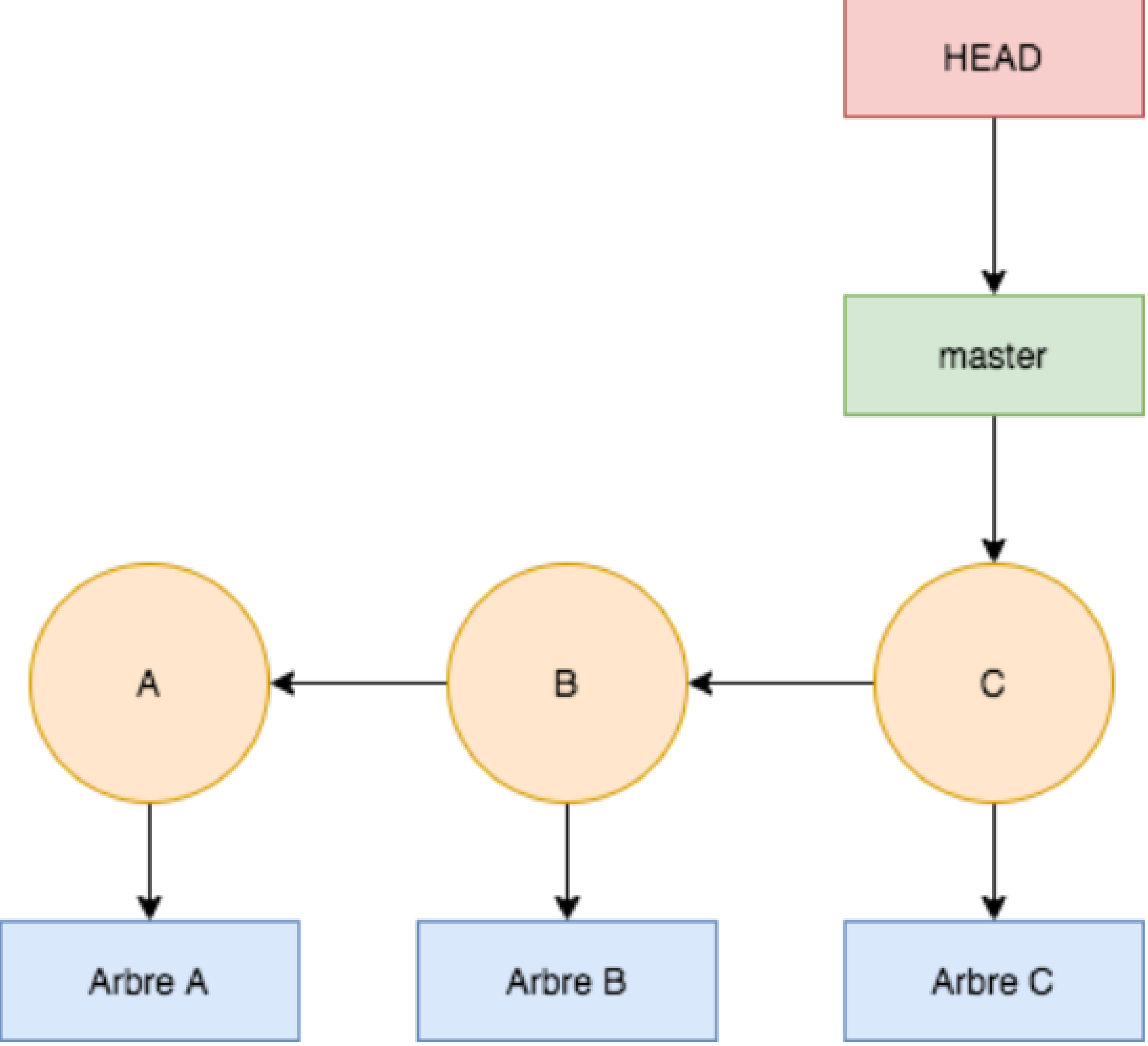
Branche de développement

Release candidate

2. *Lister et créer des branches*

Créer une branche en Git est extrêmement simple, il suffit de faire : git branch nom.

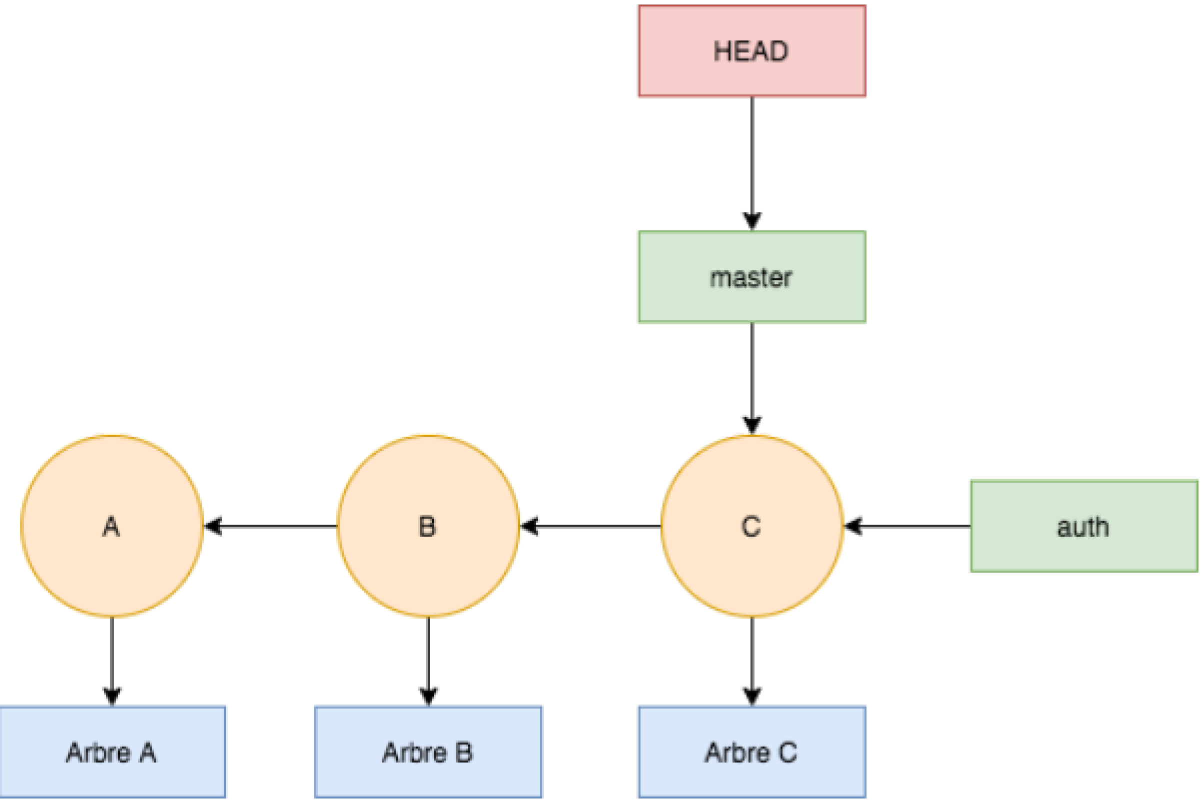
Prenons notre exemple et expliquons le fonctionnement :



Nous sommes à la version correspondant au commit C et nous souhaitons démarrer un nouveau développement qui va prendre du temps, par exemple développer un système d'authentification.

Nous allons donc créer une branche en faisant :

git branch auth



Nous avons pour le moment simplement créé un nouveau pointeur sur le commit C, qui est notre nouvelle branche.

Nous pouvons vérifier simplement :

cat .git/refs/heads/auth
cat .git/refs/heads/main

Qui donneront toutes les deux le même hash, qui est celui du commit C dans notre exemple.

Ces deux références pointent donc exactement sur le même commit.

Attention ! Pour le moment HEAD pointe toujours sur main, nous ne nous sommes pas positionné sur la nouvelle branche auth, nous l'avons simplement créée.

D'ailleurs si vous faites :

git log

Vous verrez bien :

commit Hash_C (HEAD -> main, auth)

Il est bien indiqué que HEAD pointe sur main, et que main et auth pointe sur le commit.

Lister les branches

Avec Git il est très simple de visualiser toutes les branches locales d'un projet en une commande :

git branch

Si vous ne passer pas de nom, git branchva simplement lister tous les noms des branches. Dans notre exemple, nous aurons :

auth
*** main**

L'étoile indique que HEAD pointe actuellement sur la branche main.

Cela signifie que si vous faites des modifications et que vous les sauvegardées dans un commit, ce dernier se placera sur la branche main.

Vous pouvez également obtenir plus d'informations avec git branch -v.

git branch -v

Cette option vous donnera notamment le début du hash du commit sur lequel pointe chaque branche, ainsi que le début du message de validation, par exemple :

auth   d9d259c nouveau commit sur auth
* main 9e633d5 Premier commit

Modifier le nom d'une branche

Modifier le nom d'une branche est très simple puisque c'est simplement le nom d'un fichier situé dans **.git/refs/heads**.

Il suffit de faire :

git branch -m “nouveauNom”

Supprimer une branche

Pour supprimer une branche il suffit de faire :

git branch -d branche

Si la branche contient des modifications qui n'ont pas été fusionnées (nous verrons en détails la fusion dans les prochaines leçons), il faudra forcer la suppression (mais attention tous les changements et tous les commits de la branche seront perdus) :

git branch -D branche

L'option -D est un raccourci pour --delete --force et signifie "forcer la suppression de la branche".

Nous passons maintenant à la pratique.

git branch

git branch ⇨ Listes des branches

git branch nomdebranche

git branch foo

Refs

Master	=>	F
Foo	=>	F

HEAD

Branche (master)

A ← B ← C ← D ← E ← F

Branche (foo)

Refs		
Master	=>	F
Foo	=>	F

Git commit -m “new commit”

HEAD

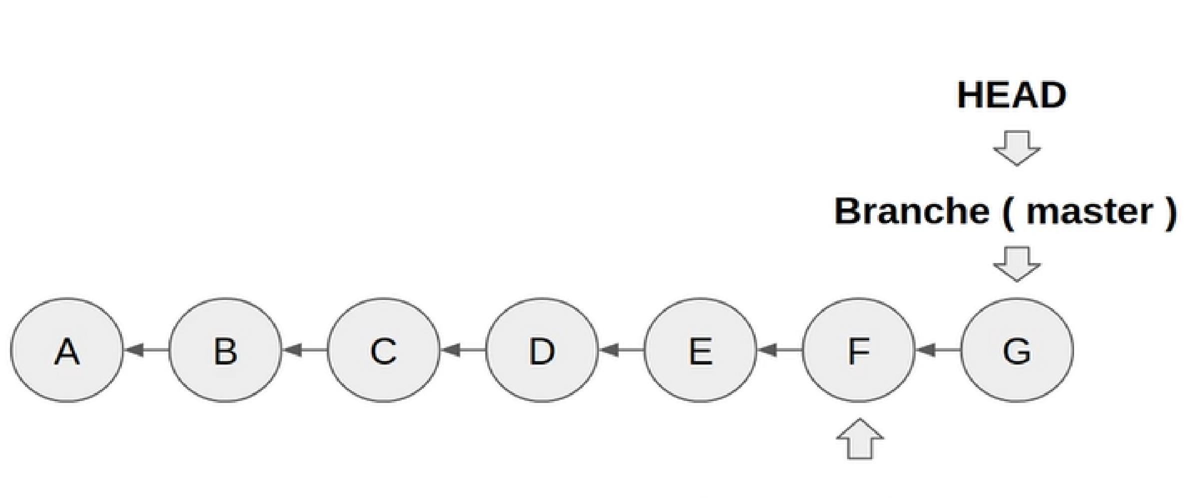
Branche (master)

A ← B ← C ← D ← E ← F ← G

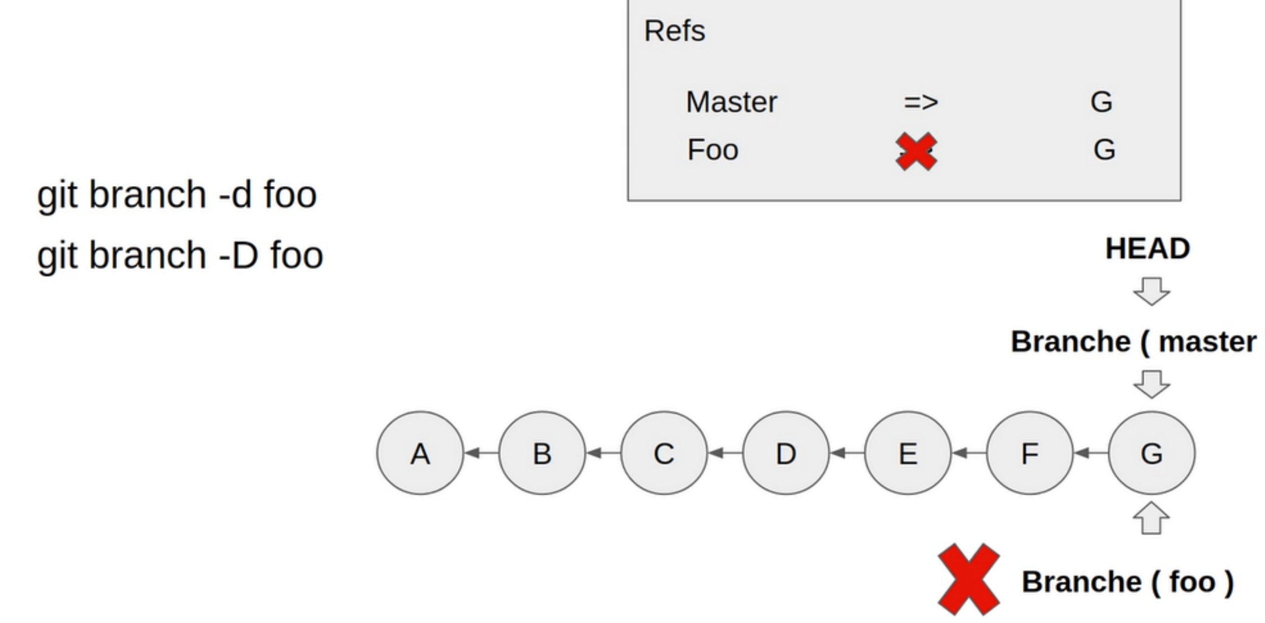
Branche (foo)

HEAD ⇒

Refs		
Master	=>	G
Foo	=>	F



git branch unebranche -d (-D)



git branch -m nouvellebranche

Revenons à notre projet avec les commande ci-dessous :

```
git branch
git branch foo
git branch
ls
git branch -m master2
git branch
git branch -m master
git branch
git branch -d foo
git branch
```

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
* master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch foo

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
  foo
* master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch -m master2

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master2)
$ git branch
  foo
* master2

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master2)
$ git branch -m master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
  foo
* master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch -d foo
Deleted branch foo (was 7927c82).

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
* master
```

Nous allons voir ce qu'il en ai au niveau de références.
Créer à nouveau la branche **foo**.

git branch foo
cd .git
ls
cd refs
ls
cd heads
ls
cat foo
cat master
git log -1

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch foo

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ cd .git

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ ls
COMMIT_EDITMSG  config  description  HEAD  hooks/  index  info/  logs/  objects/  ORIG_HEAD  refs/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ cd refs

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs (GIT_DIR!)
$ ls
heads/  tags/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs (GIT_DIR!)
$ cd heads

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs/heads (GIT_D
IR!)
$ ls
foo  master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs/heads (GIT_D
IR!)
$ cat foo
7927c8258ebd6d9954f1596e81a84faf3d253fa2

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs/heads (GIT_D
IR!)
$ cat master
7927c8258ebd6d9954f1596e81a84faf3d253fa2

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git/refs/heads (GIT_D
IR!)
$ git log -1
commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (HEAD -> master, foo)
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Mon Sep 23 06:42:53 2024 +0200

    Ceci est mon comimit via Notepad
```

Les branches **foo** et **master** pointent vers le même HEAD.

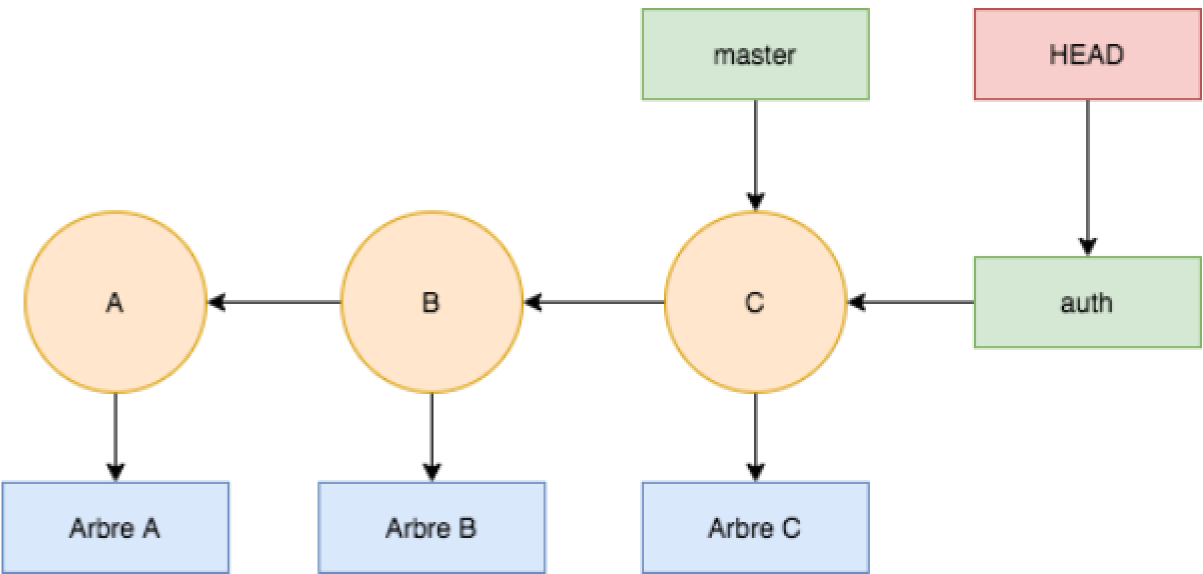
3. *Basculement de branche avec git checkout*

Pour basculer sur une branche il faut utiliser la commande git checkout branche que nous avions déjà vue.

Exemple :

git checkout auth

Nous basculons sur la branche auth et HEAD pointe alors sur la nouvelle branche :



Nous pouvons vérifier en faisant :

cat .git/HEAD

On obtient

ref: refs/heads/auth

Même chose, si nous tapons git branch, nous aurons l'étoile qui indique que HEAD pointe bien sur la branche auth :

git branch

donne comme résultat

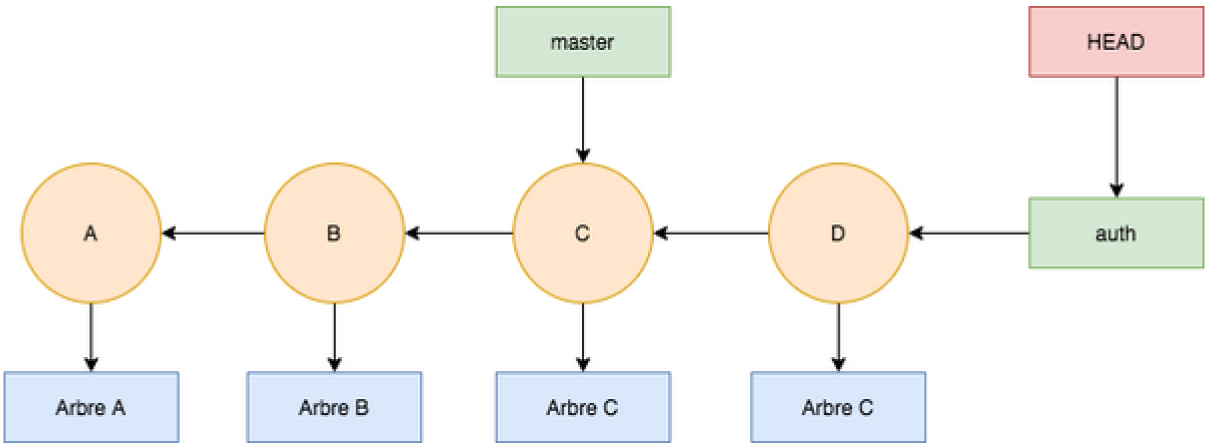
```
* auth
  main
```

Maintenant si nous faisons un nouveau commit que va t-il se passer ?

```
echo "Start auth" > auth.js
git commit -am "nouveau commit sur auth"
```

Nous avons simplement créé un nouveau fichier de test et avons indexé et sauvegardé en une seule commande grâce aux options -am de git commit.

Nous arrivons à ce point :



Nous avons effectué un nouveau commit mais sur la branche auth sur laquelle nous nous sommes placé avec git checkout auth.
Mettons maintenant que nous voulons retourner sur la branche principale pour faire un commit apportant des modifications réparant un bug.

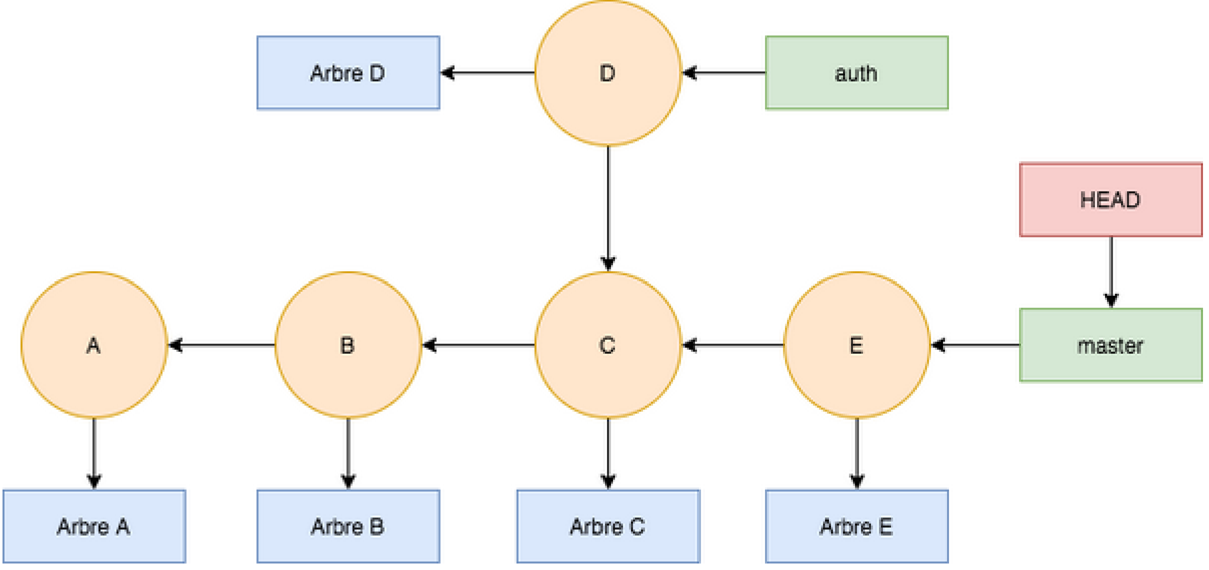
Aucun problème ! Nous faisons :

git checkout main

Ensuite, nous effectuons les modifications de réparations, puis nous faisons :

git commit -am "fix: un problème x"

Nous en sommes là :



Nous avons donc fait un commit de réparation, E dans notre exemple, qui est sur la ligne principale.

Une fois notre branche main réparée, il nous suffit de retourner sur notre branche auth pour continuer à travailler sur notre fonctionnalité :

git checkout auth

On peut aussi créer un branche et y basculer directement :

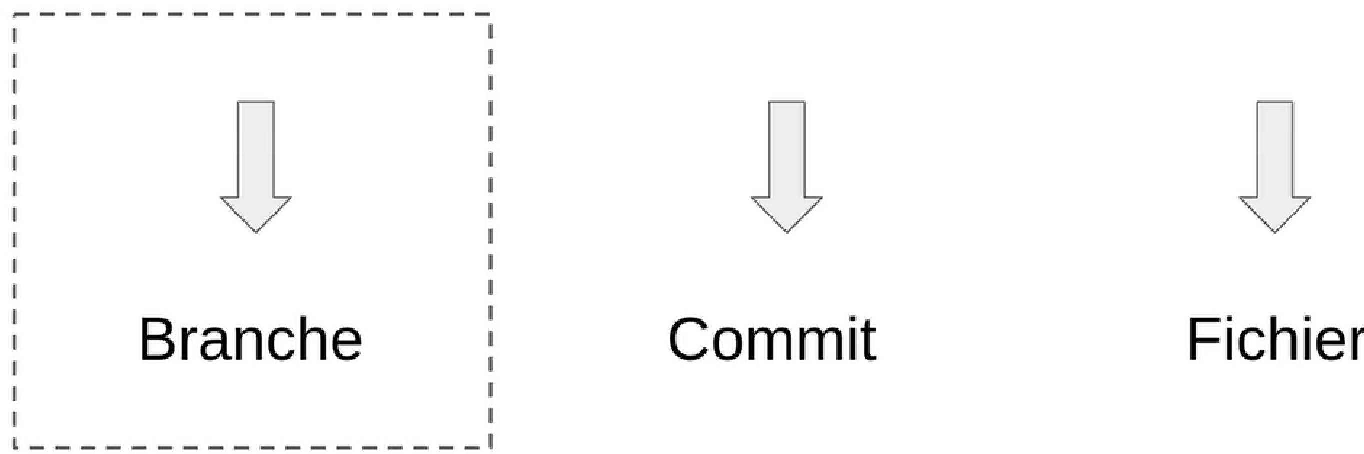
git branch feat1
git checkout feat1

Autrement, dit de créer une branche et d'y basculer HEAD immédiatement.
Cette commande est git checkout -b nomBranche :

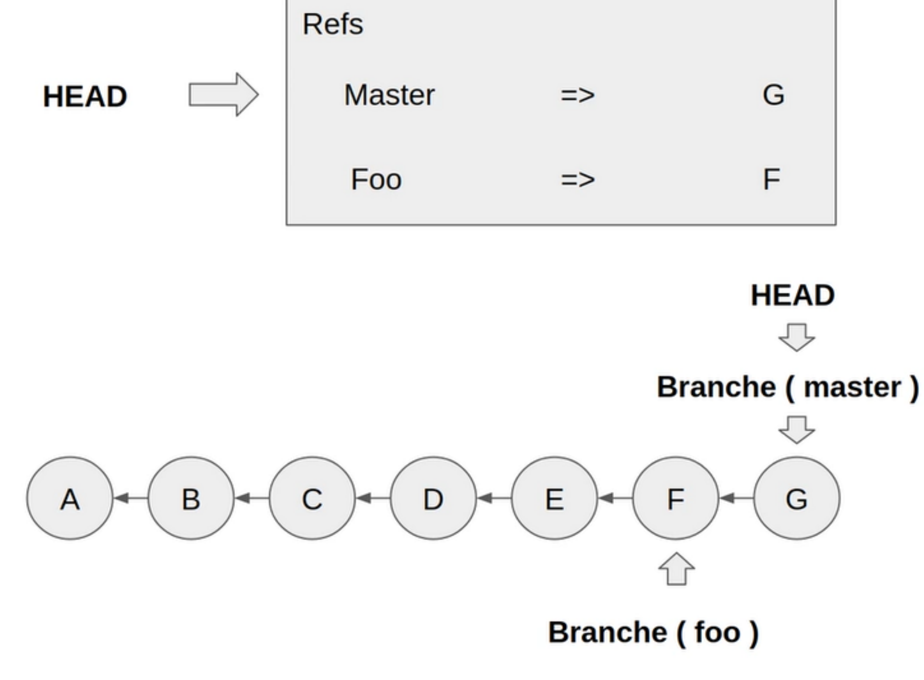
git checkout -b feat1

Passons maintenant à la pratique :

Nous avions vu précédemment qu'il était possible de faire un git checkout sur une branche, sur un commit et sur un fichier.
Nous allons maintenant nous concentrer sur le git checkout sur une branche.



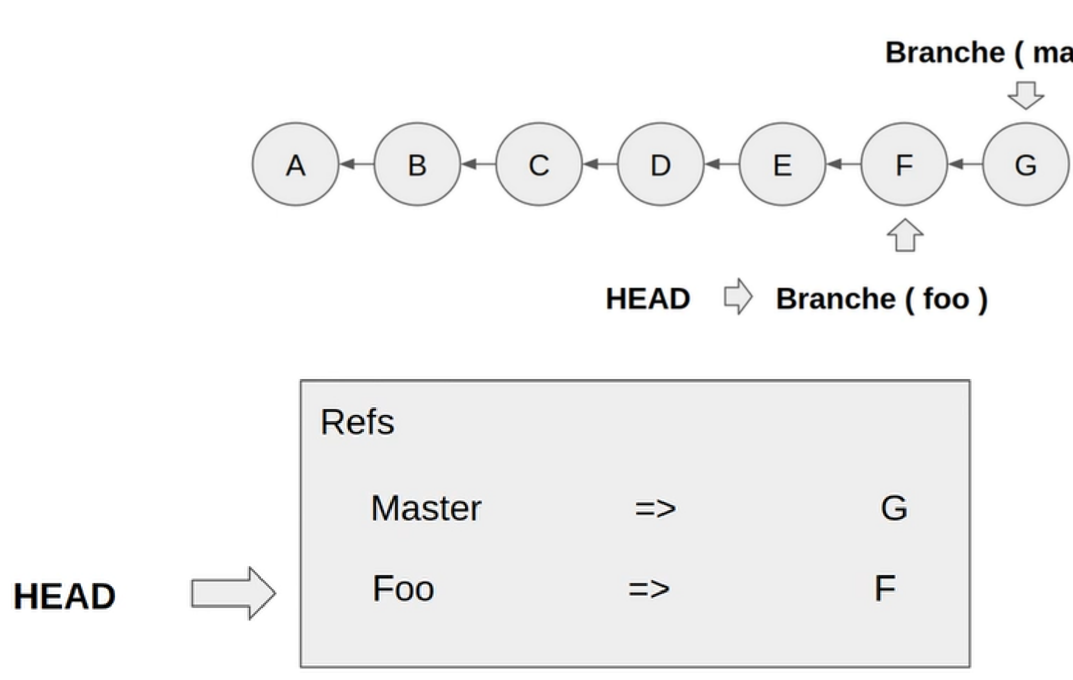
Regardons dans l'exemple ci-dessous un repos avec une branche **Master** et une branche **Foo**.
Donc initialement le HEAD de la branche Master est sur le commit G et celui de la branche Foo est sur le commit F.



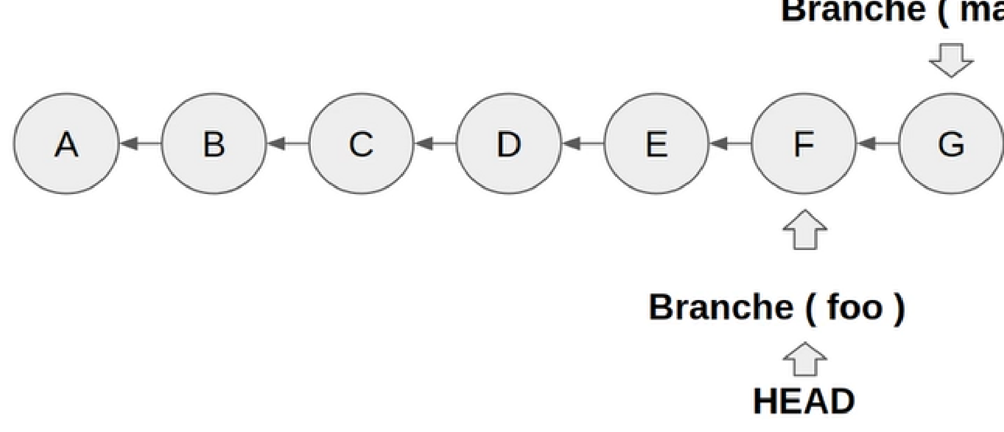
EL Hadji Gaye

git checkout branche

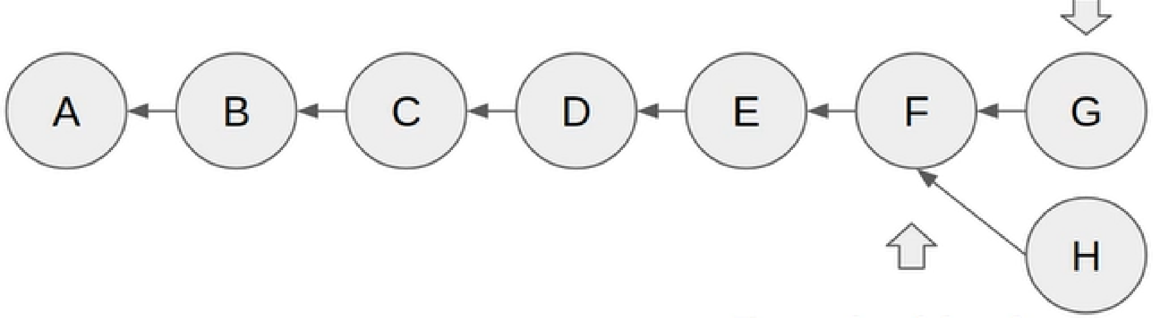
git checkout foo

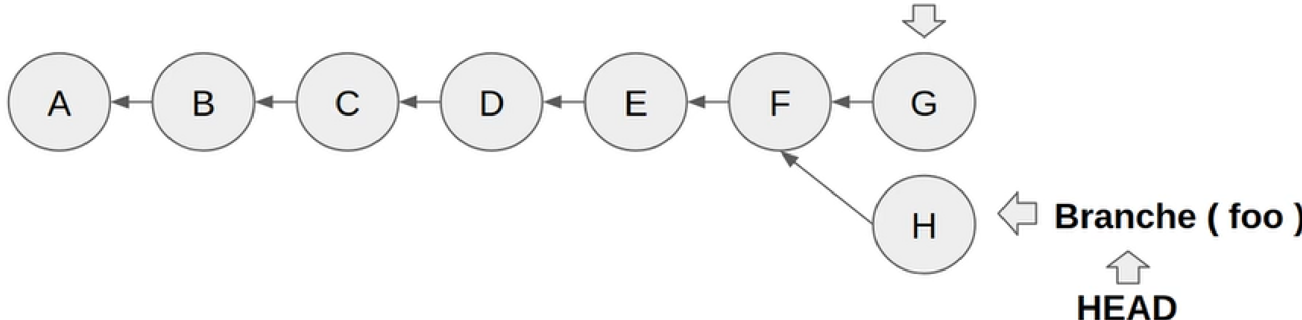


Git commit -m “new commit”



Après le commit sur la branche **Foo** le pointer **HEAD** se deplace vers le commit **H**.





Revenons maintenant à notre projet.

git status
git branch

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git status
On branch master
nothing to commit, working tree clean

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
  foo
* master
```

Réalisons une petite modification dans la branche master.

```
index.html ●
index.html > html > body > button
        You, 17 hours ago | 1 author (You)
 1  <!DOCTYPE html>
 2  <html>
 3      <head>
 4          <meta charset='utf-8'>
 5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
 6          <title>Seconde modification Page Title</title>
 7          <meta name='viewport' content='width=device-width, initial-scale=1'>
 8      </head>
 9      <body>
10          <button> button master</button>        You, 17 hours ago • Uncommitted
11      </body>
12  </html>
```

git add index.html
git commit -m "new commit for master"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "new commit for master"
[master 0a5802c] new commit for master
 1 file changed, 1 insertion(+), 1 deletion(-)
```

git log

```
commit 0a5802c6e7f5726128b3e8dc00d423885d714c37 (HEAD -> master)
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 11:04:14 2024 +0200

    new commit for master

commit 7927c8258ebd6d9954f1596e81a84faf3d253fa2 (foo)
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Mon Sep 23 06:42:53 2024 +0200

    Ceci est mon comimit via Notepad
```

On voit bien **master** pointe vers le nouveau commit et **foo** pointe vers l'ancien commit.

Passons maintenant à la branche foo.

git checkout foo
git branch

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout foo
Switched to branch 'foo'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git branch
* foo
  master
```

On voit bien que le contenu de **index.html** a changé :

```
index.html ×
index.html > html > head
You, 2 days ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>        You, 3 days ago • first commit
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Seconde modification Page Title</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10
11     </body>
12 </html>
```

Regardons maintenant au niveau des HEAD :

cd .git/
ls
cat HEAD
cd ..

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ cd .git

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ ls
COMMIT_EDITMSG  config  description  HEAD  hooks/  index  info/  logs/  objects/  ORIG_HEAD  refs/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ cat HEAD
ref: refs/heads/foo

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp/.git (GIT_DIR!)
$ cd ..

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
```

Réalisons une modification sur la branche **foo** :

```
index.html M
index.html > html > body > h2
You, 26 seconds ago | 1 author (You)
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset='utf-8'>
5           <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6           <title>Seconde modification Page Title</title>
7           <meta name='viewport' content='width=device-width, initial-scale=1'>
8       </head>
9       <body>
10          <h2> h2 foo</h2>        You, 22 seconds ago • Uncommitted changes
11      </body>
12  </html>
```

git add index.html
git commit -m "new commit on foo"
git log --oneline

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git commit -m "new commit on foo"
[foo d1f87af] new commit on foo
 1 file changed, 1 insertion(+), 1 deletion(-)

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git log --oneline
d1f87af (HEAD -> foo) new commit on foo
7927c82 Ceci est mon comimit via Notepad
40606b1 commit gitignore
c8d70e1 second commit
1104560 first commit
```

Regardons aussi l'historique de la branche master :

git checkout master
git log --oneline

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git checkout master
Switched to branch 'master'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log --oneline
0a5802c (HEAD -> master) new commit for master
7927c82 Ceci est mon comimit via Notepad
40606b1 commit gitignore
c8d70e1 second commit
1104560 first commit
```

Refaite un modification dans le fichier **index.html** puis refaite un checkout **foo**.

```
index.html M ×
index.html > html > head > meta
You, 5 seconds ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>        You, 3 da
6          <title>Seconde modification Page Title</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <button> button master</button>
11         <button>button</button>
12     </body>
13 </html>
```

git checkout foo

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout foo
error: Your local changes to the following files would be overwritten by checkout:
        index.html
Please commit your changes or stash them before you switch branches.
Aborting
```

On risque donc de perdre nos modifications.

git add index.html
git commit -m "commit une toute petite modification"
git checkout foo

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "commit une toute petite modification"
[master ff7ec04] commit une toute petite modification
 1 file changed, 1 insertion(+)

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout foo
Switched to branch 'foo'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
```

4. Fusionner des branches avec git merge

Cette commande va nous permettre de fusionner deux branches.

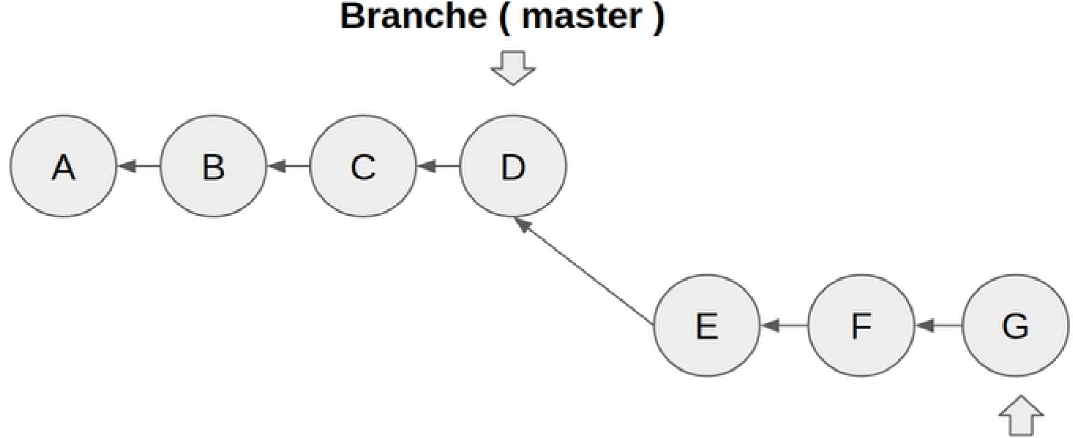
git merge maBranche

Il existe plusieurs types de merge :

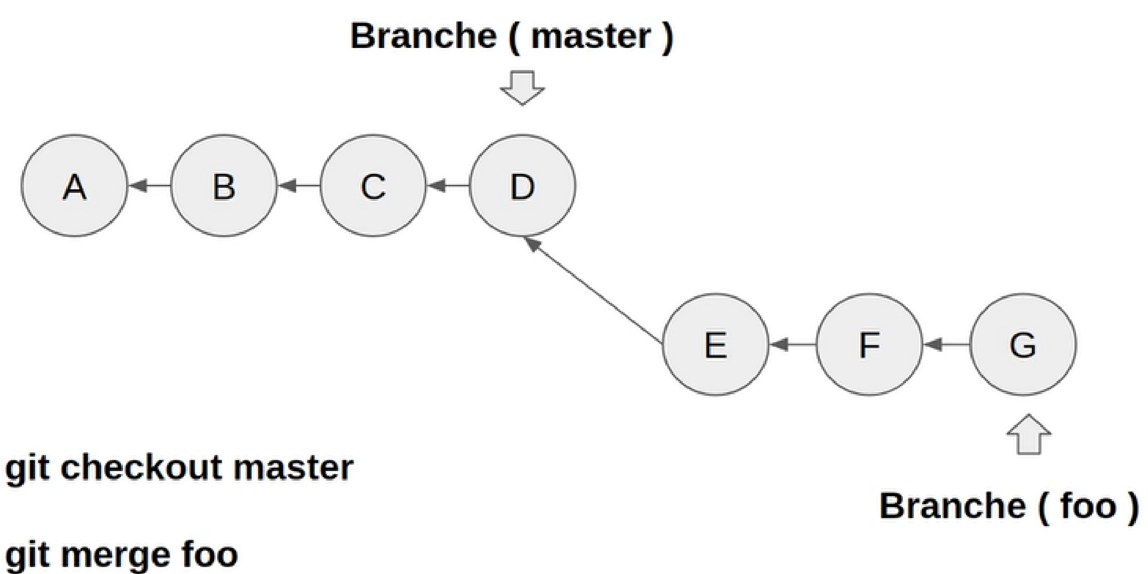
Fusion avance rapide (fast forward merge)

Dans ce premier scenario nous avons les branches **master** et **foo**. La branche foo a été crée à partir du commit D.

Merge fast forward

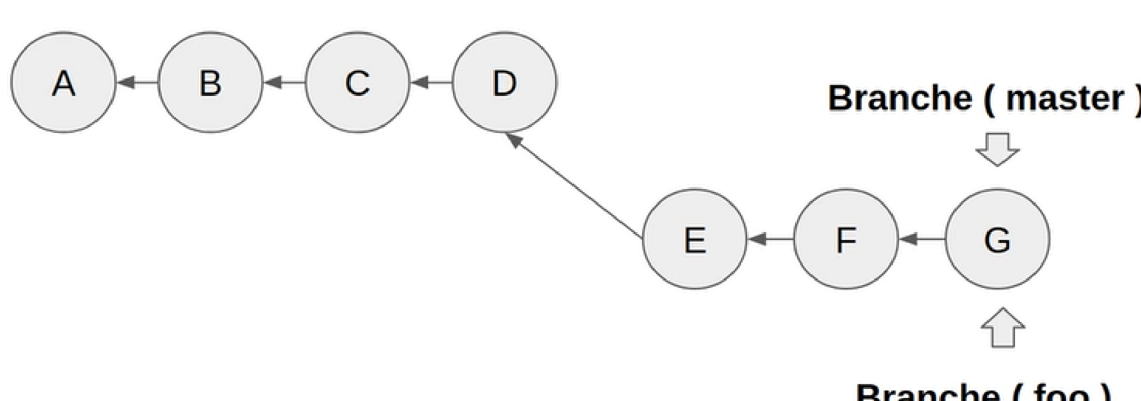


Merge fast forward

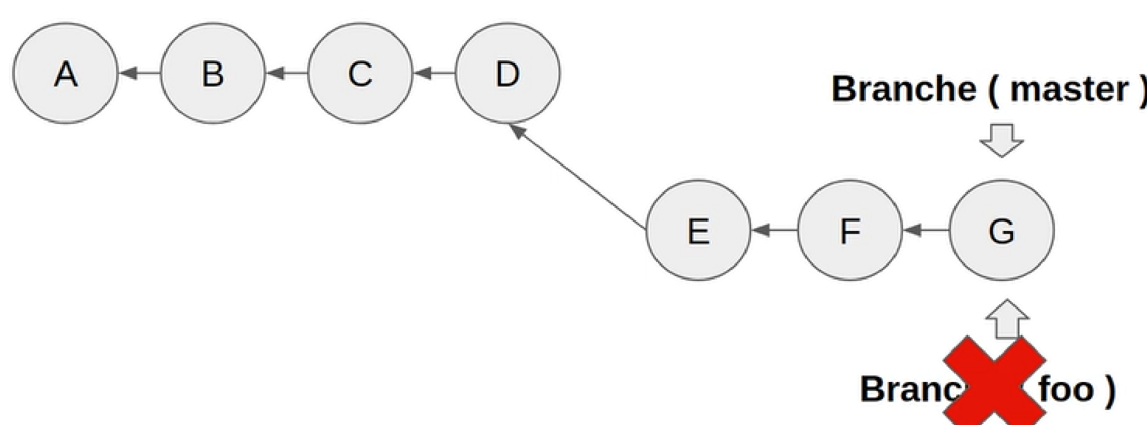


Le **HEAD** de master va se deplacer sur le commit **G**.

Merge fast forward



Merge fast forward



Apres la suppression de la branche foo on obtient :

Merge fast forward

Branche (master)

A ← B ← C ← D ← E ← F ← D

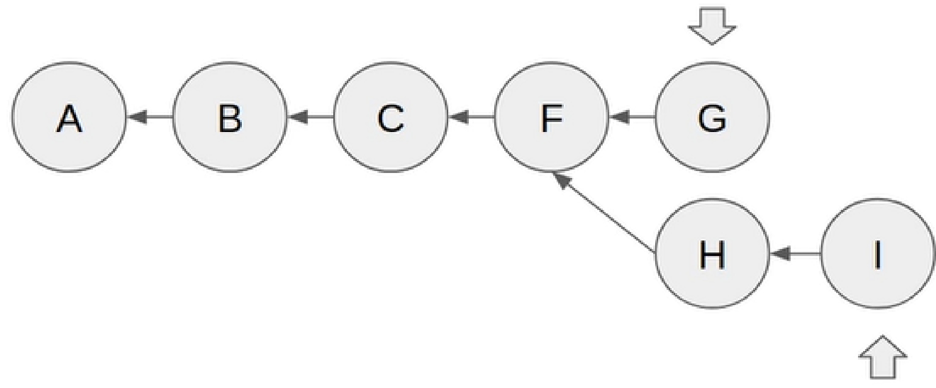
EL Hadji Gaye

Fusion à trois sources (three-way merge)

Dans ce deuxieme scénario nous avons crée la branche **foo** avec le commit F mais la branche **master** a continué à evoluer.

Merge three-way

Branche (master)



Branche (foo)

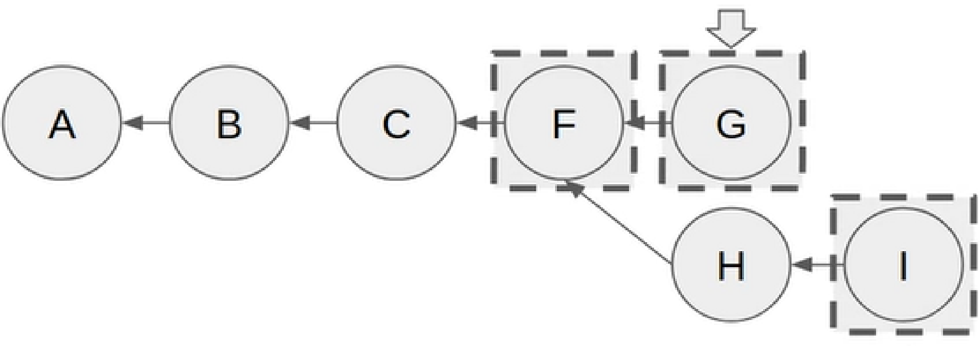
git checkout master

git merge foo

Pour realiser ce genre de commit Git va prendre en consideration le dernier commit sur **master**, le dernier commit **foo** et le dernier commit en commun de **foo** et **master** (le premier ancêtre en commun) et il fait la fusion des trois.

Merge three-way

Branche (master)



La fusion de ces trois commit va nous créer un nouveau commit **J** dit commit à deux parents.

Merge three-way

Commit de merge

2 parents

Branche (master)

A B C F G J

H I

Branche (foo)

Merge three-way

Branche (master)

A B C F G J

H I

Pratiquons maintenant !

Retounons à notre projet.

git branch
git checkout master
git branch -D foo

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git branch
* foo
  master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git checkout master
Switched to branch 'master'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch -D foo
Deleted branch foo (was d1f87af).

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
* master
```

Recréer et switcher sur la branche **foo**.

git checkout -b foo
git branch

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout -b foo
Switched to a new branch 'foo'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git branch
* foo
  master
```

```html
index.html M
index.html > html > head > title
You, 12 seconds ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Seconde modification Page Title2</title>        You, 8 seconds
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <button> button master</button>
11         <button>button</button>
12     </body>
13 </html>
```

git add index.html
git commit -m "first commit on foo"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git commit -m "first commit on foo"
[foo 7957464] first commit on foo
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Retournons à la branche master :

git checkout master
git merge foo

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git merge foo
Updating ff7ec04..7957464
Fast-forward
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On a bien un merge **Fast-forward**.

git log

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log
commit 7957464a10771a2613776b072898343e91cc7cad (HEAD -> master, foo)
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 12:59:34 2024 +0200

    first commit on foo

commit ff7ec04d3011fd02c887e33e2864d81d15b78308
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 12:04:34 2024 +0200

    commit une toute petite modification

commit 0a5802c6e7f5726128b3e8dc00d423885d714c37
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 11:04:14 2024 +0200

    new commit for master
```

git branch
git branch -D foo
git checkout -b foo

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
  foo
* master

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch -D foo
Deleted branch foo (was 7957464).

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout -b foo
Switched to a new branch 'foo'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
```

Realisons maintenant une modification dans la branche master.

git checkout master

```
index.html M ×
index.html > html > head > title
You, 55 seconds ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Seconde modification Page Title master not on foo</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <button> button master</button>
11         <button>button</button>
12     </body>
13 </html>
```

git add index.html
git commit -m "commit master not on foo"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "commit master not on foo"
[master 8a20467] commit master not on foo
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Retournons sur la branche foo puis réaliser une modification dessus. Par exemple enlever un bouton.

git checkout foo

```
index.html M ×
index.html > html > body > button
You, 34 minutes ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Seconde modification Page Title2</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <button> button master</button>        You, 2 hours ago • new commit f
11     </body>
12 </html>
```

git add index.html
git commit -m "commit update foo"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git commit -m "commit update foo"
[foo 76bbb13] commit update foo
 1 file changed, 1 deletion(-)
```

git checkout master
git merge foo

C:\Users\elhad\Desktop\AutoEntrepreneur\Formations\Git\courses\myapp\.git\MERGE_MSG - Notepad++

Fichier Édition Recherche Affichage Encodage Langage Paramètres Outils Macro Exécution Modules d'extension Doc

```
1  Merge branch 'foo'
2  # Please enter a commit message to explain why this merge is necessary,
3  # especially if it merges an updated upstream into a topic branch.
4  #
5  # Lines starting with '#' will be ignored, and an empty message aborts
6  # the commit.
```

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (foo)
$ git checkout master
Switched to branch 'master'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git merge foo
Auto-merging index.html
Merge made by the 'ort' strategy.
 index.html | 1 -
 1 file changed, 1 deletion(-)
```

On peut supprimer la branche foo :

git branch -D foo
git log

```
commit 31cfa2fd3e1c0451885763e5b7d2082a8b61c064 (HEAD -> master)
Merge: 8a20467 76bbb13
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 13:40:23 2024 +0200

    Merge branch 'foo'

commit 76bbb13e30598b63ccf09543b75a95e5f07bff3b
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 13:37:08 2024 +0200

    commit update foo

commit 8a20467bd9a11c2f9b60673421907a81507c7313
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 13:29:19 2024 +0200

    commit master not on foo

commit 7957464a10771a2613776b072898343e91cc7cad
Author: ElHadji <elhadji.gaye83@gmail.com>
Date:   Wed Sep 25 12:59:34 2024 +0200
```

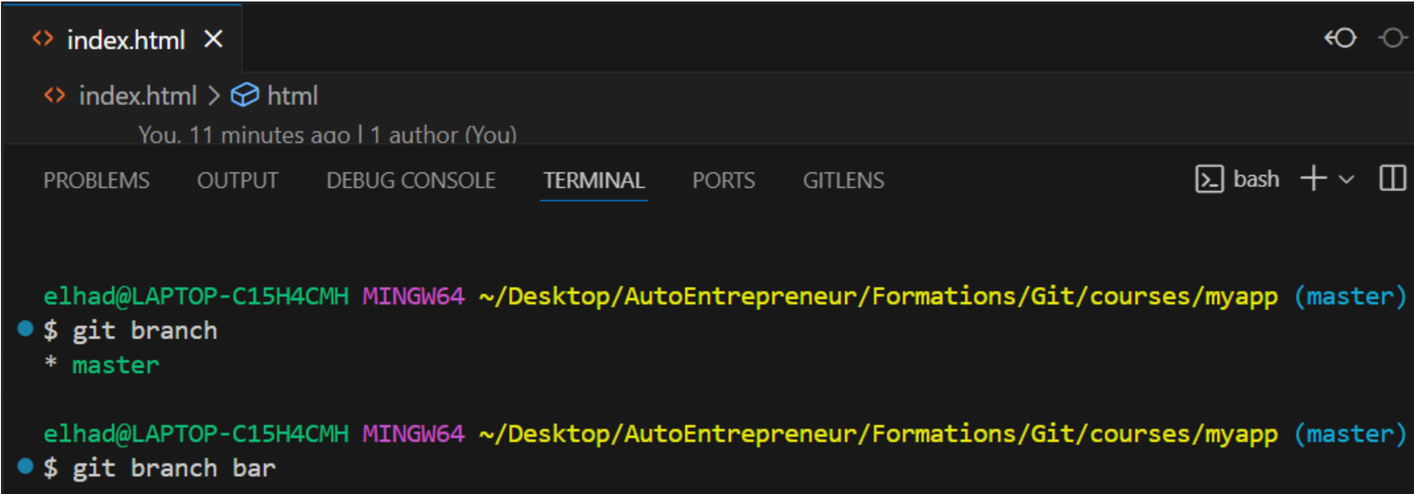
5. *Conflit entre branches Git*

On peut malheureusement se retrouver dans une situation où le merge entre deux branches provoque un conflit. Il faut alors résoudre les conflits.

Revenons sur notre projet !

Lancer les commandes ci-dessous :

git branch
git checkout -b bar



Modifier par exemple le titre de la page **index.html**.

index.html M

index.html > html > head > title

You, 7 seconds ago | 1 author (You)

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset='utf-8'>
        <meta http-equiv='X-UA-Compatible' content='IE=edge'>
        <title>Page Title master</title>        You, 3 seconds ago • Uncommitted
        <meta name='viewport' content='width=device-width, initial-scale=1'>
    </head>
    <body>
        <button> button master</button>
    </body>
</html>
```

git add index.html
git commit -m "commit only on master"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "commit only on master"
[master a409927] commit only on master
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Nous allons retourner dans la branche **bar** et faire une modification similaire par exemple juste changer le titre.

git checkout bar

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git checkout bar
Switched to branch 'bar'
```

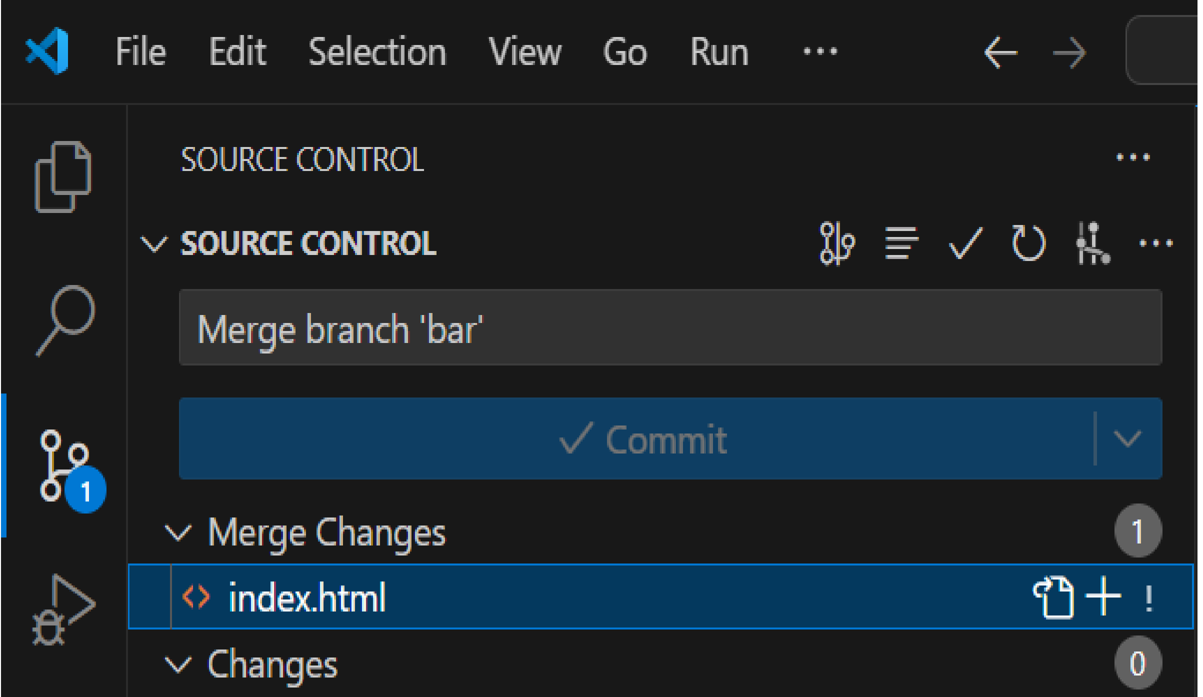
```
index.html M ×
index.html > html
You, 6 seconds ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Page Title bar</title>
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <button> button master</button>
11     </body>
12 </html>       You, 3 days ago • first commit
```

git add index.html
git commit -m "commit only on bar"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (bar)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (bar)
$ git commit -m "commit only on bar"
[bar 113e01c] commit only on bar
 1 file changed, 1 insertion(+), 1 deletion(-)
```

git checkout master
git merge bar

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (bar)
$ git checkout master
Switched to branch 'master'

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git merge bar
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master|MERGING)
```




```
index.html !  ×
index.html > html
You, 2 minutes ago | 1 author (You)
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset='utf-8'>
5           <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
6   <<<<<<< HEAD (Current Change)
7           <title>Page Title master</title>
8   =======
9           <title>Page Title bar</title>
10  >>>>>>> bar (Incoming Change)
11          <meta name='viewport' content='width=device-width, initial   Resolve in Merge Editor
12      </head>
13      <body>
```

Nous avons choisit de garder la version de la branche **master** (**Accept Current Change**).

git status

on peut à tout moment annuler le merge en faisant : **git merge --abort**.

git add index.html
git commit -m "commit merge ok"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master|MERGING)
$ git status

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master|MERGING)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master|MERGING)
$ git commit -m "commit merge ok"
[master 6a2e4a3] commit merge ok
```

git log –oneline

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git log --oneline
6a2e4a3 (HEAD -> master) commit merge ok
113e01c (bar) commit only on bar
a409927 commit only on master
31cfa2f Merge branch 'foo'
76bbb13 commit update foo
8a20467 commit master not on foo
7957464 first commit on foo
ff7ec04 commit une toute petite modification
0a5802c new commit for master
7927c82 Ceci est mon comimit via Notepad
40606b1 commit gitignore
```

6. Rectifier un commit

Parfois vous souhaiterez modifier un commit que vous venez de faire : soit pour ajouter des fichiers que vous avez oublié, soit pour modifier le message de validation.

Dans ces deux cas il faudra utiliser :

git commit --amend

Attention ! Il ne faut pas utiliser cette commande si vous avez déjà push votre commit.

Cette commande créera un nouvel objet commit avec un nouvel arbre et un nouveau message de validation.
Il aura donc un nouveau hash pour l'identifier, et c'est pour cette raison qu'il ne faut surtout pas modifier le commit de cette manière si vous l'avez déjà publié sur le répertoire distant.

Retounons maintenant à notre projet :

git branch

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git branch
* master
```

Réalisons une petite modification dans le title du fichier **index.html**.

```
index.html M
index.html > html > head > title
You, 1 second ago | 1 author (You)
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset='utf-8'>
5          <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6          <title>Page Title master bis</title>        You, 1 second ago • Uncomm
7          <meta name='viewport' content='width=device-width, initial-scale=1'>
8      </head>
9      <body>
10         <button> button master</button>
11     </body>
12 </html>
```

git add index.html
git commit -m "new commit"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git add index.html

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit -m "new commit"
[master eab8d8f] new commit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Faire un git log

git log --oneline

```
eab8d8f (HEAD -> master) new commit
6a2e4a3 commit merge ok
113e01c commit only on bar
a409927 commit only on master
31cfa2f Merge branch 'foo'
76bbb13 commit update foo
8a20467 commit master not on foo
7957464 first commit on foo
ff7ec04 commit une toute petite modification
0a5802c new commit for master
7927c82 Ceci est mon comimit via Notepad
40606b1 commit gitignore
c8d70e1 second commit
```

Si le Lead Dev vous dit que que le titre de commit **"new commit"** est abérant et qu'il faut le modifier vous pouvez faire :

git commit --amend -m "modify title"

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/myapp (master)
$ git commit --amend -m "modify title"
[master e52ce77] modify title
 Date: Wed Sep 25 15:25:42 2024 +0200
 1 file changed, 1 insertion(+), 1 deletion(-)
```

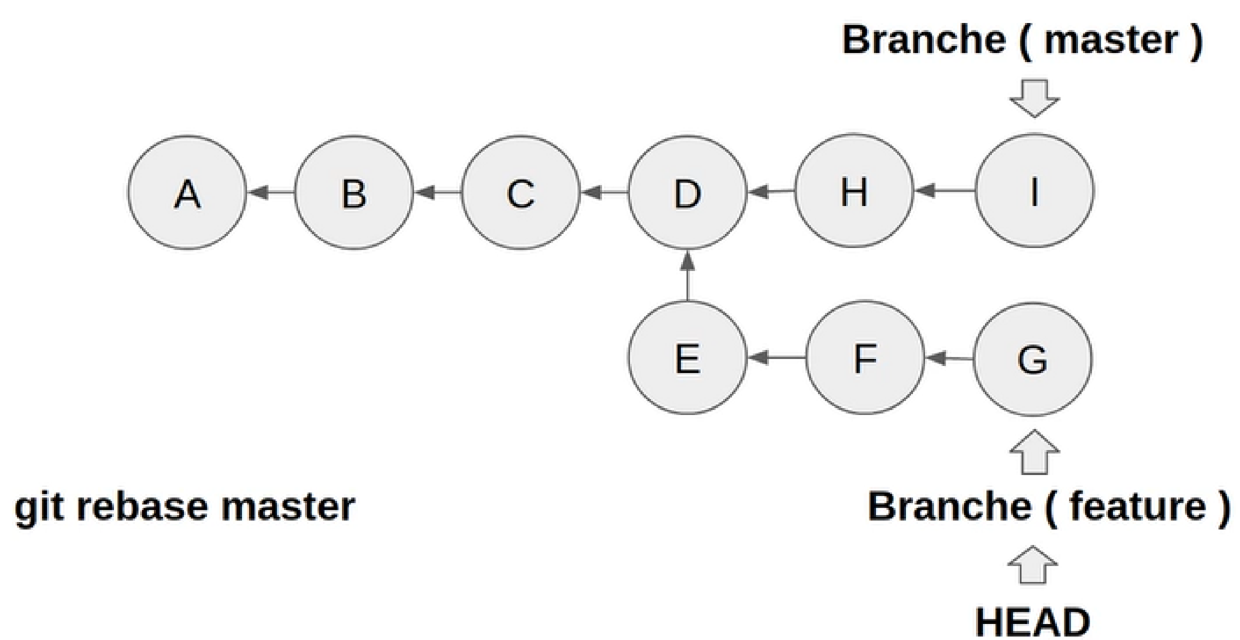
git log --oneline

```
e52ce77 (HEAD -> master) modify title
6a2e4a3 commit merge ok
113e01c commit only on bar
a409927 commit only on master
31cfa2f Merge branch 'foo'
76bbb13 commit update foo
8a20467 commit master not on foo
7957464 first commit on foo
ff7ec04 commit une toute petite modification
0a5802c new commit for master
7927c82 Ceci est mon comimit via Notepad
40606b1 commit gitignore
c8d70e1 second commit
```

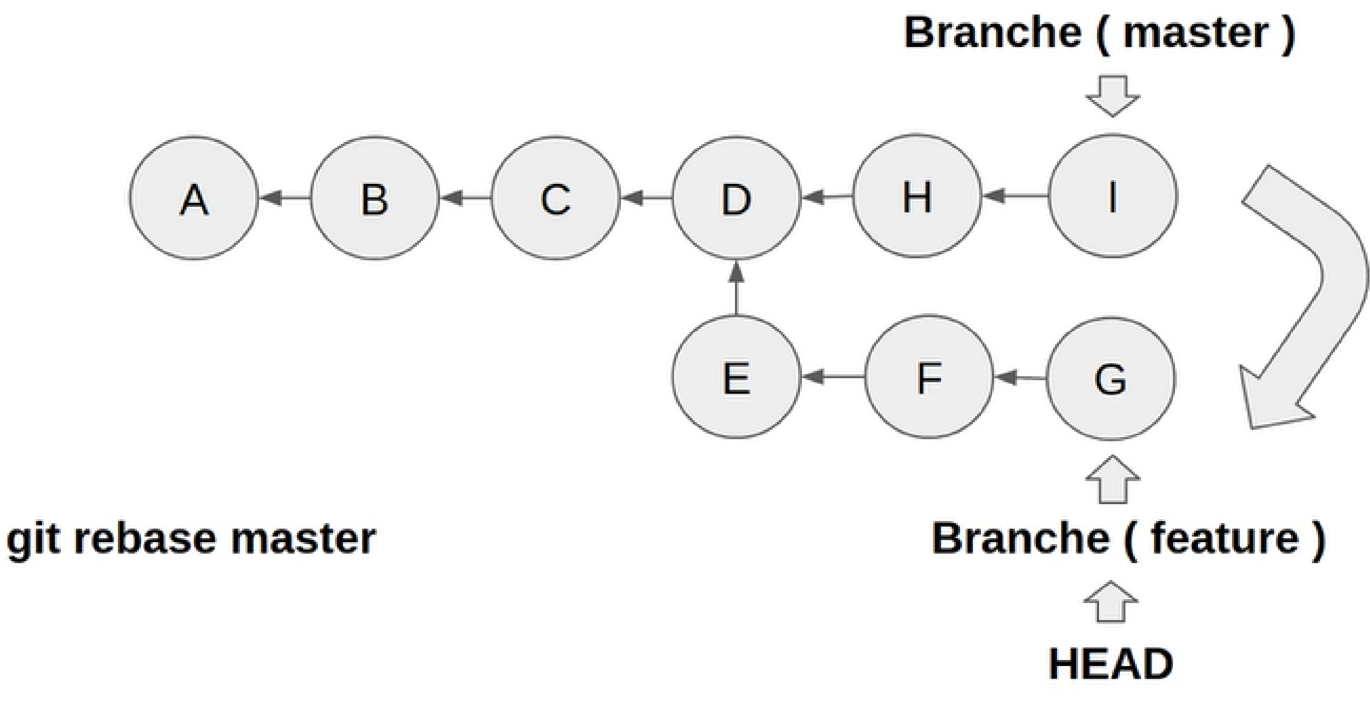
7. La commande git rebase

Pour le git rebase nous allons partir de la situation ci-dessous :

git rebase branch



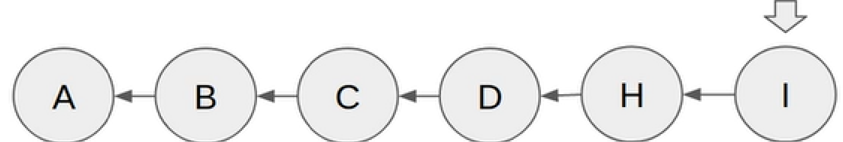
La branche **feature** a été crée à partir du commit **D**.
Supposons qu'on se retrouve dans une situation où nous avons besoin de recuperer les commits **H** et **I** du master.

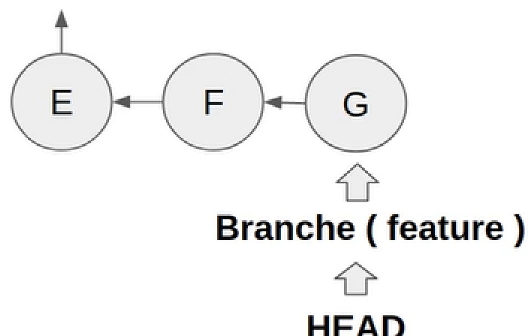


En étant dans la branche feature on peut lancer la commande :

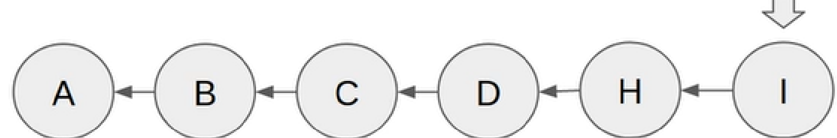
git rebase master

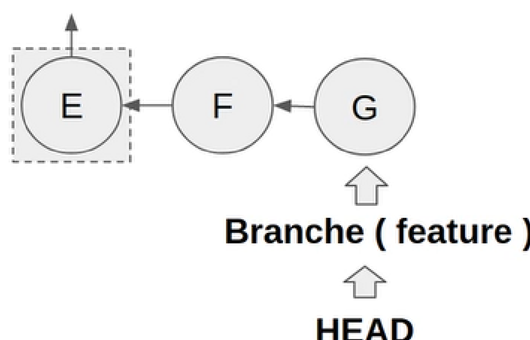
Branche (master)

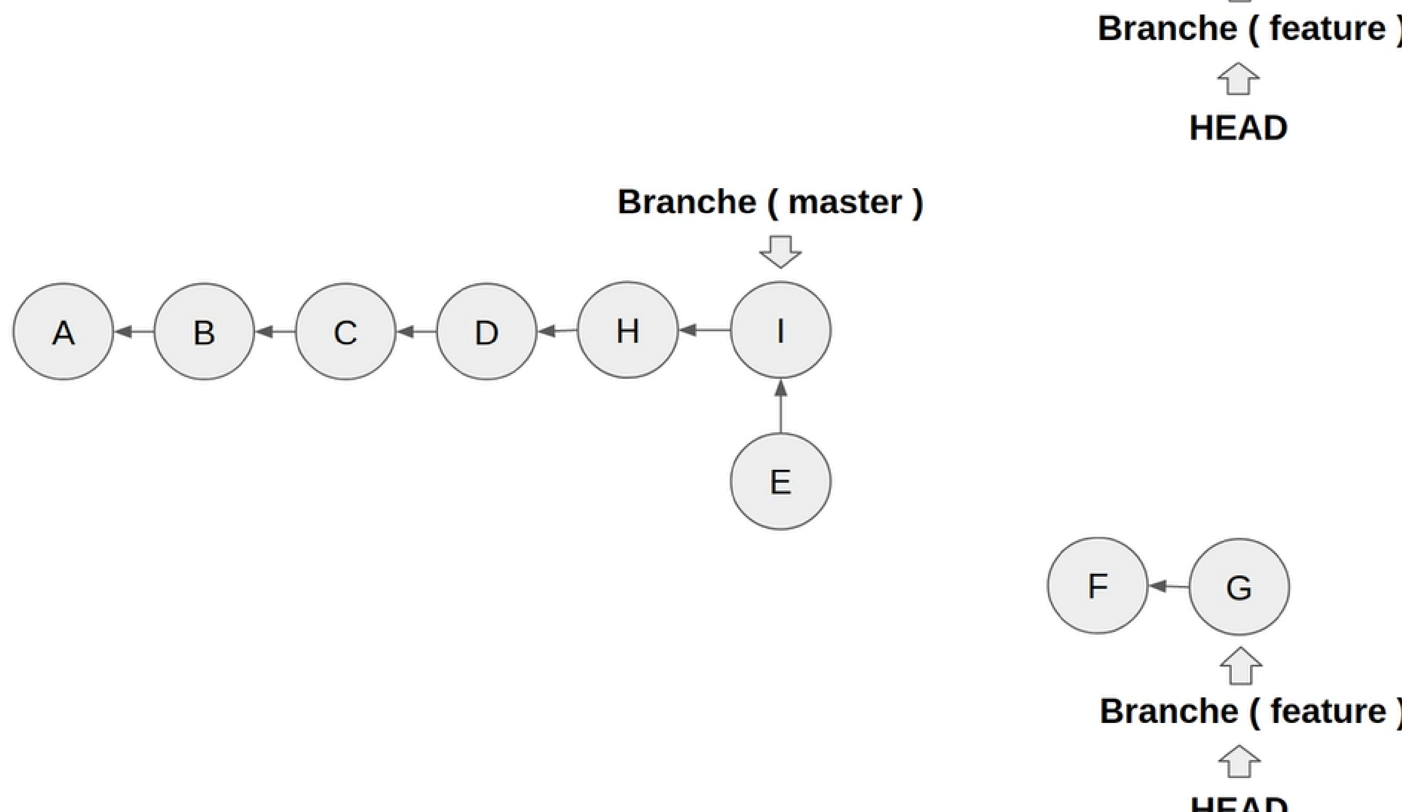




Branche (master)






Branche (master)
A
B
C
D
H
I
E
F
G
Branche (feature)
HEAD
Branche (master)
A
B
C
D
H
I
E
F
G
Branche (feature)
HEAD

Branche (master)

A B C D H I

E

F G

Branche (feature)

HEAD

Branche (master)

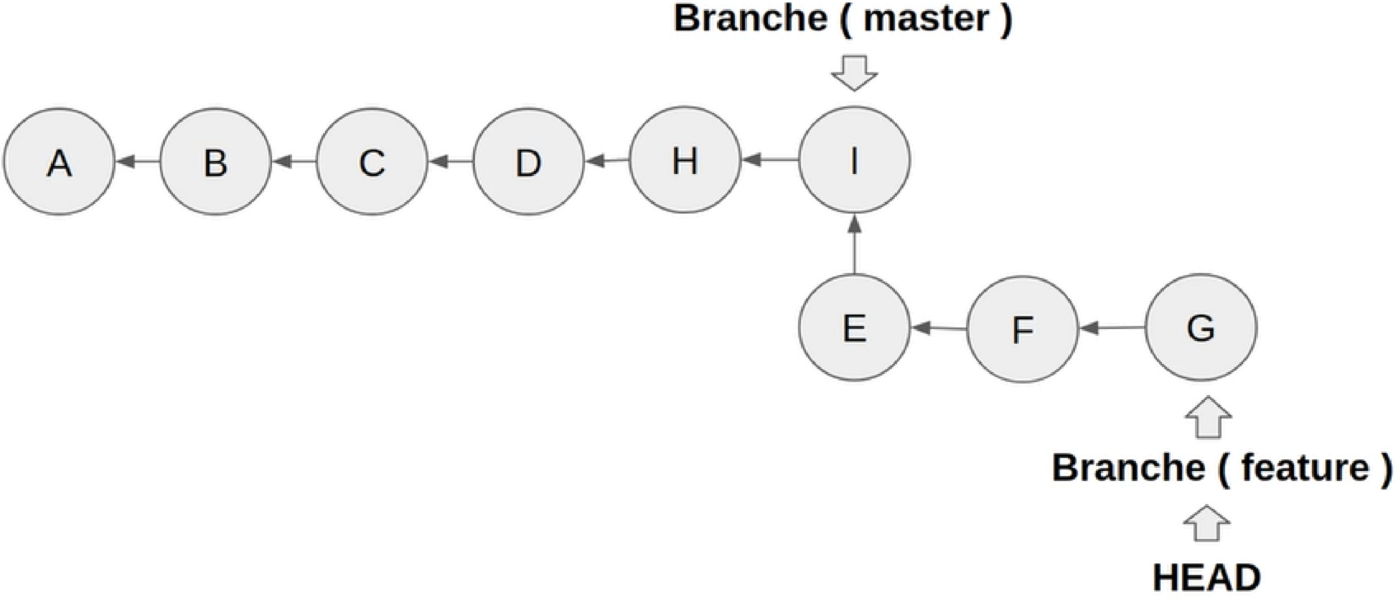
A B C D H I

E F

G

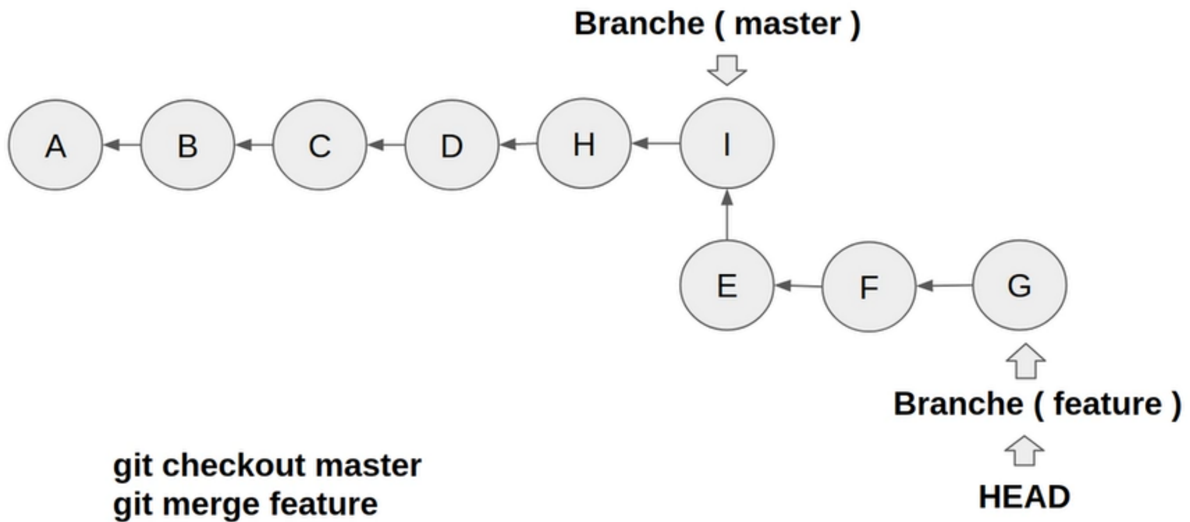
Branche (feature)

HEAD



En cas de merge ce sera très facile ce sera même un merge fast-Forward.

git rebase branch



HEAD

Branche (master)

A B C D H I

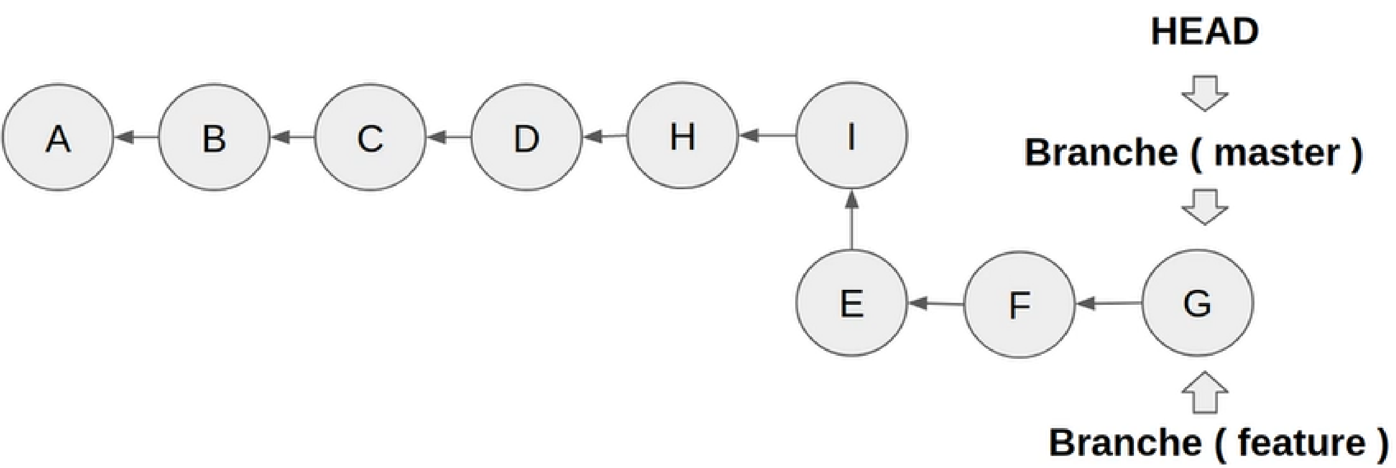
E F G

Branche (feature)

git checkout master
git merge feature

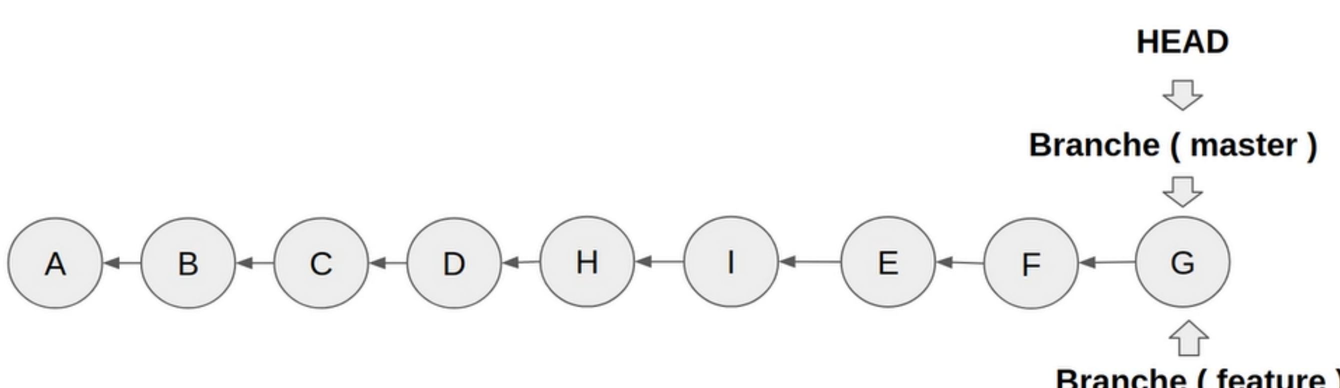
Le **HEAD** du master sera juste deplacer de **I** à **G**.

fast-forward

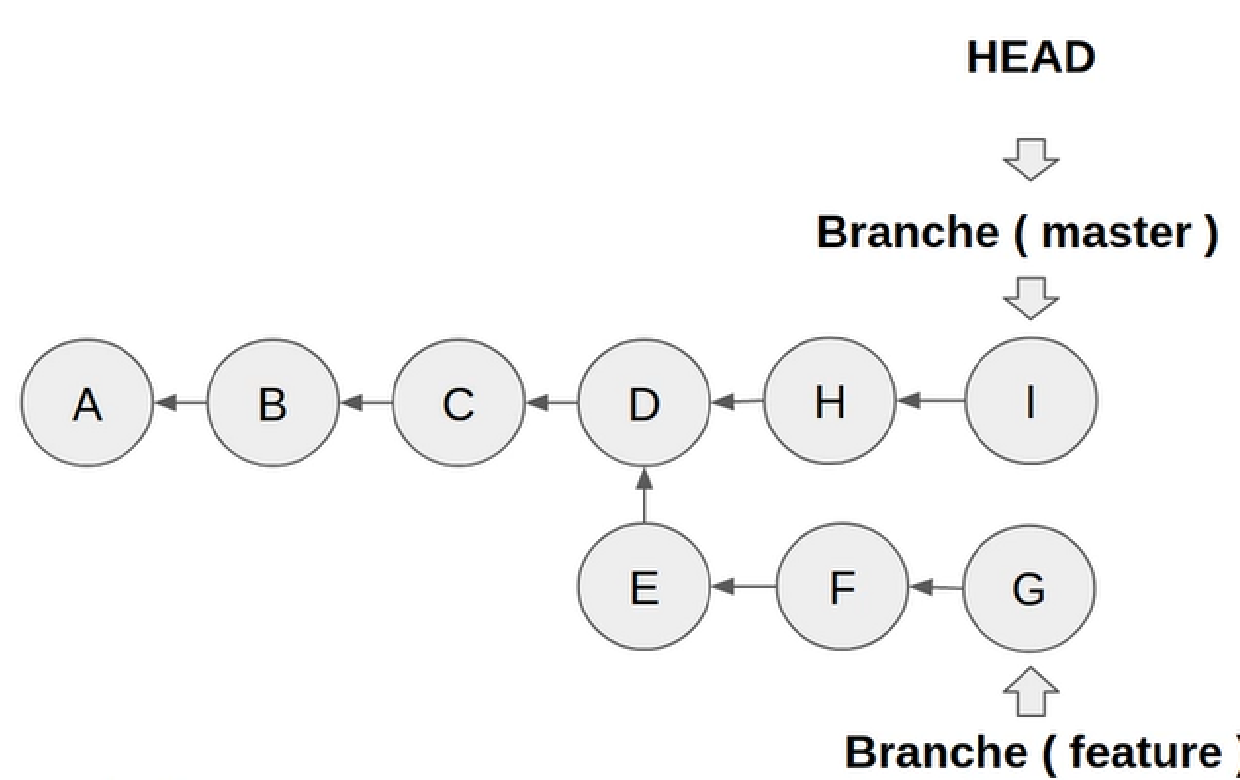


git checkout master
git merge feature

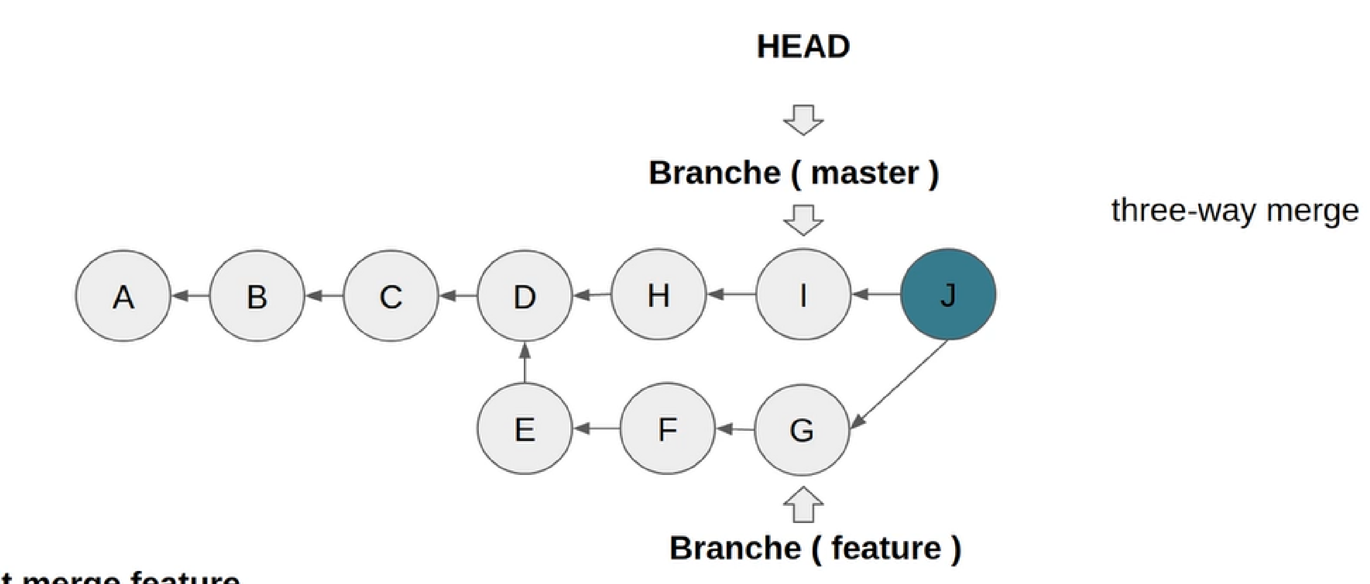
Au final l'historique de commit ressemblera à :



Dans le cas où on veut merger sans faire de rebase :



Comme nous l'avons vu précédamment il va y avoir un **three-way merge**.

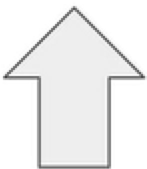


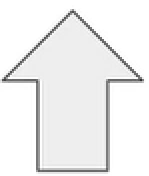
git rebase + git merge VS git merge

git rebase
git merge

vs

git merge





Histoire romancée

Vrai histoire

Historique simplifié et claire

IV) Les répertoires distants avec Gitlab

1. Introduction

Un répertoire ou dépôt distant Git permet de rendre accessible en permanence, pour toute votre équipe, l'historique d'un projet.
Ils permettent ce qu'on appelle le développement distribué.

Vous pouvez créer votre propre serveur Git, mais nous ne le verrons pas dans ce cours pour trois raisons :

Premièrement, c'est très long et complexe, et personne ne le fait. C'est d'ailleurs pour cette raison que Github, Gitlab et Bitbucket valent si cher. Créer et maintenir un serveur distant offrant les fonctionnalités nécessaires à un projet moderne est un métier en soi.

Deuxièmement, Gitlab offre un nombre hallucinant de fonctionnalités gratuitement et en open source.

Troisièmement, si vous tenez absolument à héberger vos source que sur vos serveurs, vous pouvez installer Gitlab Community Edition.

2. *Pourquoi choisir de Gitlab*

Nous choisissons de vous montrer Gitlab plutôt que Github pour deux raisons.

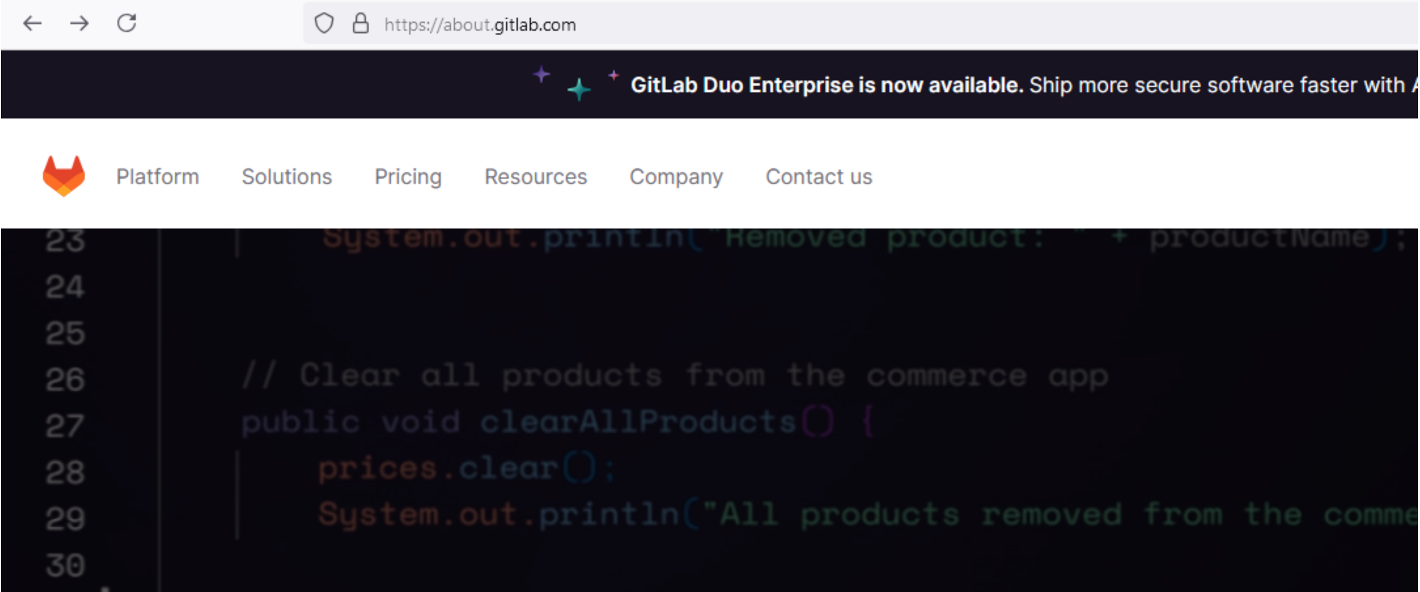
Premièrement, Github a été racheté par Microsoft en 2018.

Deuxièmement, Gitlab a plus de fonctionnalités offertes gratuitement, alors qu'il faut rapidement s'abonner avec Github pour avoir accès aux fonctionnalités intéressantes.

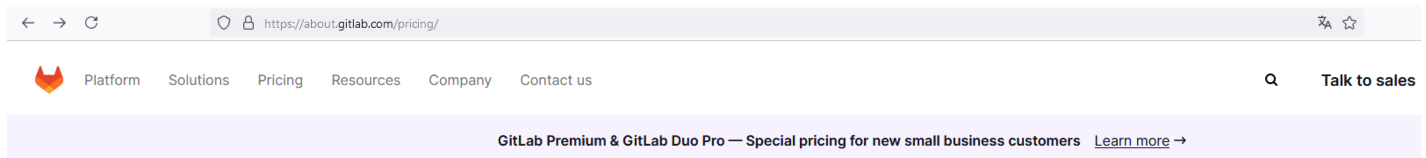
Gitlab ne risque pas de disparaître, l'entreprise a déjà levé plus de 440 millions de dollars et est en forte croissance. Elle est utilisée par énorméments d'entreprises et d'institutions, comme par exemple la NASA ou le CERN.

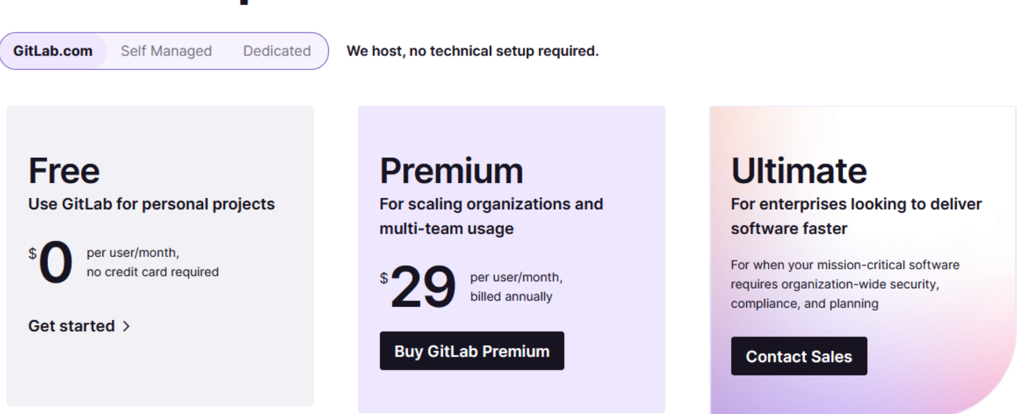
3. *Création d'un compte Gitlab*

Se rendre sur le site https://about.gitlab.com/



Cliquer sur Pricing ce qui nous renvois sur https://about.gitlab.com/pricing/





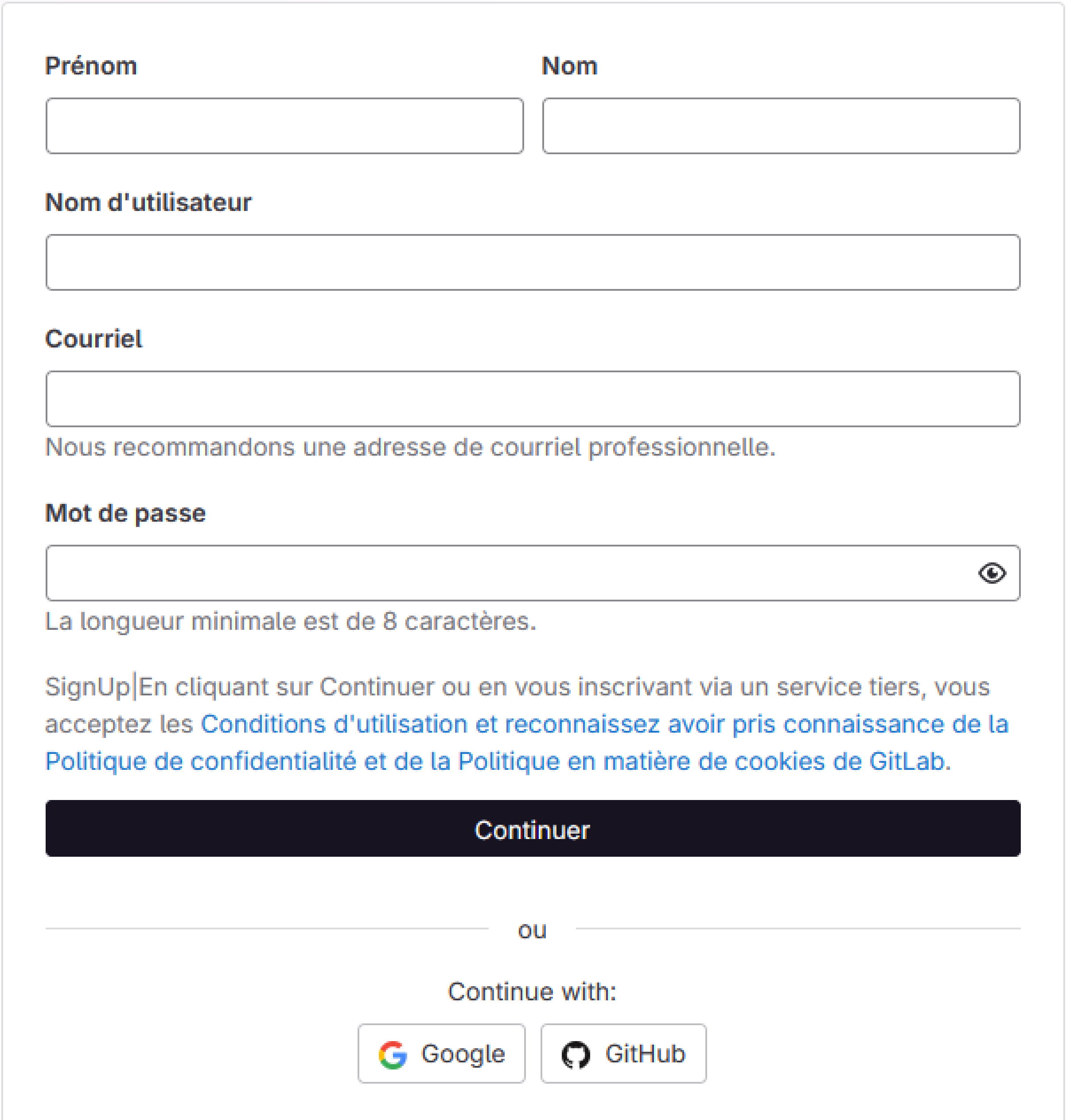
GitLab.com Self Managed Dedicated

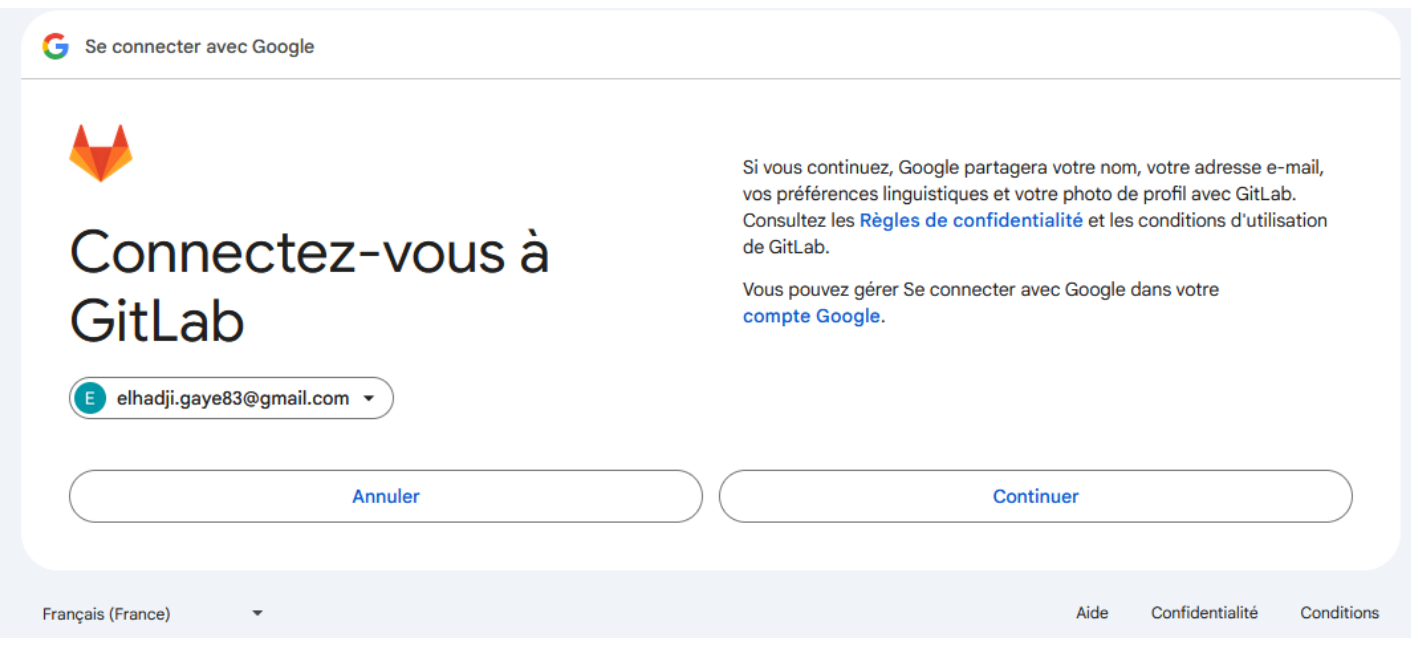
Free

Use GitLab for personal projects

$0 per user/month,
no credit card required

Get started >

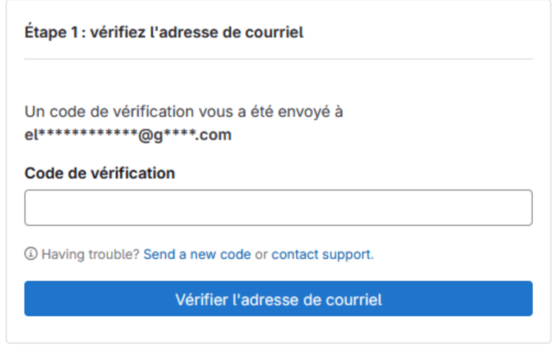

Essai gratuit de 30 jours de
GitLab Ultimate
Prénom
Nom
Nom d'utilisateur
Courriel
Nous recommandons une adresse de courriel professionnelle.
Mot de passe
La longueur minimale est de 8 caractères.
SignUp|En cliquant sur Continuer ou en vous inscrivant via un service tiers, vous acceptez les Conditions d'utilisation et reconnaissez avoir pris connaissance de la Politique de confidentialité et de la Politique en matière de cookies de GitLab.
Continuer
ou
Continue with:
Google
GitHub



ⓘ Vous devez confirmer votre adresse e-mail dans les 3 jours suivant votre inscription. Si vous ne confirmez pas votre adresse e-mail dans ce délai, votre compte sera supprimé et vous devrez vous réinscrire à GitLab.

Aidez-nous à assurer la sécurité de GitLab

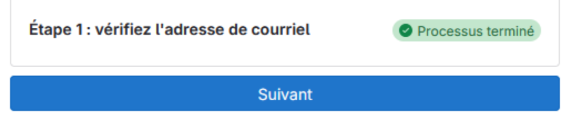
You are signed in as elhadji.gaye83. For added security, you'll need to verify your identity in a few quick steps.

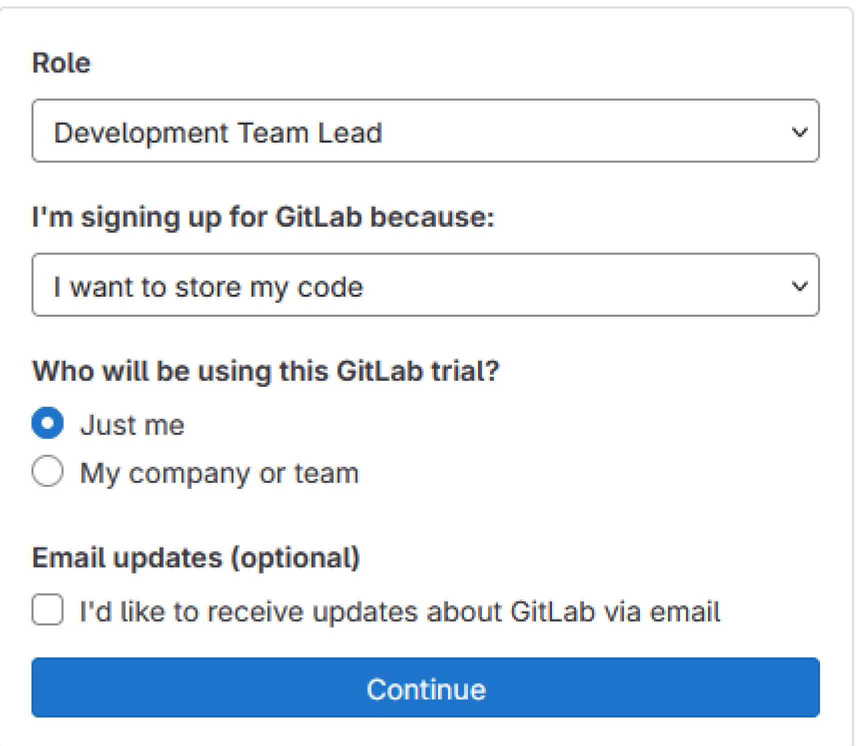


ⓘ Vous devez confirmer votre adresse e-mail dans les 3 jours suivant votre inscription. Si vous ne confirmez pas votre adresse e-mail dans ce délai, votre compte sera supprimé et vous devrez vous réinscrire à GitLab.

Aidez-nous à assurer la sécurité de GitLab

You are signed in as elhadji.gaye83. For added security, you'll need to verify your identity in a few quick steps.




Welcome to GitLab,
El!
To personalize your GitLab experience, we'd like to know a bit more about you. We won't share this information with anyone.
Role
Development Team Lead
I'm signing up for GitLab because:
I want to store my code
Who will be using this GitLab trial?
Just me
My company or team
Email updates (optional)
I'd like to receive updates about GitLab via email
Continue

About your company

To activate your trial, we need additional details from you.

First name

El Hadji

Last name

Gaye

Company name

El Hadji Gaye

Number of employees

1 - 99

Country or region

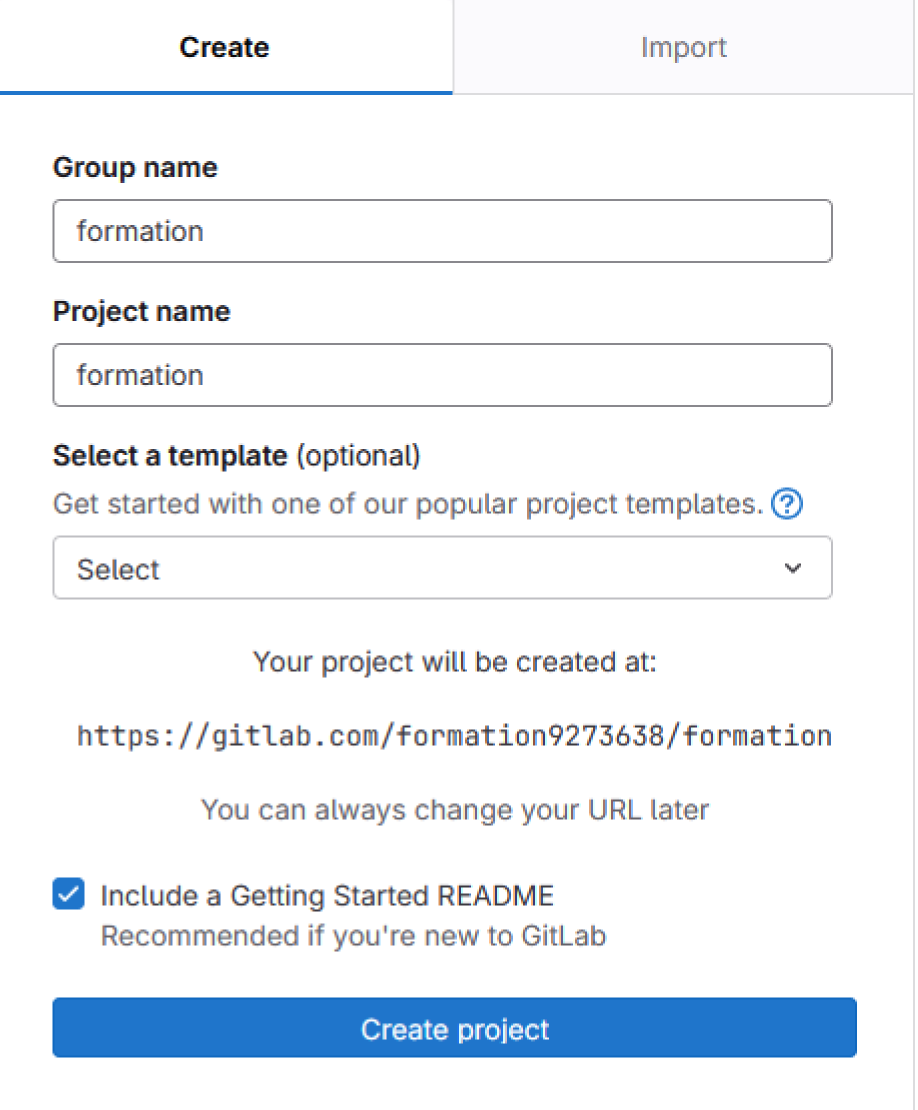
France

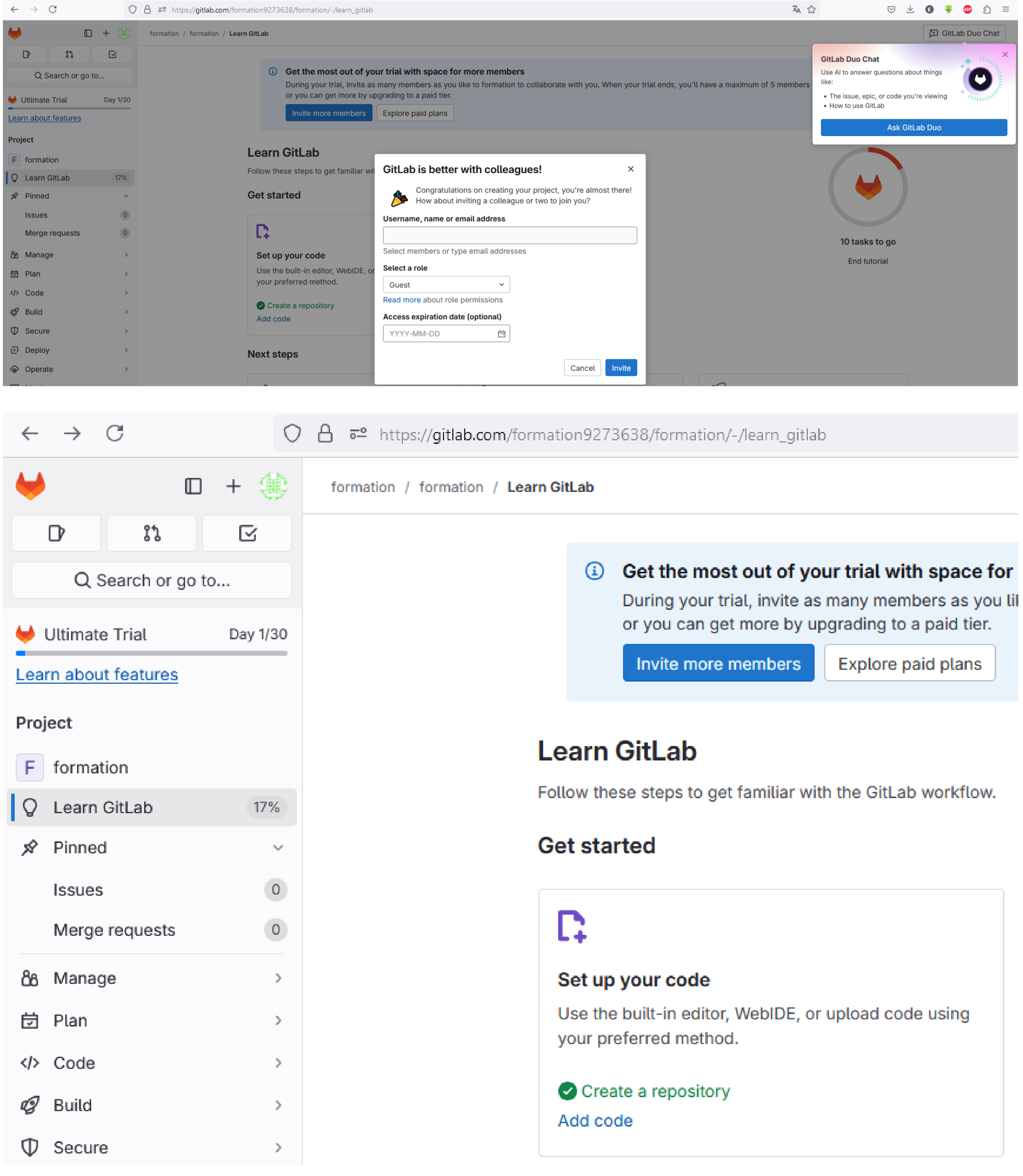
Telephone number (optional)

Allowed characters: +, 0-9, -, and spaces.

Website (optional)

Continue

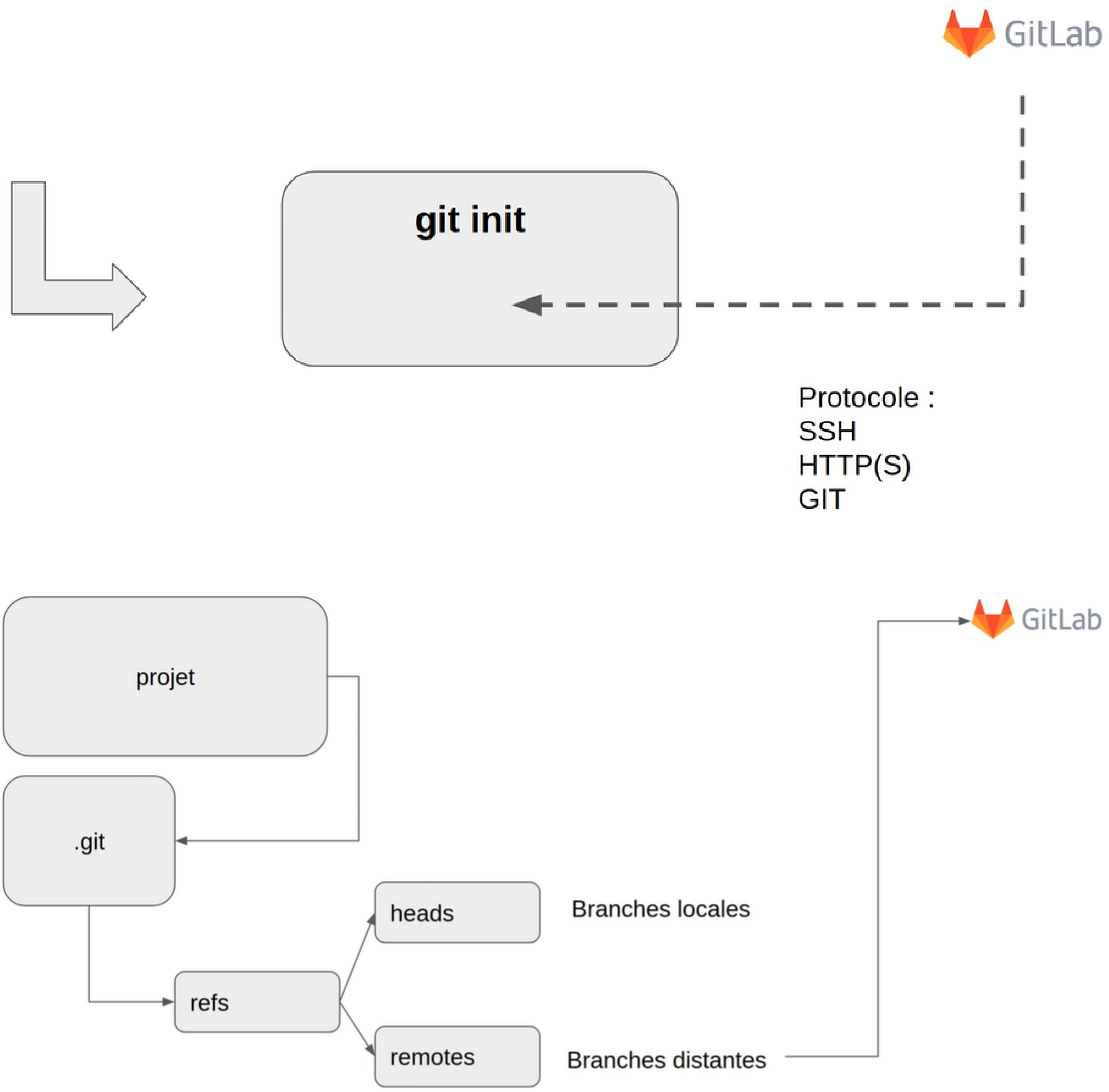

Create or import your first project
Projects help you organize your work. They contain your file repository, issues, merge requests, and so much more.
Create
Import
Group name
formation
Project name
formation
Select a template (optional)
Get started with one of our popular project templates.
Select
Your project will be created at:
https://gitlab.com/formation9273638/formation
You can always change your URL later
Include a Getting Started README
Recommended if you're new to GitLab
Create project

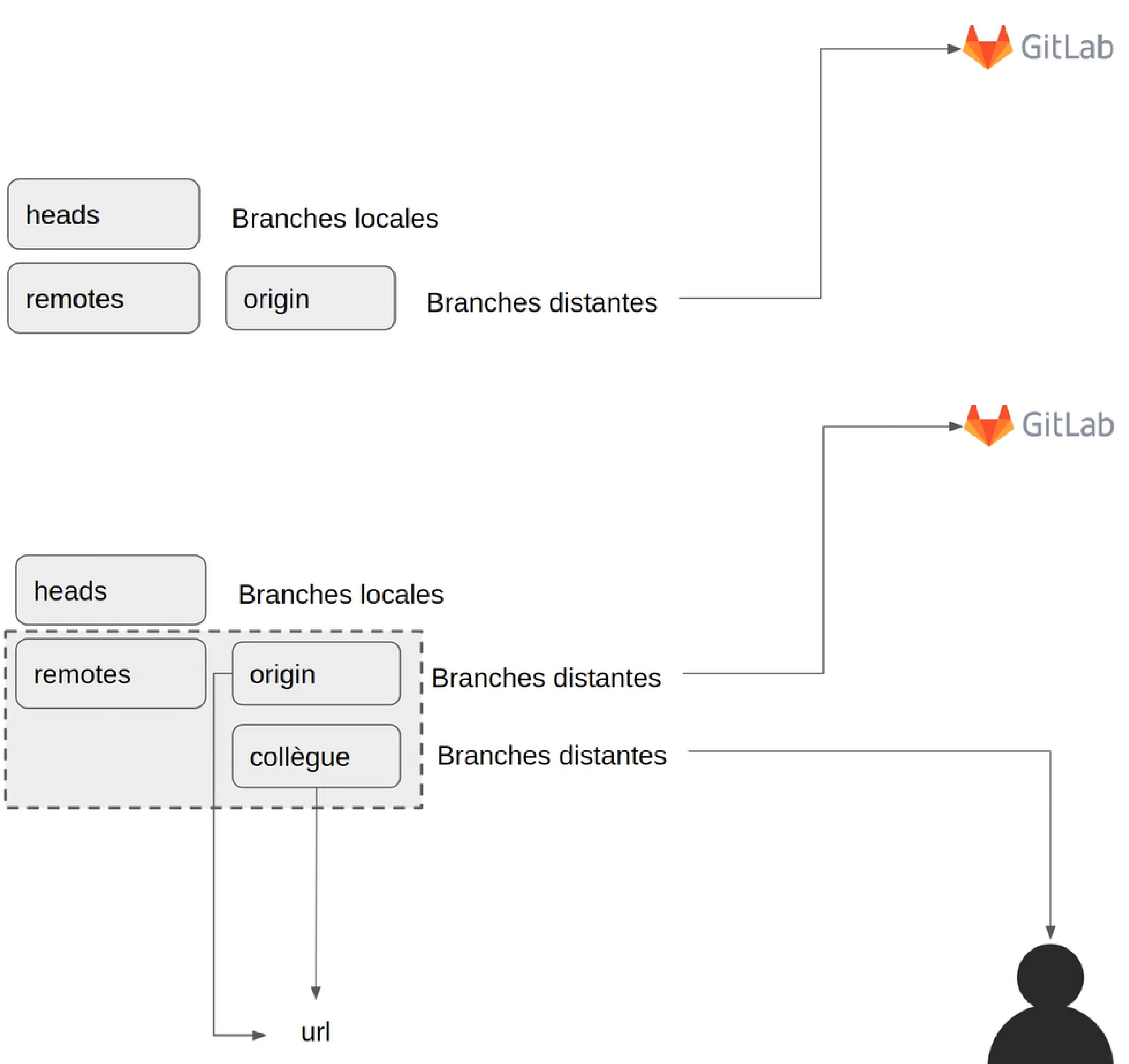


https://gitlab.com/formation9273638/formation/-/learn_gitlab
formation / formation / Learn GitLab
GitLab Duo Chat
Use AI to answer questions about things like:
The issue, epic, or code you're viewing
How to use GitLab
Ask GitLab Duo
GitLab is better with colleagues!
Congratulations on creating your project, you're almost there! How about inviting a colleague or two to join you?
Username, name or email address
Select members or type email addresses
Select a role
Guest
Read more about role permissions
Access expiration date (optional)
YYYY-MM-DD
Cancel
Invite
10 tasks to go
End tutorial
Search or go to...
Ultimate Trial
Day 1/30
Learn about features
Project
formation
Learn GitLab
17%
Pinned
Issues
Merge requests
Manage
Plan
Code
Build
Secure
Deploy
Operate
Get the most out of your trial with space for more members
During your trial, invite as many members as you like to formation to collaborate with you. When your trial ends, you'll have a maximum of 5 members
or you can get more by upgrading to a paid tier.
Invite more members
Explore paid plans
Learn GitLab
Follow these steps to get familiar with the GitLab workflow.
Get started
Set up your code
Use the built-in editor, WebIDE, or upload code using your preferred method.
Create a repository
Add code
Next steps

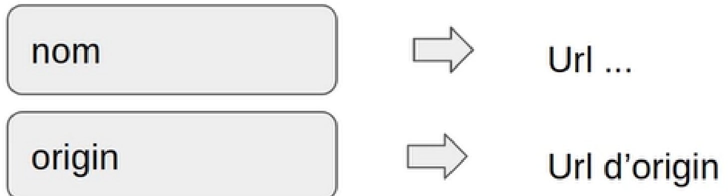
4. Cloner un repertoire distant

Cloner un repertoire distant se fait avec la syntaxe ci-dessous :




GitLab
heads
Branches locales
remotes
origin
Branches distantes
GitLab
heads
Branches locales
remotes
origin
Branches distantes
collègue
Branches distantes
url

git remote -v

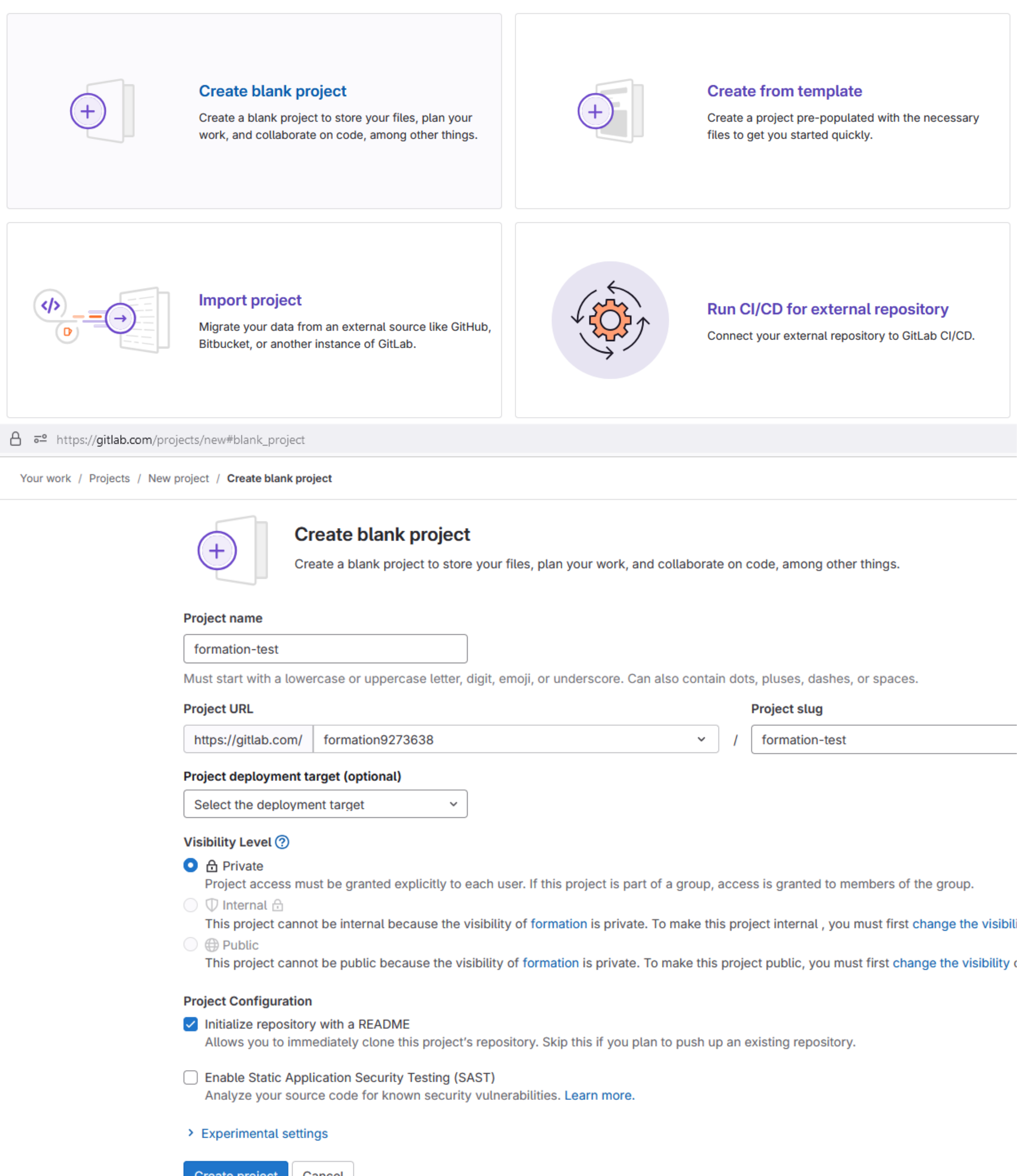


git remote add <name> <url>

git remote rm <name>

git remote rename <oldname> <newname>

Nous allons maintenant aller dans Gitlab et créer le projet « **formation-test** ».



https://gitlab.com/formation9273638/formation-test

formation / formation-test

Project 'formation-test' was successfully created.

Ultimate Trial Day 5/30

Learn about features

Project

formation-test
Learn GitLab 33%
Pinned
Merge requests 0
Manage
Plan
Code
Build
Secure
Deploy
Operate
Monitor
Analyze
Settings

F formation-test

main | formation-test / | + | History | Find file | Edit | Code

Initial commit
El Hadji Gaye authored just now — 12ff8fa1

Name	Last commit	Last update
README.md	Initial commit	just now

README.md

formation-test

Getting started

To make it easy for you to get started with GitLab, here's a list of recommended next steps.

Already a pro? Just edit this README.md and make it your own. Want to make it easy? Use the template at the bottom!

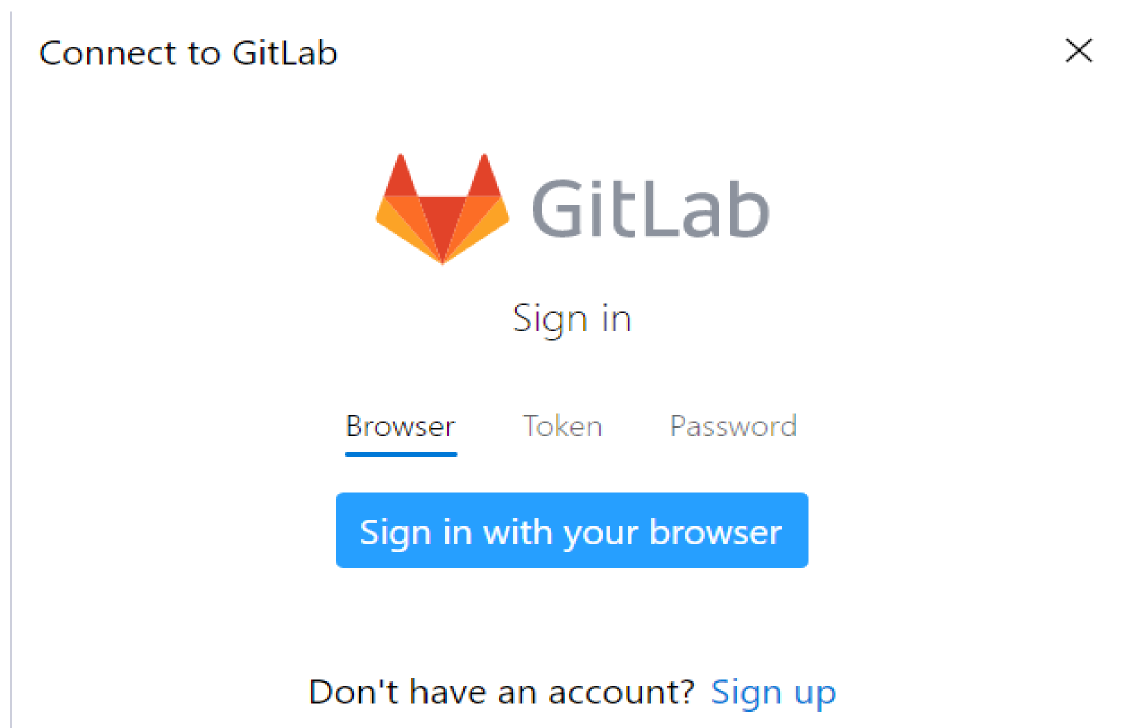
Add your files

- [ ] Create or upload files
- [ ] Add files using the command line or push an existing Git repository with the following command:

```
cd existing_repo
git remote add origin https://gitlab.com/formation9273638/formation-test.git
git branch -M main
git push -uf origin main
```

Se placer sur le repertoire Git/courses puis lancer la commande git clone

cd Git/courses
git clone https://gitlab.com/formation9273638/formation-test.git

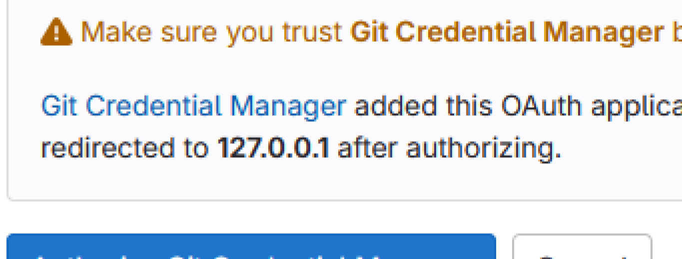


Git Credential Manager is requesting access to your account on GitLab.com.

El Hadji Gaye · @elhadji.gaye83

Allows read-write access to the repository
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

Allows read-only access to the repository
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.

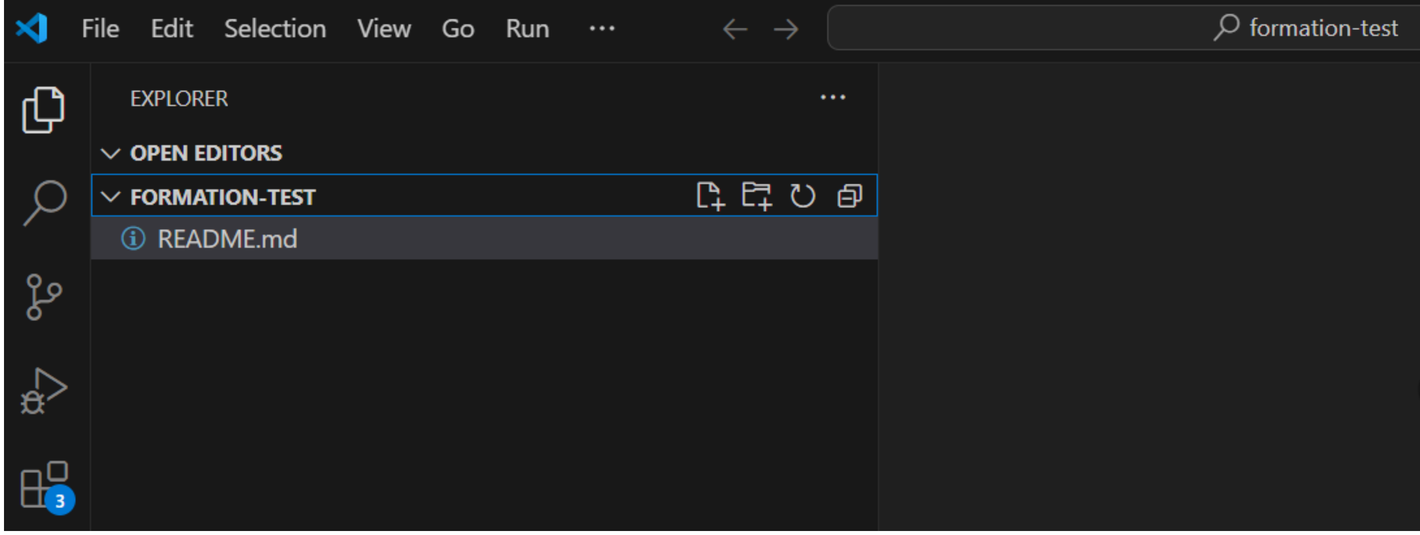


Authentication successful

You can now close this page.

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses
$ git clone https://gitlab.com/formation9273638/formation-test.git
Cloning into 'formation-test'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
```

Ouvrir le dossier **Git/courses/formation-test** avec visual studio code.



Ouvrir un terminal bash et lancer les commandes :

cd .git
ls
cat config

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test (main)
$ cd .git

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git (GI
T_DIR!)
$ ls
config  description  HEAD  hooks/  index  info/  logs/  objects/  packed-refs  refs/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git (GI
T_DIR!)
$ cat config
[core]
        repositoryformatversion = 0
        filemode = false
        bare = false
        symlinks = false
        ignorecase = true
[remote "origin"]
        url = https://gitlab.com/formation9273638/formation-test.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
        vscode-merge-base = origin/main
```

Créer le fichier **index.html** avec le contenu :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  <title>Page Title</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
</head>
<body>

</body>
</html>
```

git add index.html
git commit -m "first commit message"
git push origin main

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test (main)
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 454 bytes | 454.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://gitlab.com/formation9273638/formation-test.git
   12ff8fa..ae0e3a1  main -> main
```

cd .git
ls
cd refs
ls
cd remotes
ls
cd origin
ls
cat main
cat HEAD

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test (main)
$ cd .git

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git (GIT_DIR
!)
$ ls
COMMIT_EDITMSG  config  description  HEAD  hooks/  index  info/  logs/  objects/  packed-refs  refs/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git (GIT_DIR
!)
$ cd refs

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs (GI
T_DIR!)
$ ls
heads/  remotes/  tags/

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs (GI
T_DIR!)
$ cd remotes

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/rem
otes (GIT_DIR!)
$ ls
origin/
```

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/rem
otes (GIT_DIR!)
$ cd origin

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/rem
otes/origin (GIT_DIR!)
$ ls
HEAD  main

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/rem
otes/origin (GIT_DIR!)
$ cat main
ae0e3a178295da61d19ff7a8fcf386f8c20ca506

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/rem
otes/origin (GIT_DIR!)
$ cat HEAD
ref: refs/remotes/origin/main

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/rem
otes/origin (GIT_DIR!)
```

cd ..
cd heads
ls
cat main

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/remotes/or
igin (GIT_DIR!)
$ cd ..

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/remotes (G
IT_DIR!)
$ cd ..

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs (GIT_DIR!)

$ cd heads

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/heads (GIT
_DIR!)
$ ls
main

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/heads (GIT
_DIR!)
$ cat main
ae0e3a178295da61d19ff7a8fcf386f8c20ca506

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test/.git/refs/heads (GIT
_DIR!)
```

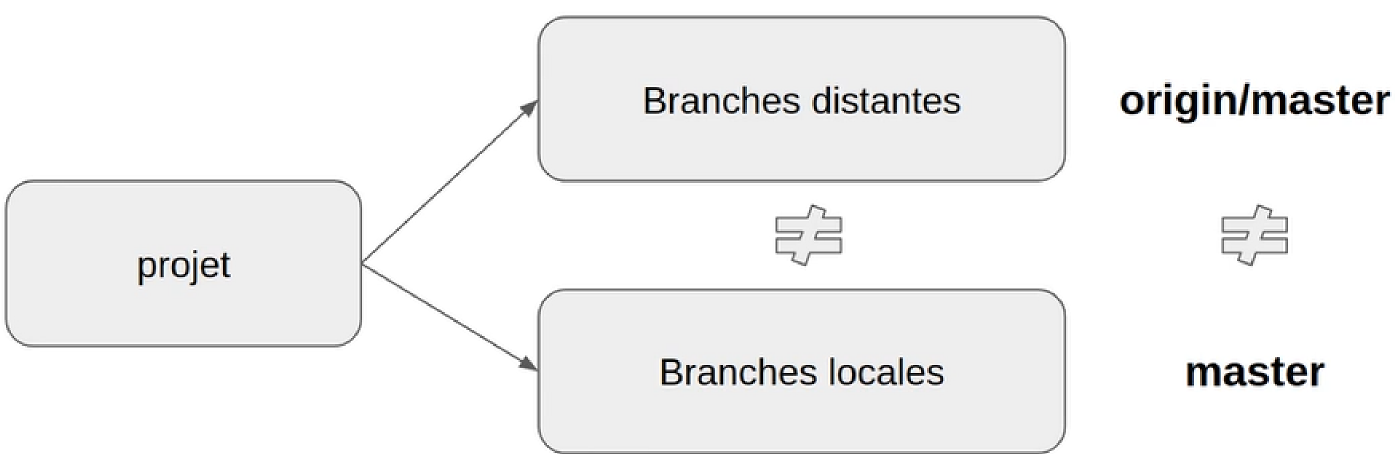
Regardons maintenant au niveau de la commande git remote.

git remote
git remote -v

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test (main)
$ git remote
origin

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test (main)
$ git remote -v
origin  https://gitlab.com/formation9273638/formation-test.git (fetch)
origin  https://gitlab.com/formation9273638/formation-test.git (push)

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses/formation-test (main)
```

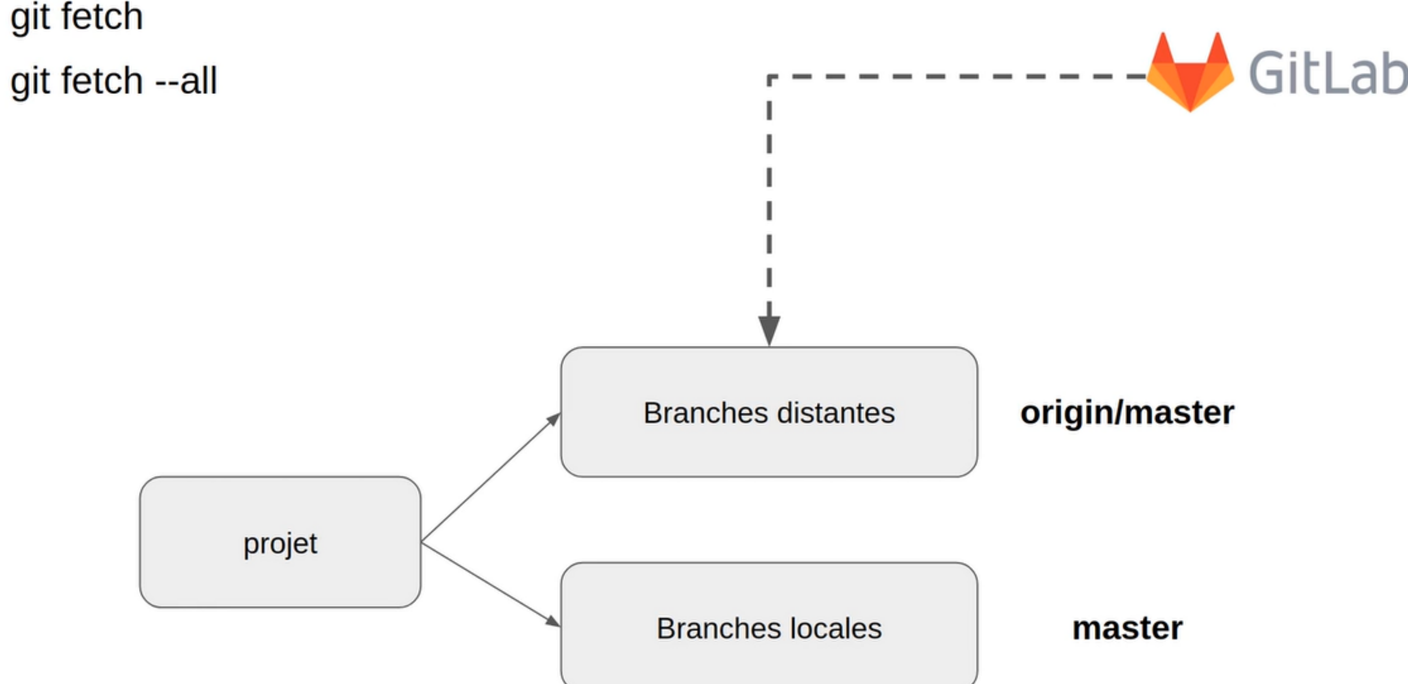
5. *Mise à jours pointeurs distants avec git fetch*

Dans cette partie de la formation nous allons nous interessé à git fetch.



La branche origin/master est la branche master telle qu'elle l'est dans origin et la branche master est la branche master telle qu'elle l'est dans notre repertoire local. Ce ne sont pas les mêmes branches et n'ont pas les mêmes commit.

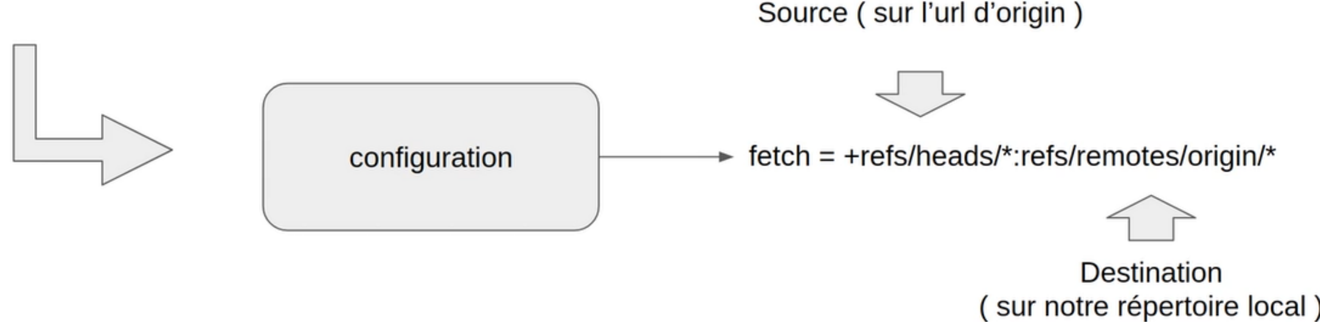
Attention les branches distantes et les branches locale sont differentes et dissocié.



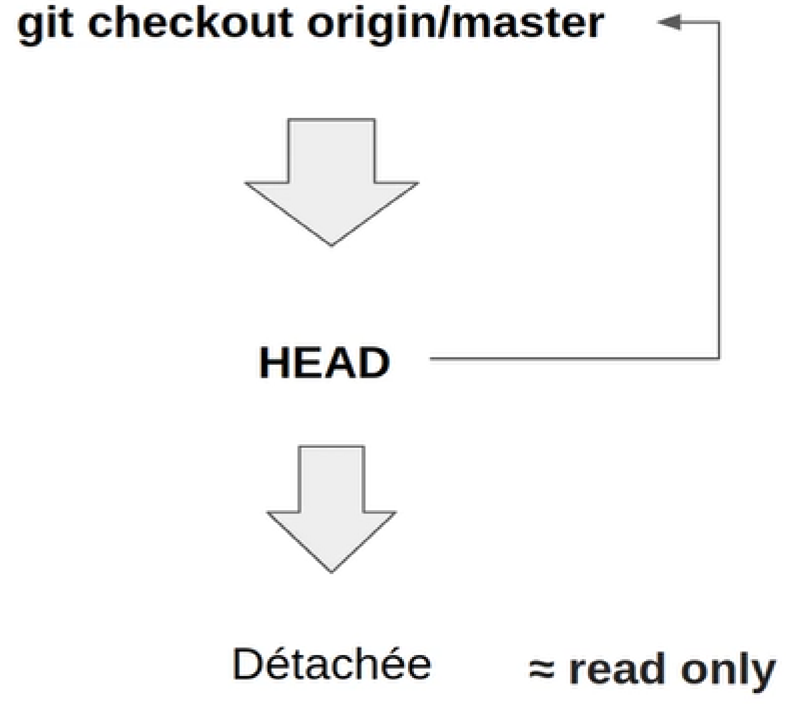
On recupère les modifications de « **origin/master** » vers « **master** » en local. Cette mise à jour se fera par merge ou par rebase.

On recupère donc les branches heads contenu sur origin (repertoires distantes) pour les charger vers nos branches en local.

git fetch origin



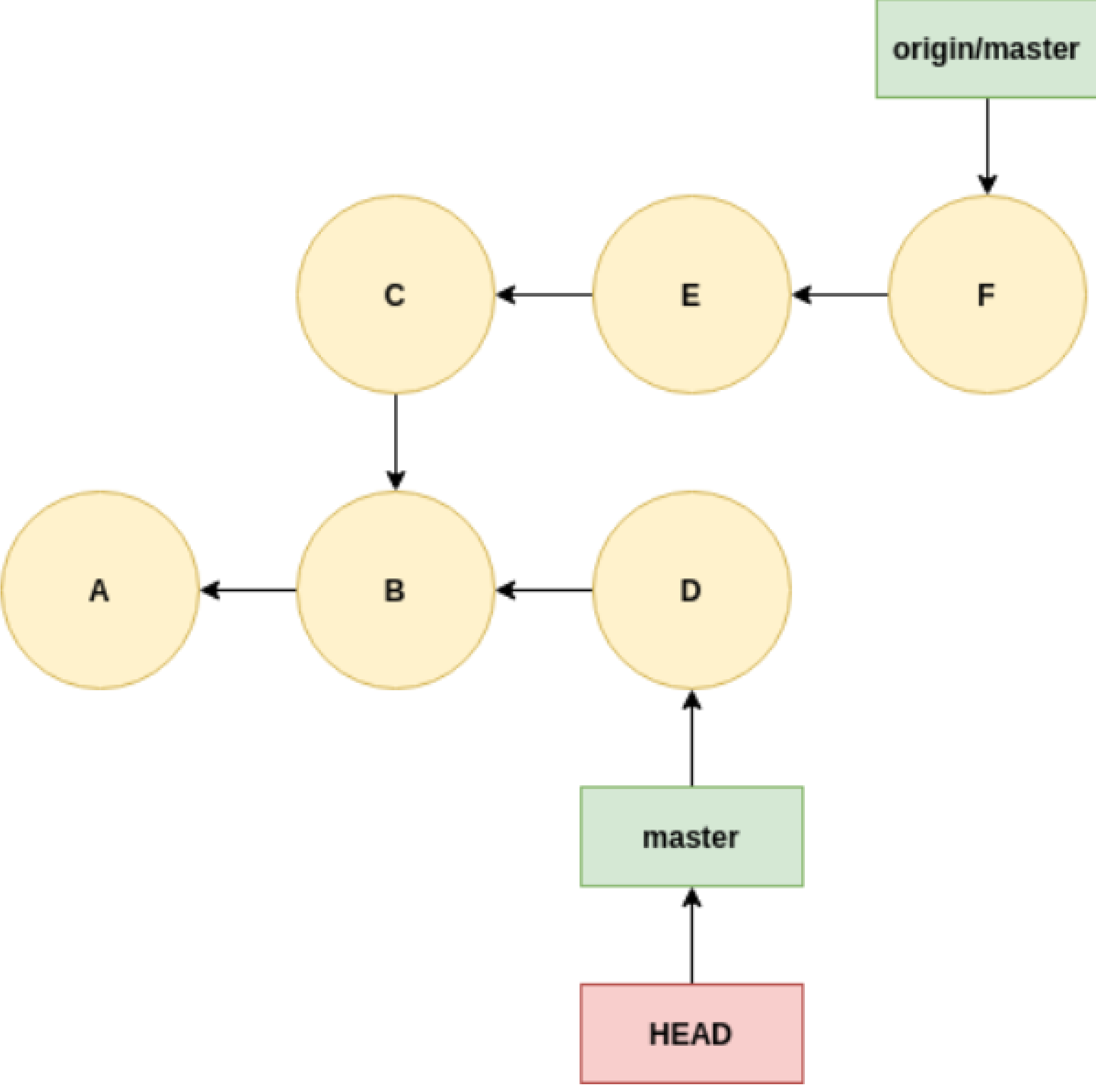
On peut aussi faire un checkout sur origin/master aussi.



6. La commande git pull

La commande git pull est le raccourci de deux commandes git fetch puis git merge.
Lorsque vous faites la commande, l'éditeur s'ouvre pour avoir l'opportunité de modifier le message du commit de fusion.

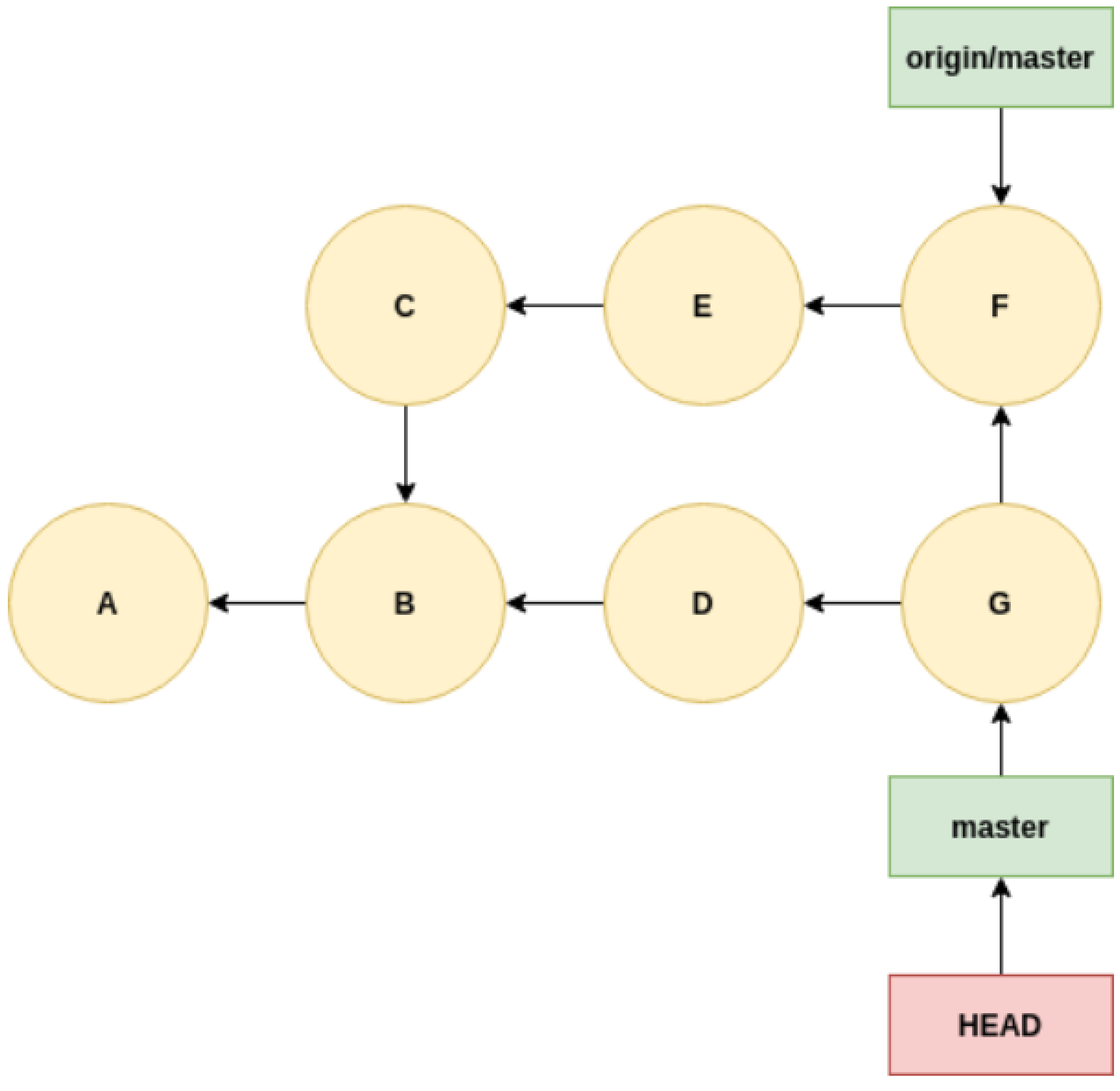
Exemple :



En faisant :

git pull

Nous aurons donc :



Le commit G étant un commit de fusion.
Si il y a des conflits, vous devrez bien sûr les résoudre.
Si vous ne souhaitez pas résoudre les conflits de fusion immédiatement vous pouvez annuler la fusion :

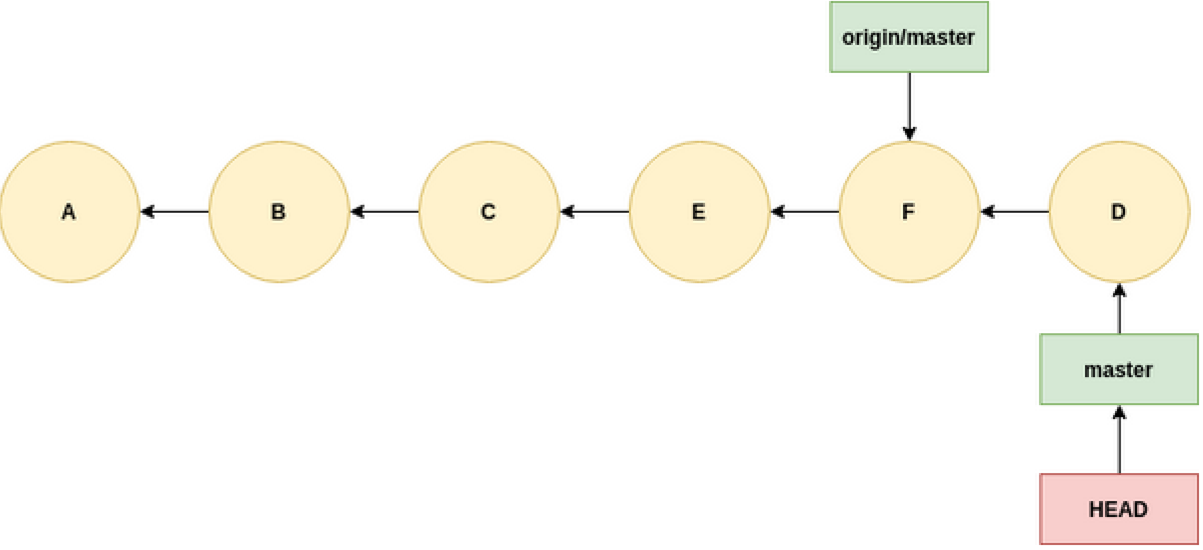
git reset --merge

Avec l'option –rebase

Vous pouvez également utiliser l'option rebase pour effectuer un rebasage plutôt qu'une fusion après le fetch :

git pull --rebase

Nous reprenons la situation initiale, mais cette fois-ci nous utilisons le rebasage.



7. La commande git push

La commande git push permet de mettre à jour les références sur le dépôt distant.

Autrement dit, elle permet d'envoyer vos changements pour mettre à jour le dépôt distant.

Ce que vous devez bien comprendre, c'est qu vos branches locales ne sont pas automatiquement synchronisées avec votre dépôt distant. Il faut utiliser git push.

Comme pour les autres commandes, si aucun dépôt distant n'est précisé, origin sera utilisé.

Pour ce qui est de la branche, par défaut, c'est la branche sur laquelle est positionnée HEAD qui est push vers la branche distante correspondante.

Donc par défaut, si vous êtes sur main :

git push

Équivaut à :

git push origin main

8. Protocole ssh avec Git et Gitlab

Le protocole SSH

SSH est un protocole de connexion sécurisé qui permet d'authentifier et de chiffrer les segments TCP (les paquets transportés sur le réseau).

SSH peut utiliser l'authentification par mot de passe ou en utilisant la cryptographie asymétrique. C'est cette dernière qui est utilisée avec Git.

La cryptographie asymétrique utilise une clé publique et une clé privée. La clé publique est ajoutée sur les serveurs où l'on souhaite se connecter par un administrateur, la clé privé reste uniquement sur l'ordinateur.

Ces deux clés permettent dans cette configuration deux choses : l'authentification et le chiffrement des communications.

Voici le déroulement :

1 - La clé publique ajoutée au serveur Git ou sur Gitlab ou Github comme une clé d'authentification valide pour ce compte.

2 - Lorsque vous essayez de faire une action sur l'hôte distant, Git va utiliser SSH si vous avez cloné le répertoire en SSH, sinon nous verrons comment le configurer. Il va envoyer une demande de connexion authentifiée par une signature utilisant la clé privée.

3 - L'hôte distant utilise la clé publique pour s'assurer de l'authenticité du message : en effet, seul le détenteur de la clé privée peut effectuer une signature correspondant à la clé publique.

4 - Les communications sont ensuite chiffrées dans les deux sens.

Générer une paire de clés

Vous pouvez commencer par vérifier si une paire de clé existe sur votre machine :

cd ~/.ssh
ls

Si vous avez un fichier id_rsa et un fichier id_rsa.pub, vous avez une paire de clé publique / privée.

Nous allons utiliser la librairie ssh-keygen qui est disponible sur tous les environnements (Windows, Linux et MacOS) et installée par défaut..

Il suffit de faire :

ssh-keygen

Cela générera par défaut une paire de clés de longueur de 2048 bits en utilisant l'algorithme RSA.

Tapez entrée à toutes les questions, sauf si vous utilisez un ordinateur partagé dans ce cas entrez un mot de passe lorsque cela vous sera demandé. Ce mot de passe sera demandé à chaque fois que la paire de clé sera utilisée.

Par défaut les clés seront enregistrées dans /home/utilisateur/.ssh/.

Où utilisateur est le nom de votre utilisateur courant.
Les options courantes sont :

-t permettant de spécifier le type d'algorithme utilisé pour la génération de clés. Laissez le par défaut, RSA est le standard.

-C pour comment : cela permet d'insérer un commentaire dans la clé. Le plus souvent on précise un email ou un identifiant pour permettre plus facilement de connaitre le propriétaire de la clé (attention ces informations sont incluses dans la clé publique et sont donc accessibles).

-b permet de préciser la longueur de la clé en bits. Par défaut, suivant votre environnement ce sera 2048 bits et parfois 3072 bits. Vous pouvez préciser -b 4096 pour allonger la clé et la rendre plus sécurisée, mais cela ne sert à rien en l'état actuelle des puissances de calcul disponible.

Pour retrouver vos clés rendez vous dans :

cd /home/utilisateur/.ssh

Ou, le raccourci :

cd ~/.ssh

Vous devrez copier la clé publique pour la mettre sur Gitlab.

Affichez là en faisant :

cat ~/.ssh/id_rsa.pub

Elle doit ressembler à :

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQC+OAb59N+WS68D
/xiYdwPPqAOqbrhFAney4jJVMiKMVrt0rYC9V+80mjKB9WA1jpf4e+3AtrXMDqiq2rbd
MuQbkjzbjSCMzl57vJlXkixNpAgzdF3HU13RhMvi+PVMrbKwPIMMQPgdJIemRm85HI4G
Pz0WaTDezBwLOvTbmiPzJl/LbfmcCGD8ViciPOhlLFcj3nPLzmU3PtE8UC57SoiYaXYl
Guc+p0tQN7TjvITQHNvSvukzOQTZjC/w+52v6tmjULej6nLXRN4wTEpB21lnYw+UYRY2
egd385yGVx13kIs/4DR9apD/rW4BqwOHuVDHwur5f41tlys3Rg0hAgysSWjhQaT+qt7s
y3PgkM7p0IAXeGuOst7IKXt+vTTgUrhBMVWiOb1dHvCcdjaCASdQ/3U1JE5Anqe+eoim
zSr3vJQfpZfZwS34lQqqRnQgIjwiHXPPJxpFxd7mctt7AnVCLMpOnKZLpaRRVei3FjQ6
mwlqOXpanVyivvIVx60MHItKbw1ZV4/NOHri5B+3qC4OQKIVB0iUmmjxEqxXXw84EV5v
AmasWrztShBID+WCezvYIG3Bw3NHiVxv+cnBmrRSLLToZGkAFbjoDLE7mckLVXtUmA+/
wWTuznI7O//7jxgm7eowbyM9JmLIrtx1hvuIBg9KsTkAYhj59s5knd37eQ==
```

ous pouvez ensuite la copier et vous rendre sur Gitlab.

Allez sur Profil (en haut à droite), puis Paramètres.

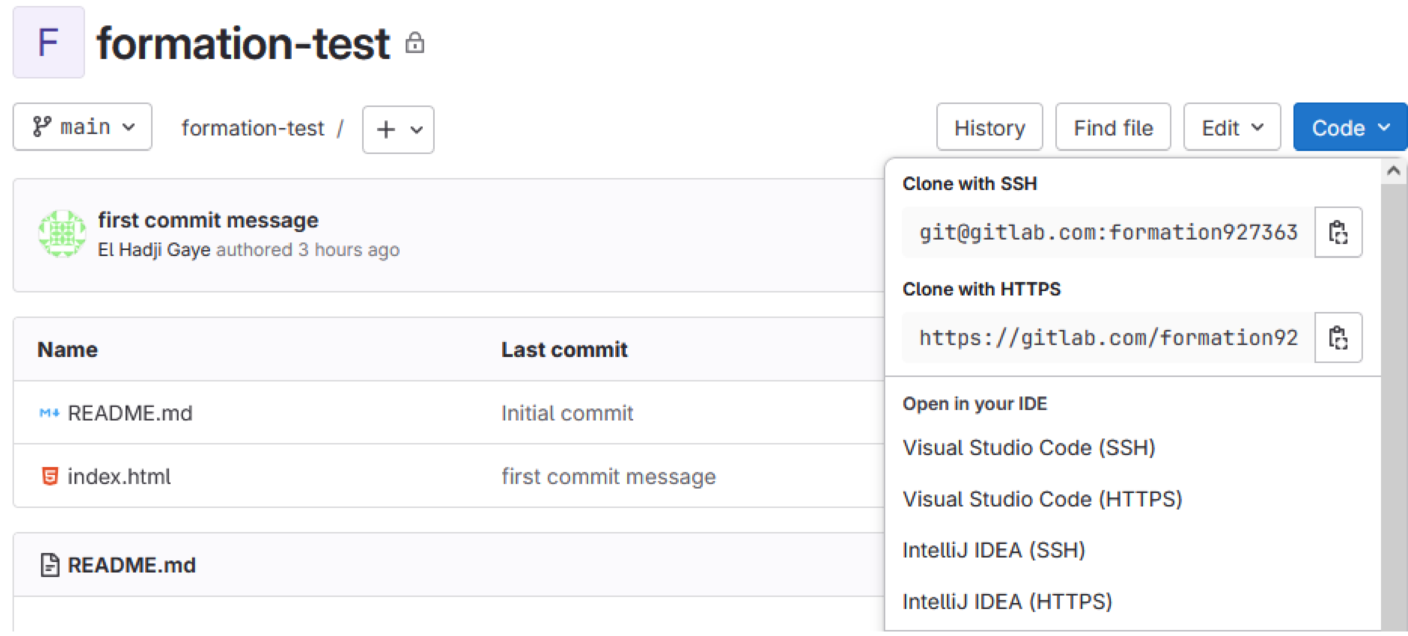
Ensuite dans la colonne de gauche allez dans Clefs SSH.

Copier la clé dans l'espace et cliquez sur Ajouter une clef.

Vous pouvez maintenant vous connecter en utilisant SSH !

Revenons maintenant à notre projet.

Se rendre dans notre projet https://gitlab.com/formation9273638/formation-test



Supprimer le precedant projet « **formation-test** » en local et essayer de le re-cloner en SSH.

Ici il s'agit de l'URL git@gitlab.com:formation9273638/formation-test.git

Ouvrir un git bash et processer :

cd Git/courses
git clone git@gitlab.com:formation9273638/formation-test.git

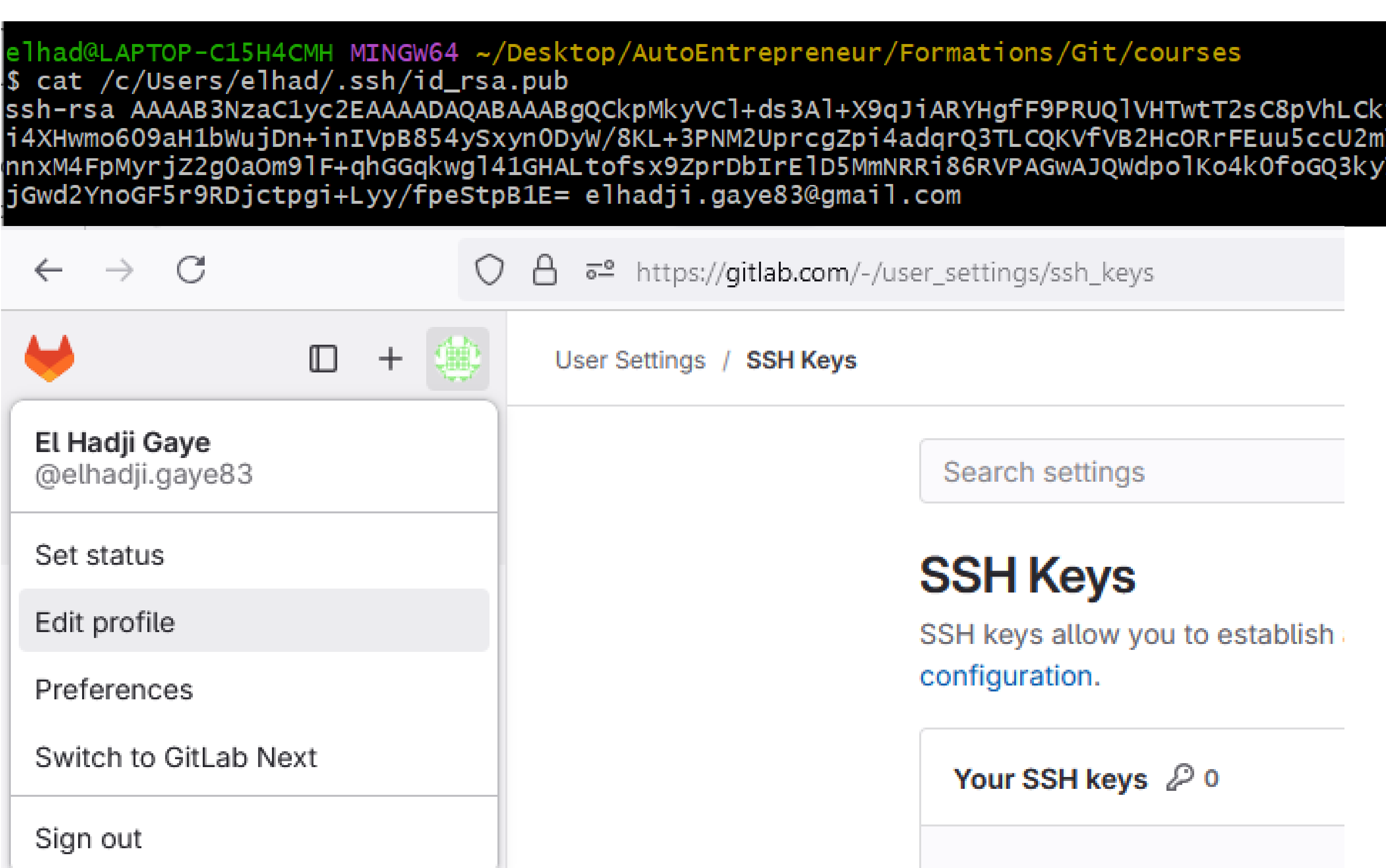
MINGW64:/c/Users/elhad/Desktop/AutoEntrepreneur/Formations/Git/courses

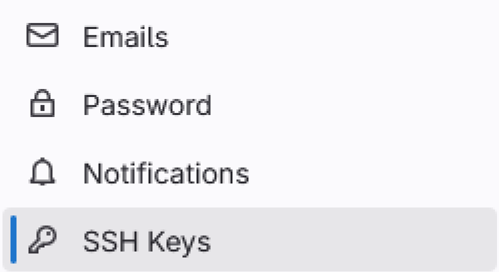
```
elhad@LAPTOP-C15H4CMH MINGW64 ~
$ cd C:/Users/elhad/Desktop/AutoEntrepreneur/Formations/Git/courses

elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses
$ git clone git@gitlab.com:formation9273638/formation-test.git
Cloning into 'formation-test'...
The authenticity of host 'gitlab.com (172.65.251.78)' can't be established.
ED25519 key fingerprint is SHA256:eUXGGm1YGsMAS7vkcx6JOJdOGHPem5gQp4taiCfCLB8.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
Host key verification failed.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

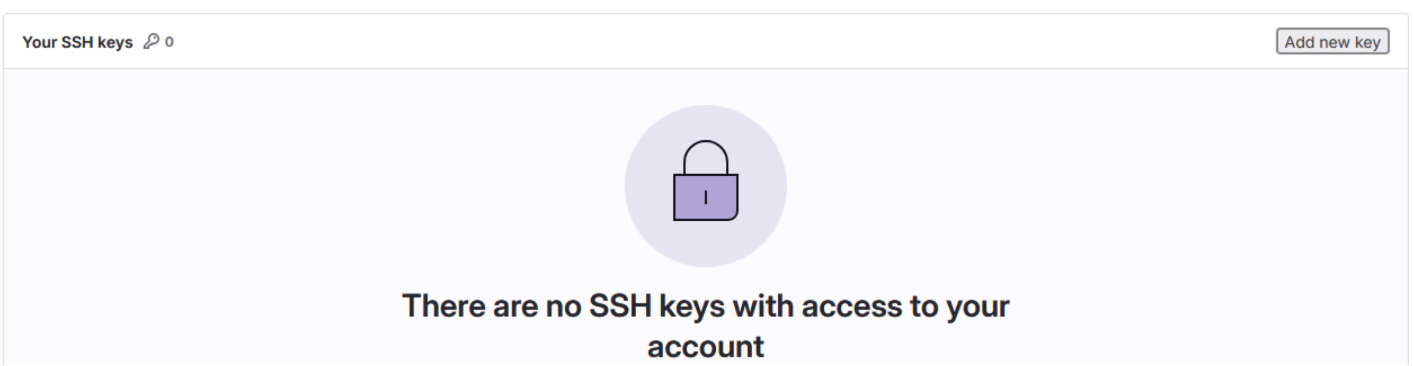
ssh-keygen -t rsa -C "elhadji.gaye83@gmail.com"

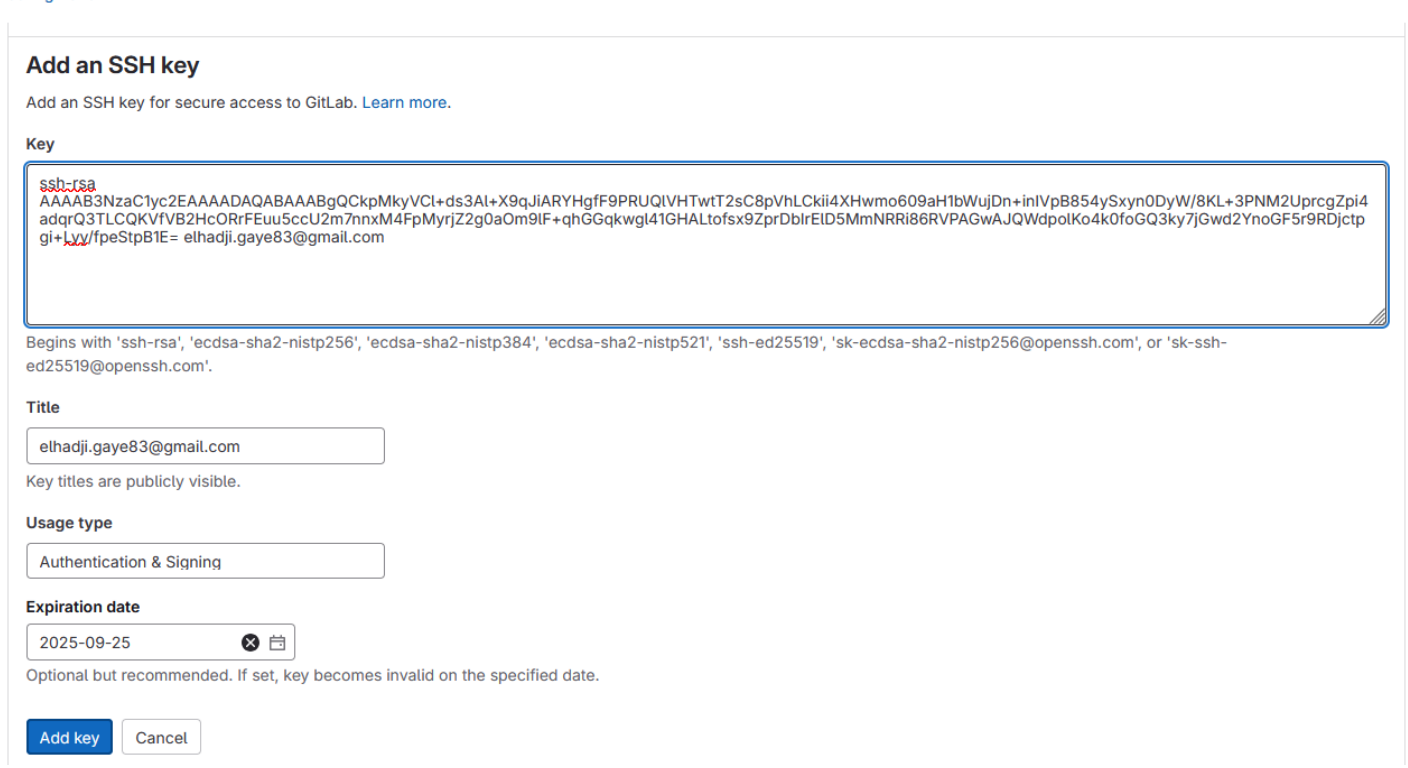
```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses
$ ssh-keygen -t rsa -C "elhadji.gaye83@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/elhad/.ssh/id_rsa):
/c/Users/elhad/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/elhad/.ssh/id_rsa
Your public key has been saved in /c/Users/elhad/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:WXi4zksaafld1GjmorUow1ibY7/cQT1yv7QRebyijgM elhadji.gaye83@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|                 |
|         o       |
|        o o      |
|         = . o o |
|        S o O + o|
|      .E . B o o.|
|     +=o= + o.+. |
|    ..o* o.=...+ |
|     ..=Oo*o  o  |
+----[SHA256]-----+
```

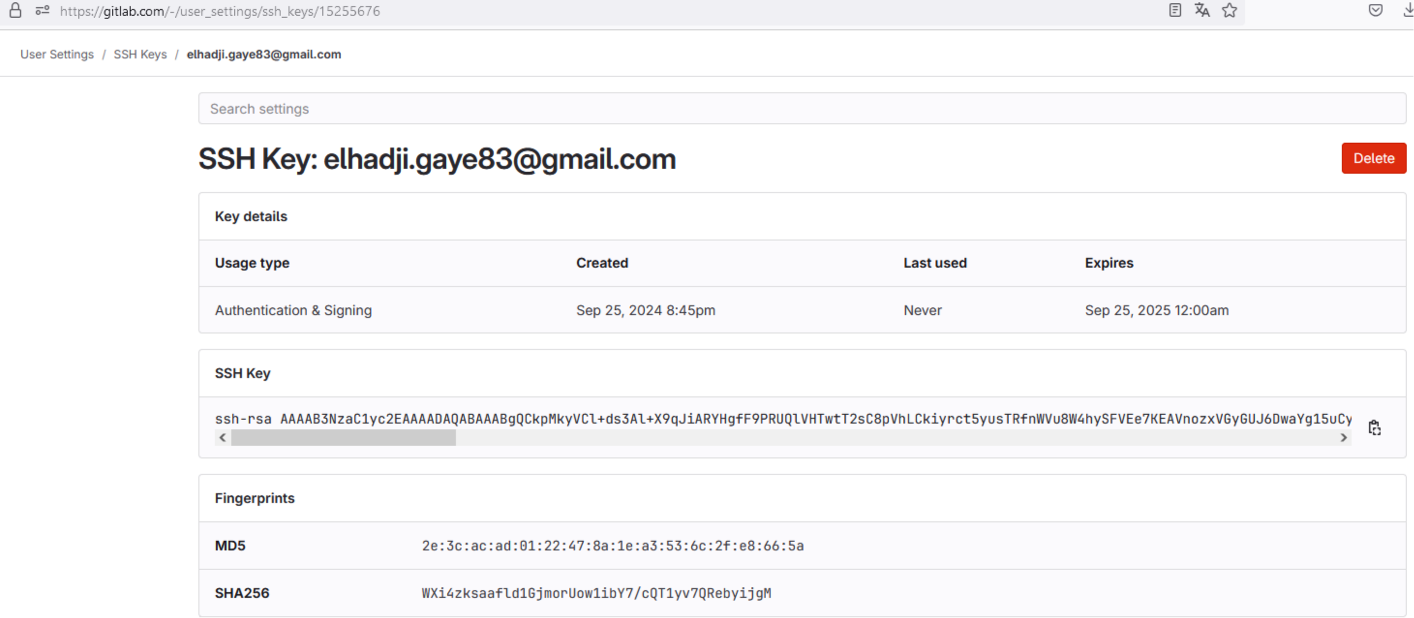
Recupèrer le fichier C:/Users/elhad/.ssh/id_rsa.pub




Emails
Password
Notifications
SSH Keys


SSH Keys
SSH keys allow you to establish a secure connection between your computer and GitLab. SSH fingerprints verify that the client is connecting to the correct host. Check the current instance configuration.
Your SSH keys 0
Add new key
There are no SSH keys with access to your account


SSH Keys
SSH keys allow you to establish a secure connection between your computer and GitLab. SSH fingerprints verify that the client is connecting to the correct host. Check the current instance configuration.
Add an SSH key
Add an SSH key for secure access to GitLab. Learn more.
Key
Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'.
Title
elhadji.gaye83@gmail.com
Key titles are publicly visible.
Usage type
Authentication & Signing
Expiration date
2025-09-25
Optional but recommended. If set, key becomes invalid on the specified date.
Add key
Cancel



On va pouvoir réessayer notre git clone.

git clone git@gitlab.com:formation9273638/formation-test.git

```
elhad@LAPTOP-C15H4CMH MINGW64 ~/Desktop/AutoEntrepreneur/Formations/Git/courses
$ git clone git@gitlab.com:formation9273638/formation-test.git
Cloning into 'formation-test'...
The authenticity of host 'gitlab.com (172.65.251.78)' can't be established.
ED25519 key fingerprint is SHA256:eUXGGm1YGsMAS7vkcx6JOJdOGHPem5gQp4taiCfCLB8.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'gitlab.com' (ED25519) to the list of known hosts.
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
```

V) Git-Flow : workflow d'entreprise

1. Introduction

Travailler avec Git en équipe n'est pas quelque chose d'intuitif. Il faut se former et essayer plusieurs façons de travailler pour trouver celle qui convient le mieux. En effet, pour que l'utilisation d'un gestionnaire de version soit efficace, il faut que les intervenants suivent les mêmes pratiques.

Par exemple, il n'est pas facile pour un développeur de savoir quand créer une branche, quand créer un tag, ou encore de savoir dans quelle branche doit être mergée la branche d'une nouvelle fonctionnalité.

Vincent Driessen a proposé un système efficace de gestion des branches sur son blog professionnel (http://nvie.com/posts/a-successful-git-branching-model). Ce système de gestion des branches est voué à être utilisé par des équipes de petite taille jusqu'à des équipes importantes. Le but de ce système est de séparer efficacement les branches et les différentes versions du projet. Cette méthode de travail est largement répandue dans les projets utilisant Git, elle reçoit néanmoins quelques critiques de développeurs la jugeant trop complexe et générant de nombreux conflits inutiles.

2. *Les branches éternelles*

Les branches master et develop sont les branches qui ne seront jamais supprimées pendant toute la durée de vie du projet. Ce sont les seules branches à exister au début du projet et à la fin du projet.

La branche de production (master ou main)

Cette branche est la branche qui contiendra toutes les versions publiées pour les utilisateurs. C'està-dire que c'est cette branche qui recevra toutes les nouvelles fonctionnalités et toutes les corrections de bugs. Chaque commit de cette branche est représenté par une nouvelle version spécifiée par un tag. Tous les commits de cette branche sont des commits produits par les merges d'autres branches (à l'exception du commit d'origine).

Voici les branches qui peuvent être créées à partir de master :

- la branche develop (qui ne sera créée qu'une seule fois au début du projet),
- les branches de correctifs (hotfix).

Voici les branches dont master pourra recevoir les modifications au travers d'un merge :

- les branches de nouvelles versions (release),
- les branches de correctifs (hotfix).

La branche de développement (develop)

La branche de développement correspond à la branche qui recevra toutes les nouvelles fonctionnalités qui ne sont pas encore intégrées à la version principale. Cette branche correspond aux futures versions qui seront publiées et n'est pas encore considérée comme stable. Cette branche est créée dès le début du projet à partir de la branche master.

Voici les branches qui peuvent être créées à partir de la branche develop :

- les branches de fonctionnalités (feature),
- les branches de versions (release).

Voici les branches à partir desquelles develop pourra recevoir les modifications :

- les branches de correctifs (hotfix),
- les branches de versions (release),
- les branches de fonctionnalités (feature).

3. Les branches éphémères

Les branches éphémères sont des branches qui ont une durée de vie limitée. Elles sont créées dans un but très précis et, une fois celui-ci accompli, ces branches sont supprimées.

Les branches de versions (release)

Ces branches sont des branches de versions Beta du projet. Prenons l'exemple d'un logiciel dont la version 2.0 est prévue très prochainement. L'entreprise éditrice décide un mois avant la publication de cette version de créer une branche release pour préparer cette sortie. Avant de créer cette branche, l'entreprise doit s'assurer que toutes les nouvelles fonctionnalités attendues pour la version 2.0 ont été intégrées dans la branche develop. Une fois que cette vérification est effectuée, la branche release-2.0 peut être créée à partir de la branche develop. Cette branche aura deux utilités principales :

- Elle va servir à rendre la prochaine version stable. C'est-à-dire que la branche release-2.0 sera testée au maximum et que des correctifs spécifiques aux nouvelles fonctionnalités seront inclus dans cette branche.

- Cette branche va permettre aux développeurs de commencer à travailler sur les fonctionnalités de la version 3.0 pendant que la version 2.0 est en phase de test. Aucune fonctionnalité propre à la version 3.0 ne doit se trouver dans la branche release-2.0.

Cette branche ne sert pas à recevoir de nouvelles fonctionnalités. Elle doit être créée à partir de develop qui doit contenir toutes les nouvelles fonctionnalités de la version 2.0.
Cette branche sera mergée dans les branches develop et master. Le commit dans la branche master donnera lieu à un commit sur lequel le tag v2.0 sera appliqué.

Cette branche sera supprimée après avoir été mergée dans master et develop.

Les branches de correctifs (hotfix)

Ces branches sont celles qui vont accueillir les correctifs destinés à la production. La création d'une branche de correctifs survient lorsqu'un bug est découvert et que le développeur commence à le corriger. Le ou les commits de cette branche sont uniquement des commits destinés à la résolution du bug (et la mise à jour des numéros de versions si nécessaire).

Une branche de correctif est créée à partir de la branche master et, une fois le correctif commité, cette branche sera mergée dans master et dans develop et sera ensuite supprimée. Pour respecter la norme de Git-Flow, il faut préfixer les noms des branches de correctifs par hotfix-.

Les branches de fonctionnalités (feature)

Chacune des branches de fonctionnalité a pour but d'intégrer une nouvelle fonctionnalité à la branche develop. Ces branches sont locales, c'est-à-dire qu'elles ne sont pas partagées sur le dépôt central. La création d'une telle branche a toujours lieu à partir de la branche develop et lorsque la nouvelle fonctionnalité est créée et commitée, cette branche doit être intégrée à develop.
Lorsque la fonctionnalité a été intégrée à develop, la branche de fonctionnalité doit être supprimée.