

Git & GitHub

CHAPITRE 1

GIT : SYSTÈME DE CONTRÔLE DES VERSIONS

Objectifs



- ✓ Comprendre les principes fondamentaux des Système de contrôle des Versions(VCS)et leur importance dans le développement de logiciels.
- ✓ Savoir utiliser Git pour suivre les modifications de code et travailler avec des branches et des révisions.
- ✓ Comprendre comment utiliser Git pour collaborer efficacement avec d'autres développeurs sur un projet.
- ✓ Être capable de résoudre les conflits de fusion et d'éviter les pertes de données.
- ✓ Comprendre comment utiliser GitHub pour héberger des dépôts Git et faciliter la collaboration.



Chapitre 1:

Les bases de Git

Plan



- 1. Qu'est ce que Git**
- 2. Fonctions &utilité des systèmes de gestion de versions (SCV)**
- 3. Modèles centralisés et décentralisés des SCV**
- 4. Qu'est ce que GitHub**
- 5. Installation et paramétrage de Git**
- 6. Fonctionnement de base de Git**
- 7. Git : commandes utiles**
- 8. Créer un dépôt Git**
- 9. Modifier un dépôt Git**
- 10. Annuler des actions et consulter l'historique Git**

Qu'est-ce que Git ?



- ❖ Git est **un système de gestion de version décentralisé open-source**, créé en 2005 par Linus Torvalds, le créateur de Linux.
- ❖ Il permet de suivre les modifications apportées à un ensemble de fichiers au fil du temps.
- ❖ Les développeurs peuvent travailler sur leur propre branche de développement et fusionner les modifications apportées par les autres, ce qui facilite la collaboration.
- ❖ Git permet de résoudre les conflits qui peuvent survenir lors de la fusion de modifications.
- ❖ Git est rapide, fiable et peut être utilisé pour gérer des projets de toute taille.
- ❖ Git est compatible avec une large gamme d'outils et de services de développement, ce qui en fait un choix populaire pour la gestion de version de code.

Fonctions & utilité des SCV (1)



- **Suivi des modifications** : Les SCV enregistrent chaque version et les modifications apportées, ce qui permet aux développeurs de retrouver facilement des versions antérieures du code et de comprendre son évolution.
- **Collaboration** : Les SCV permettent à chaque développeur de travailler sur sa propre branche, ce qui facilite la collaboration et la fusion des modifications apportées.
- **Gestion des conflits** : Les SCV permettent de gérer les conflits qui peuvent survenir lors de la fusion de modifications apportées par différents développeurs.
- **Sauvegarde** : Les SCV offrent une sauvegarde fiable et efficace de l'ensemble du code source et de son historique, ce qui réduit les risques de perte de données.
- **Développement itératif** : Les SCV permettent de travailler de manière itérative en travaillant sur des fonctionnalités spécifiques de manière isolée, ce qui facilite le développement par étapes et réduit les risques de régressions.

Fonctionnalités



Git a donc trois grandes fonctionnalités :

1. Revenir à une version précédente de notre code en cas de problème.
2. Suivre l'évolution de notre code étape par étape.
3. Travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs.

→ Git se repose sur deux modèles des logiciels : modèle centralisé & décentralisé

Modèle centralisé & décentralisé (1)



- ❖ Les logiciels de gestion de version sont tous construits sur l'un des deux modèles suivants :
 - le modèle **centralisé**
 - le modèle **décentralisé** appelé encore modèle **distribué**.
- ❖ Le principe de base d'un modèle **centralisé** est la **centralisation du code source** lié au projet : la source du code du projet est hébergé sur un serveur distant central et **les différents utilisateurs doivent se connecter à ce serveur** pour travailler sur ce code.
- ❖ Dans un modèle **décentralisé** (distribué), le principe de base est **opposé** : le code source du projet est toujours hébergé sur un serveur distant mais chaque utilisateur est invité à **télécharger et à héberger l'intégralité du code source du projet sur sa propre machine**.

Différence entre dépôt local et dépôt distant ? (1)



- ❖ Un **dépôt** est un **dossier** où Git stocke les **fichiers** de code source ainsi que leur **historique** de modifications.
- ❖ Il peut être **local** ou **distant**.
 - un **dépôt local** qui gère l'historique avec lequel nous travaillons,
 - un **dépôt distant** avec lequel nous synchronisons notre dépôt local, lorsqu'on a besoin de **partager** ou **sauvegarder notre projet**.
- ❖ Git utilise une **architecture** dite **distribuée**.
- ❖ L'idée étant de permettre :
 - en premier lieu de **travailler localement**, sans dépendance réseau (contrairement aux architectures centralisées),
 - en second lieu de favoriser la **redondance de l'historique sur chaque machine locale**.

Différence entre dépôt local et dépôt distant ? (2)



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

- ❖ On réalise une **version**, que l'on va petit à petit **améliorer**.
- ❖ Ces **versions** sont **stockées** au fur et à mesure dans le **dépôt local**.

Un **dépôt distant** permet de stocker les différentes versions de notre code afin de garder un **historique délocalisé**, c'est-à-dire un historique hébergé sur Internet ou sur un réseau.

- ❖ On peut avoir plusieurs dépôts distants avec des droits différents (lecture seule, écriture,...).
 - ❖ Imaginez qu'on a des codes importants enregistrés sur notre ordinateur, mais soudainement, le PC est endommagé.
- Solution : Copier nos sources sur un **dépôt distant** lorsqu'on commence un nouveau projet, avec **GitHub par exemple** !
- On pourra aussi les **rendre publics**, et chacun pourra y ajouter ses évolutions.



Qu'est-ce que GitHub?

- ❖ GitHub est une plateforme en ligne basée sur Git.
 - ❖ Il permet la gestion de projet collaborative.
 - ❖ GitHub offre un hébergement gratuit pour les dépôts Git.
 - ❖ Il facilite la collaboration entre les développeurs.
 - ❖ Il fournit des outils de communication tels que les commentaires, les notifications et les discussions.
 - ❖ GitHub est largement utilisé pour l'hébergement de projets open source.
- ➔ Pour récapituler, Git est un logiciel de gestion de version tandis que GitHub est un service en ligne d'hébergement de dépôts Git qui fait office de serveur central pour ces dépôts.

Git ou GitHub :Quelle est la différence ?

Résumé

Git	Github
C'est un logiciel	C'est un service
Il est installé localement sur le système	Il est hébergé sur le Web
C'est un outil de ligne de commande	Il fournit une interface graphique
C'est un outil de gestion de différentes versions des modifications apportées aux fichiers d'un référentiel git.	C'est un espace pour télécharger une copie du référentiel Git



Installation de Git

The screenshot shows the official Git website at <https://git-scm.com/>. The main navigation bar includes links for 'About', 'Documentation', 'Downloads', and 'Community'. The 'Downloads' section is highlighted. It features sections for 'macOS', 'Windows', and 'Linux/Unix'. A prominent callout box highlights the 'Latest source Release 2.40.0' with a 'Download for Windows' button. Below this, there's a note about older releases and a GitHub link. Other sections include 'GUI Clients', 'Logos', and 'Git via Git'.

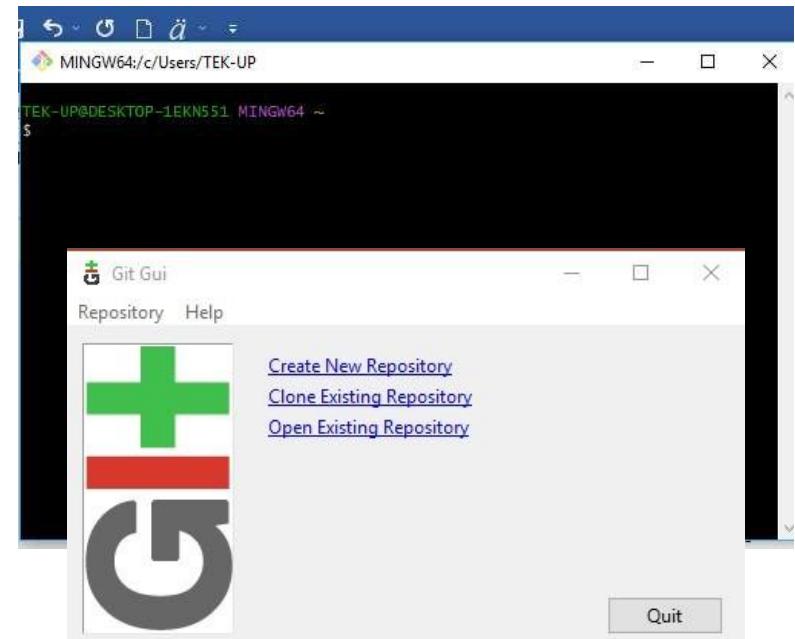
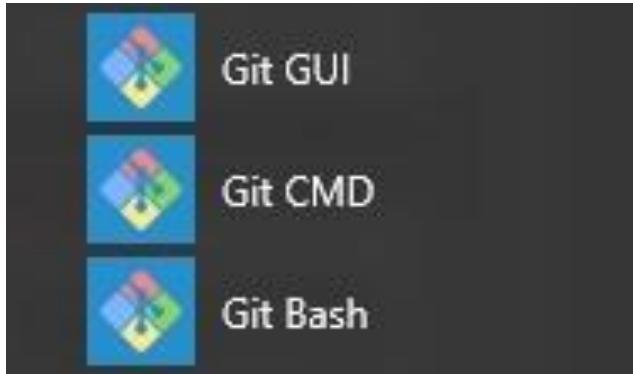
Lien : <https://git-scm.com/downloads>

Installation de Git (1)



Vérifier l'installation

Appuyez sur Suivant à chaque fenêtre puis sur **Installer**. Lors de l'installation, laissez toutes les options par défaut, elles conviennent bien.



Si vous êtes sous Windows : cochez ensuite Launch Git Bash. Pour les utilisateurs de Mac ou Linux, votre terminal suffira amplement.

Git Bash est l'interface permettant d'utiliser Git en ligne de commande

Installation de Git (2)

Vérifier l'installation



Vérifions maintenant que l'installation s'est bien déroulé. Nous allons utiliser une commande de base de l'outil qui est “git version” et qui permet d'afficher le numéro de version de Git.

Pour cela il faut lancer le terminal “git bash”, installé en même temps que Git.

Il permet d'utiliser **Git en lignes de commandes**.

Pour le trouver, vous pouvez soit taper dans la barre de recherche du menu démarrer “git bash” soit faire un click droit sur un dossier.

Une fois le terminal lancé il ne nous reste plus qu'à taper la commande “git version”.

Si l'installation c'est bien déroulée, vous devriez avoir l'affichage suivant (au numéro de versions près)

```
TEK-UP@DESKTOP-1EKN551 MINGW64 ~  
$ git version  
git version 2.40.0.windows.1
```

Paramétrage de Git



Une fois Git installé, nous allons paramétriser le logiciel afin d'enregistrer certaines données pour ne pas avoir à les fournir à nouveau plus tard.

Nous allons notamment ici renseigner un **nom d'utilisateur et une adresse mail** que Git devra utiliser ensuite.

```
git config --global user.name « leila bousbia»  
git config --global user.email leila.bousbiaa@gmail.com
```

git config permet de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git.

--global : va nous permettre d'indiquer à Git que le nom d'utilisateur et l'adresse mail renseignés doivent être utilisés globalement (c'est-à-dire pour tout projet Git).

Paramétrage de Git



Pour vous assurer que vos informations ont bien été enregistrées, vous pouvez taper:

git config <paramètre> : exemple:

`git config user.name` et `git config user.email`

Si vous souhaitez vérifier vos réglages, vous pouvez utiliser la commande **git config --list** pour lister tous les réglages que Git a pu trouver jusqu'ici :

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Fonctionnement de base de git



Un “**dépôt**” correspond à la copie et à l’importation de l’ensemble des fichiers d’un projet dans Git. Il existe deux façons de créer un dépôt Git :

- **Créer un nouveau dépôt vide** ;
- **On peut cloner un dépôt Git déjà existant.**

Nous allons voir comment faire cela dans la suite de ce cours. Avant cela, il faut comprendre comment Git conçoit la gestion des informations ainsi que le fonctionnement général de Git.

Fonctionnement de base de git



Un “**dépôt**” correspond à la copie et à l’importation de l’ensemble des fichiers d’un projet dans Git. Il existe deux façons de créer un dépôt Git :

- **Créer un nouveau dépôt vide** ;
- **On peut cloner un dépôt Git déjà existant.**

Nous allons voir comment faire cela dans la suite de ce cours. Avant cela, il faut comprendre comment Git conçoit la gestion des informations ainsi que le fonctionnement général de Git.

Fonctionnement de base de git: La gestion des informations selon G



- Git pense les données à la manière d'un flux d'instantanés ou "snapshots".
A chaque fois qu'on va valider ou enregistrer l'état d'un projet dans Git, il va prendre un instantané du contenu de l'espace de travail à ce moment et va enregistrer une référence à cet instantané pour qu'on puisse y accéder par la suite.
- Chaque instantané est stocké dans une base de donnée locale, c'est-à-dire une base de donnée située sur notre propre machine.
- Le fait que l'on dispose de l'historique complet d'un projet localement fait que la grande majorité des opérations de Git peuvent être réalisées localement, c'est-à-dire sans avoir à être connecté à un serveur central distant. Cela rend les opérations beaucoup plus rapides et le travail de manière générale beaucoup plus agréable.



Fonctionnement de base de git: Les états des fichiers :

- **Comment Git fait il pour suivre les modifications sur les fichiers d'un projet ?**
Pour comprendre cela, il faut savoir qu'un fichier peut avoir deux grands états dans Git : il peut être sous suivi de version ou non suivi.
- Pour comprendre cela, il faut savoir qu'un fichier peut avoir deux grands états dans Git : il peut être sous suivi de version ou non suivi.
- Un fichier possède l'état “**suivi**” si il appartenait au dernier instantané capturé par Git, c'est-à-dire si il est enregistré en base. Tout fichier qui n'appartenait pas au dernier instantané et qui n'a pas été indexé est “non suivi”.



Fonctionnement de base de git: Les états des fichiers :

- Lorsqu'on démarre un dépôt Git en important un répertoire déjà existant depuis notre machine, les fichiers **sont au départ tous non suivis**. On va donc déjà devoir demander à Git de **les indexer et de les valider** (les enregistrer en base).
- Lorsqu'on clone un dépôt Git déjà existant, c'est différent puisqu'on copie tout l'historique du projet **et donc les fichiers sont tous déjà suivis par défaut**.
- Chaque fichier suivi peut avoir l'un de ces trois états :
 - Modifié (“modified”);
 - Indexé (“staged”);
 - Validé (“committed”).



Fonctionnement de base de git: Les états des fichiers :

Lors du démarrage d'un dépôt Git à partir d'un dépôt local,

- on demande à Git de **valider l'ensemble des fichiers** du projet.
- **Un fichier est “validé” lorsqu'il est stocké dans la base de donnée locale.**
- Lors du clonage d'un dépôt déjà existant, les fichiers sont enregistrés par défaut en base et **ils sont donc validés** par défaut.

Ensuite, lorsqu'on va travailler sur notre projet,

- on va certainement ajouter de nouveaux fichiers ou modifier des fichiers existants.
- **Les fichiers modifiés vont être considérés comme “modifiés”** par Git tandis que les **nouveaux fichiers vont être “non suivis”**. Un fichier modifié est considéré comme “modifié” par Git tant qu'il n'a pas été indexé.

On dit qu'on **“indexe”** un fichier(**commande: git add**)

- lorsqu'on indique à Git **que le fichier modifié ou que le nouveau fichier doit faire partie du prochain instantané dans sa version actuelle.**

Lorsqu'on demande à **Git de prendre l'instantané, c'est-à-dire lorsqu'on lui demande d'enregistrer en base l'état du projet actuel** (c'est-à-dire l'ensemble des fichiers indexés et non modifiés),

- **les fichiers faisant partie de l'instantané sont à nouveau considérés comme “validés” et le cycle peut recommencer.**(**commande: git commit –m 'nom de commit: modification'**)

Fonctionnement de base de git: Les zones de travail :



- Les états de fichiers sont liés à des zones de travail dans Git. En fonction de son état, un fichier va pouvoir apparaître dans telle ou telle zone de travail.
- Tout projet Git est composé de trois sections :
 - le répertoire de travail (working tree),
 - la zone d'index (staging area) et
 - le répertoire Git (repository).

Fonctionnement de base de git: Les zones de travail :



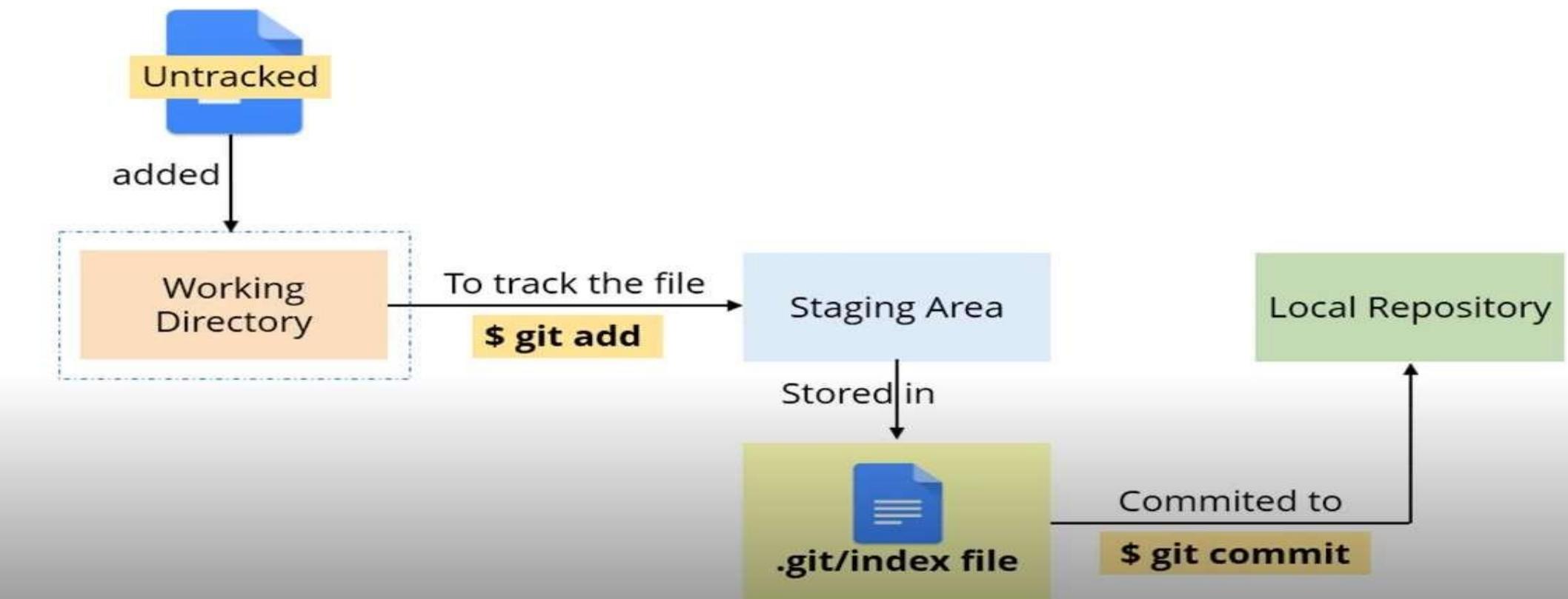
- **Le répertoire de travail ou “working tree”** : correspond à une extraction unique (“checkout”) d’une version du projet. Les fichiers sont extraits de la base de données compressée située dans le répertoire Git et sont placés sur le disque afin qu’on puisse les utiliser ou les modifier.
- **La zone d’index ou “staging area”** : correspond à un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané ou du prochain “commit”.
- **Le répertoire Git** : est l’endroit où Git stocke les méta-données et la base de données des objets de votre projet. C’est la partie principale ou le cœur de Git.

Fonctionnement de base de git: Processus de travail :



- Le processus de travail va ainsi être le suivant :
- Nous allons travailler sur nos fichiers dans le répertoire de travail.
- Lorsqu'on modifie ou crée un fichier, on peut ensuite choisir de l'indexer. Tant qu'un fichier n'est pas indexé, il possède **l'état modifié ou est non suivi** si c'est un nouveau fichier.
- **Dès qu'il est indexé que son nom est ajouté à la zone d'index, il possède l'état indexé.**
- Finalement, **on va valider (“commit”) la version indexée de nos fichiers pour les ajouter au répertoire Git.**

Fonctionnement de base de Git





Résumé

- ❖ Dépôt (repository)
- ❖ Commit
- ❖ Copie de travail
- ❖ Index



Résumé

Dépôt (repository)

- ❖ Le répertoire caché `.git` est nommé dépôt (en anglais repository).
- ❖ Il contient **toutes les données** dont GIT a besoin pour **gérer l'historique**.
- ❖ Sauf par exceptions, on **ne peut pas modifier** son contenu directement, mais uniquement en passant par **les commandes GIT**.

Résumé



Copie de travail :working copy

- ❖ On appelle copie de travail (en anglais working copy) les fichiers effectivement présents dans le répertoire géré par GIT.
- ❖ Leur état peut être différent du dernier commit de l'historique.



Résumé

L'Index

- ❖ L'index est un espace temporaire contenant les modifications prêtes à être « commitées ».
- ❖ Ces modifications peuvent être :
 - creation de fichier
 - modification de fichier
 - suppression de fichier



Remarque importante

- ❖ Bien que conceptuellement, chaque commit contient tous les fichiers du projet, GIT utilise un système de stockage très efficace :
 - le commit ne stocke que les fichiers modifiés par rapport au commit précédent;
 - l'ensemble peut être compressé (`git gc`) pour réduire encore la redondance (lorsque deux versions successives d'un fichier sont très proches).

NB : bien que GIT (et les autres VCS) soient plus particulièrement conçus pour des fichiers texte, ils fonctionnent aussi avec des fichiers binaires (images, bureautique, etc.).

Git : commandes utiles (1)

- ❖ Nous pouvons également travailler sur **Git CMD**

`mkdir projet` % création d'un projet .git

`cd .\projet` % accéder à l'intérieur de dossier projet

`git init` % initialiser un git et le versionning

`git help config` % ouvre le navigateur «git-config - Get and set repository or global options »

`git help name_of_command` % accéder à la commande demandée

`git config --global color.ui true` % améliorer la visibilité des informations par couleur

`git config --list` % afficher la liste des opérations

`Touch` % créer fichier .txt dans le dossier

`ls` % pour afficher le contenu du dossier et s'assurer que tout a bien fonctionné

Git : commandes utiles (2)

- ❖ Créer un fichier dans projet (copie1.txt)

git status % observer qu'il n'est pas suivi

➔ Nous remarquons que ce fichier n'est pas tracké « untracked » (c'est que le fichier n'est pas encore ajouté à l'index ou enregistré dans le dépôt local)

git add copie1.txt % ajouter le fichier dans l'index

git reset copie1.txt % retirer le fichier de l'index

git commit -m "mon premier commit" % garder votre code à jour avec les dernières modifications et capturez l'état du projet à ce moment-là

git commit -a -m "ajout Travail TP1 dans copie1.txt" % enregistrer la modification

➔ Vérifier dans le fichier copie1.txt l'ajout de Travail TP1

cat copie1.txt % afficher le contenu du fichier copie1.txt

Git: commandes utiles (3)

- ❖ Créer deux fichiers dans projet (copie2.txt et copie3.txt)

```
git status % observer qu'il n'est pas suivi
```

→ Nous remarquons que ce fichier n'est pas tracké

```
git add *.txt % ajouter les deux fichiers en même temps dans l'index
```

```
git commit -m "Ajouter tous les fichiers .txt"
```

```
echo "hello first file" >> file1.txt % Créer et modifier le fichier
```

```
echo "add new line" >> file2.txt % Créer et modifier le fichier
```

```
git add . % ajouter les deux fichiers dans le répertoire
```

```
git status % observer qu'il n'est pas suivi
```

```
git commit -m "my second commit "
```

Git : commandes utiles (4)

Tracking and Logs

`git log` % afficher la liste par défaut des 10 derniers commits

`git log --n` % afficher les n derniers commits

`git log --oneline` % afficher le log de commit

`git log -author= "email or name"` % afficher le commit spécifique du développeur

Git : commandes utiles (5)

- ❖ D'autres commandes utiles

`git reset <filename>` % Retirer un fichier de l'index

`git diff` % nous permet d'observer les modifications dans la zone de travail et dans l'index

`pwd` % affiche le nom et chemin du répertoire courant. Elle permet de savoir où on se situe sur notre machine

- ❖ Pour Sortir de la liste et avoir la main, des fois nous sommes obligé de taper :q



MINGW64:/c/Users/TEK-UP/desktop/projet-git

Créer un dépôt Git



```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~ (master)
$ cd desktop #on va se placer sur le bureau
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop (master)
  Nous allons créer un répertoire "projet-git" qui se trouve sur le bureau et qui contient deux fichiers texte vides
$ mkdir projet-git #créer un projet git sur le bureau
```

- Ce répertoire va nous servir de base pour les exemples qui vont suivre (ce sera le répertoire importé).

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop (master)
$ cd projet-git #on se place dans le dossier
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ touch fichier1.txt #créer un fichier texte dans le dossier
```

- ✓ La commande **cd** sert à de placer dans un répertoire.
Dès qu'on est sur le bureau, on utilise **mkdir** pour créer un répertoire vide qu'on appelle "projet-git".

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ ls #afficher le contenu du dossier
README.txt fichier1.txt
```

- ✓ On se place dans ce répertoire et on crée deux fichiers texte grâce à la commande Bash **touch**.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$
```

- ✓ On utilise enfin **ls** pour afficher le contenu du répertoire et s'assurer que tout a bien fonctionné.

Créer un dépôt Git

- Pour initialiser un dépôt Git, on utilise ensuite la commande `git init` comme ci-dessous.
- Cela crée un sous répertoire `.git` qui contient un ensemble de fichiers qui vont permettre à un dépôt Git de fonctionner.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git init #Initialiser un dépôt Git
Initialized empty Git repository in C:/Users/TEK-UP/Desktop/projet-git/.git/
```

- Lorsqu'on utilise `git init`, Git nous renvoie un message en nous informant que le dépôt Git a bien été initialisé et qu'il est vide. C'est tout à fait normal puisque nous n'avons encore versionné aucun fichier (nous n'avons ajouté aucun fichier du répertoire en base).

Créer un dépôt Git

- On peut utiliser ici la commande `git status` pour déterminer l'état des fichiers de notre répertoire. Cette commande est extrêmement utile.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.txt
    fichier1.txt

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

- Ici, `git status` nous informe que notre projet possède deux fichiers qui ne sont pas sous suivi de version (“untracked”) et qui sont les fichiers “README.txt” et “ficiher1.txt”.
- Il nous dit aussi qu’aucun fichier n’a été validé (“commit”) en base pour le moment ni ajouté pour validation.
- La commande `git statuts` nous informe également sur la branche sur laquelle on se trouve (“master” ici). Nous reparlerons des branches plus tard.

Créer un dépôt Git

- L'étape suivante va donc ici être d'indexer nos fichiers afin qu'ils puissent ensuite être validés, c'est-à-dire ajoutés en base et qu'on puisse ainsi avoir un premier historique de version.
- Pour indexer des fichiers, on utilise la commande **git add**. On peut lui passer un nom de fichier pour indexer le fichier en question, le nom d'un répertoire pour indexer tous les fichiers du répertoire d'un coup ou encore un “fileglob” pour ajouter tous les fichiers correspondant au schéma fourni.
- Les fileglobs utilisent les extension de chemin de fichier. Grossièrement, cela signifie que certains caractères comme ***** et **?** vont posséder une signification spéciale et nous permettre de créer des schémas de correspondances.
- Le caractère ***** par exemple correspond à n'importe quel caractère. Lorsque j'écris **git add *.txt**, je demande finalement à Git d'ajouter à l'index tous les fichiers du projet qui possèdent une extension **.txt**, quelque soit leur nom.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add *.txt #Indexer tous les fichiers texte
```

Créer un dépôt Git

- Si on relance une commande `git status`, on obtient les informations suivantes :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: README.txt
    new file: fichier1.txt
```

- `git status` nous dit qu'on a maintenant deux nouveaux fichiers ajoutés à l'index.
- La commande `git add` permet en fait de faire plusieurs choses : elle permet d'indexer des fichiers déjà sous suivi de version et de placer sous suivi des fichiers non suivi (en plus de les indexer).
- Ici, on est certains que nos deux nouveaux fichiers ont bien été ajoutés à l'index puisqu'ils apparaissent dans la section “changes to be committed” (“modifications à valider”).

Créer un dépôt Git

- Pour valider ces fichiers et les ajouter en base, nous allons maintenant utiliser la commande `git commit` comme cela :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m "Version initiale du projet" #valider les fichiers
[master (root-commit) 3119da5] Version initiale du projet
 2 files changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 README.txt
  create mode 100644 fichier1.txt
```

- On nous informe ici qu'on se situe sur la branche master, qu'il s'agit du premier commit (root-commit) et on nous donne sa somme de contrôle (3119da5) qui permet de l'identifier de manière unique.
- On nous dit également que deux fichiers ont été modifiés et que 0 lignes ont été ajoutées ou supprimées dans ces fichiers.
- Si on effectue à nouveau un `git status`, voici le message renvoyé :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

- Git nous informe désormais qu'il n'y a plus aucun fichier à vider, ce qui signifie que tous les fichiers sont sous suivi de version et sont enregistrés en base et qu'aucune modification n'a été apportée à ces fichiers depuis le dernier commit.

Récap

- A chaque fois qu'on souhaite enregistrer une modification de fichier ou un ajout de fichier dans le dépôt Git, on va devoir utiliser les commandes `git add` et `git commit`.
- N.B. Le commit (la validation / l'enregistrement en base de données) d'un fichier se basera sur l'état de ce fichier au moment du `git add`.
- Cela signifie que si vous effectuez une commande `git add` sur un fichier puis modifiez à nouveau le fichier puis effectuez un `git commit`, c'est le fichier dans son état au moment du dernier `git add` qui sera validé et les dernières modifications ne seront donc pas enregistrées dans le dépôt Git.
- Si vous souhaitez enregistrer toujours la dernière version d'un fichier, pensez donc bien toujours à effectuer un `git add` juste avant un `git commit`.
- Pour mettre en un coup les fichiers modifiés et déjà sous suivi dans la zone d'index puis pour les valider, vous pouvez également utiliser `git commit` avec une option `-a` comme ceci : `git commit -a -m`. Cela vous dispense d'avoir à taper `git add`.

Modifier un dépôt Git : Supprimer un fichier d'un projet et/ou l'exclure du suivie de la version Git

- Supprimer un fichier de votre dossier avec la commande Bash `rm` ne suffira pas pour que Git oublie ce fichier : il apparaitra dans chaque `git status` dans une section “changes not staged for commit” (modifications qui ne seront pas validées).
- Exemple : Ajouter un fichier `test.txt`, faire un commit puis le supprimer avec la commande `rm`.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    test.txt

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

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add test.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m "Ajout du fichier test.txt"
[master ebaa3b9] Ajout du fichier test.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.txt
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ rm test.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ 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:    test.txt

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

Comme vous pouvez le voir, Git continue de suivre le fichier et la suppression simple du fichier ne sera pas validée comme changement par Git.

Modifier un dépôt Git : Supprimer un fichier d'un projet et l'exclure du suivi de version Git

- Pour supprimer un fichier et l'exclure du suivi de version, nous allons utiliser la commande **git rm** (et non pas simplement une commande Bash **rm**).

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git rm test.txt #supprime le fichier du projet et du suivi de version
rm 'test.txt'

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    test.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m "Suppression de test.txt du projet et du suivi"
[master b8f666c] Suppression de test.txt du projet et du suivi
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 test.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Modifier un dépôt Git : Supprimer un fichier d'un projet et l'exclure du suivie de version Git

- Pour simplement exclure un fichier du suivi Git mais le conserver dans le projet, on va utiliser la même commande `git rm` mais avec cette fois-ci une option `--cached`.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git rm test.txt #supprime le fichier du projet et du suivi de version
rm 'test.txt'

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    test.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m "Suppression de test.txt du projet et du suivi"
[master b8f666c] Suppression de test.txt du projet et du suivi
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 test.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Modifier un dépôt Git : Supprimer un fichier d'un projet et/ou l'exclure du suivie de version Git

- Pour simplement exclure un fichier du suivi Git mais le conserver dans le projet, on va utiliser la même commande `git rm` mais avec cette fois-ci une option `--cached`.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ touch fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m 'Ajout du fichier fichier2.txt'
[master dac6945] Ajout du fichier fichier2.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fichier2.txt
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git rm --cached fichier2.txt
rm 'fichier2.txt'
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    fichier2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    fichier2.txt
```

Modifier un dépôt Git : Supprimer un fichier d'un projet et/ou l'exclure du suivie de version Git

- Pour simplement exclure un fichier du suivi Git mais le conserver dans le projet, on va utiliser la même commande `git rm` mais avec cette fois-ci une option `--cached`.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ touch fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m 'Ajout du fichier fichier2.txt'
[master dac6945] Ajout du fichier fichier2.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fichier2.txt
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git rm --cached fichier2.txt
rm 'fichier2.txt'
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:   fichier2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    fichier2.txt
```

Modifier un dépôt Git : Supprimer un fichier d'un projet ou l'exclure du suivie de version Git

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m 'Abandon du suivi de fichier2'
[master ffe4962] Abandon du suivi de fichier2
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 fichier2.txt
```

- Ici, le fichier a bien été exclu du suivi Git mais existe toujours dans notre projet. On va ensuite pouvoir modifier ce fichier (lui ajouter du texte par exemple) comme n'importe quel fichier et Git ne se préoccupera pas des modifications.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ echo 'Ajout du texte' >> fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ cat fichier2.txt
Ajout du texte

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status #pour vérifier que git ne suit plus le fichier2.txt
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    fichier2.txt

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

Récap: Supprimer un fichier d'un projet et/ou l'exclure du suivi de version Git

- La commande `rm` est utilisée pour supprimer un fichier de votre système de fichiers. Lorsque vous utilisez cette commande, le fichier est supprimé physiquement de votre disque dur, mais il n'est pas supprimé de l'historique de version de Git, car Git n'a pas été informé de cette suppression.
- La commande `git rm` est utilisée pour supprimer un fichier du suivi de version de Git. Cela signifie que le fichier est supprimé de votre répertoire de travail et de l'historique de Git. Lorsque vous utilisez cette commande, Git marque le fichier comme supprimé et enregistre cette modification dans son historique.
- La commande `git rm --cached` est utilisée pour supprimer un fichier du suivi de version de Git, sans supprimer le fichier réellement. Cela signifie que le fichier est toujours présent dans votre répertoire de travail, mais il ne sera plus suivi par Git. Cette commande est souvent utilisée pour supprimer un fichier du suivi de version lorsqu'un fichier a été ajouté par erreur à Git, ou pour retirer un fichier du suivi de version sans supprimer le fichier réel.
- En résumé, `git rm` est utilisée pour supprimer un fichier du suivi de version de Git et de votre répertoire de travail, `git rm --cached` est utilisée pour supprimer un fichier du suivi de version de Git sans supprimer le fichier réel, et `rm` est utilisée pour supprimer un fichier de votre système de fichiers sans le supprimer du suivi de version de Git.

Modifier un dépôt Git : Retirer un fichier de la zone d'index de Git

- Le contenu de la zone d'index est ce qui sera proposé lors du prochain commit. Imaginons qu'on ait ajouté un fichier à la zone d'index par erreur. Pour retirer un fichier de l'index, on peut utiliser `git reset HEAD nom-du-fichier`. A la différence de `git rm`, le fichier continuera d'être suivi par Git. Seulement, le fichier dans sa version actuelle va être retiré de l'index et ne fera donc pas partie du prochain commit.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add fichier2.txt
warning: in the working copy of 'fichier2.txt', LF will be replaced by CRLF the next time Git touches it

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -m 'Sauvegarde fichier2.txt'
[master 121a1f9] Sauvegarde fichier2.txt
 1 file changed, 1 insertion(+)
 create mode 100644 fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ echo 'Modification du fichier 2'>> fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ cat fichier2.txt
Ajout du texte
Modification du fichier 2

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add fichier2.txt
warning: in the working copy of 'fichier2.txt', LF will be replaced by CRLF the next time Git touches it

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   fichier2.txt
```

Ici, on ajoute à nouveau le fichier `fichier2.txt` à l'index et on le passe donc sous suivi de version. On valide cela avec `git commit` puis on modifie le contenu de notre fichier et on ajoute la dernière version de notre fichier à la zone d'index pour qu'il fasse partie du prochain commit.

Modifier un dépôt Git : Retirer un fichier de la zone d'index de Git

- Finalement, on change d'idée et on veut retirer cette version de la zone d'index. On fait cela avec `git reset HEAD fichier2.txt`.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git reset HEAD fichier2.txt #retirer le fichier de la zone d index
Unstaged changes after reset:
M      fichier2.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (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:   fichier2.txt

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

La commande `git reset HEAD nom-du-fichier`: Permet de retirer un fichier de la **staging area** (zone de préparation), qui est l'endroit où les fichiers sont préparés avant d'être ajoutés au dépôt Git lors d'un commit. Si un fichier a été ajouté à la **staging area** par erreur, `git reset HEAD nom-du-fichier` permet de le retirer de la **staging area** pour qu'il ne soit pas inclus dans le prochain commit.

Modifier undépôtGit: Empêcher l'indexation de certains fichiers dans Git Notion de fichier .gitignore

- Lorsqu'on dispose d'un projet et qu'on souhaite utiliser Git pour effectuer un suivi de version, il est courant qu'on souhaite exclure certains fichiers du suivi de version comme certains fichiers générés automatiquement, des fichiers de configuration, des fichiers sensibles, etc.
- On peut informer Git des fichiers qu'on ne souhaite pas indexer en créant un fichier **.gitignore** et en ajoutant les différents fichiers qu'on souhaite ignorer. Notez qu'on peut également mentionner des schémas de noms pour exclure tous les fichiers correspondant à ce schéma et qu'on peut même exclure le contenu entier d'un répertoire en écrivant le chemin du répertoire suivi d'un slash.
- Le fichier **.gitignore** est généralement placé à la racine du dépôt Git et doit être commité pour que les règles d'ignorance soient appliquées à tous les utilisateurs du dépôt. Vous pouvez également avoir plusieurs fichiers **.gitignore** dans différents sous-répertoires pour spécifier les règles d'ignorance pour des fichiers spécifiques à ces sous-répertoires.
- Le fichier **.gitignore** est utile pour éviter de suivre des fichiers temporaires, des fichiers de configuration, des fichiers de sortie de compilation, des fichiers de journalisation, des fichiers de sauvegarde et d'autres types de fichiers qui ne doivent pas être inclus dans le dépôt Git.

Comment créer un fichier **.gitignore**?

Modifier undépôtGit: Empêcher l'indexation de certains fichiers dans Git Notion de fichier .gitignore

1. Créer un fichier .gitignore :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ touch .gitignore
```

2. Configurer votre projet

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git config --global core.excludesfile ~/.gitignore_global
```

La commande "git config --global core.excludesfile~/ .gitignore" permet de définir un fichier global qui contient une liste de patterns (motifs) à exclure lorsqu'on effectue des opérations avec Git. Ces patterns peuvent être utilisés pour exclure des fichiers ou des répertoires spécifiques des commits, des pushes ou des pulls.

Le paramètre "--global" indique que la configuration s'appliquera à tous les projets Git sur la machine de l'utilisateur, pas seulement à un projet spécifique.

Par exemple, si vous utilisez souvent un éditeur de texte particulier qui crée des fichiers de sauvegarde avec l'extension .bak, vous pouvez ajouter l'entrée "*.bak" dans le fichier d'exclusion global pour empêcher Git de suivre ces fichiers dans tous les projets.

Modifier undépôtGit:Empêcher l'indexation de certains fichiers dansGit Notiondefichier.gitignore

.gitignore - Bloc-notes

Fichier Edition Format Affichage Aide
ignorer tous les fichiers log
*.log

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git add .gitignore
warning: in the working copy of '.gitignore', LF will be replaced by CRLF the next time Git touches it
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore
    modified:   fichier2.txt
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ touch test.log
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore
    modified:   fichier2.txt
```

Modifier un dépôt Git: Renommer un fichier dans Git

- On peut également renommer un fichier de notre projet depuis Git en utilisant cette fois-ci une commande `git mv ancien-nom-fichier nouveau-nom-fichier`.
- On peut par exemple renommer notre fichier “README.txt” en “LISEZMOI.txt” de la manière suivante :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ ls
README.txt  fichier1.txt  fichier2.txt  test.log  test2.log  test3.log

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git mv README.txt LISEZMOI.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ ls
LISEZMOI.txt  fichier1.txt  fichier2.txt  test.log  test2.log  test3.log
```

Modifier un dépôt Git: Renommer un fichier dans Git

Le fichier a bien été renommé dans notre répertoire et le changement est prêt à être validé dans le prochain commit.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed: README.txt -> LISEZMOI.txt
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git commit -a -m "renommage du fichier"
[master 3a78e39] renommage du fichier
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README.txt => LISEZMOI.txt (100%)
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Annuler des actions et consulter l'historique: Consulter l'historique des modifications Git

- La commande `git log`. Cette commande affiche la liste des commits réalisés du plus récent au plus ancien. Par défaut, chaque commit est affiché avec sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur, la date et le message du commit.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git log
commit 3a78e39a44dad3a47f2e1db40ab0913aa133f0f2 (HEAD -> master)
Author: leila bousbia<leila.bousbiaa@gmail.com>
Date:   Sun Apr 9 17:13:29 2023 +0200

    renommage du fichier

commit 9c58c7f98fc7f044277552f7c9e230073c52030a
Author: leila bousbia<leila.bousbiaa@gmail.com>
Date:   Sun Apr 9 17:06:08 2023 +0200

    Ajout du fichier gitignore
```

- La commande `git log` supporte également de nombreuses options. Certaines vont pouvoir être très utiles comme par exemple les options `-p`, `--pretty`, `--since` ou `--author`.

Annuler des actions et consulter l'historique: Consulter l'historique des modifications Git

- Utiliser `git log -p` permet d'afficher explicitement les différences introduites entre chaque validation.

```
commit 3a78e39a44dad3a47f2e1db40ab0913aa133f0f2 (HEAD -> master)
Author: Sawssen JALEL <sawssen.jalel@tek-up.tn>
Date:   Sun Apr 9 17:13:29 2023 +0200

    renommage du fichier

diff --git a/README.txt b/LISEZMOI.txt
similarity index 100%
rename from README.txt
rename to LISEZMOI.txt
```

- Utiliser `git log -n` permet d'afficher explicitement les n dernières validations.
- Utiliser `git log --oneline` permet d'afficher explicitement chaque commit en une seule ligne.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
$ git log --oneline
3a78e39 (HEAD -> master) renommage du fichier
9c58c7f Ajout du fichier gitignore
121a1f9 Sauvegarde fichier2.txt
ffe4962 Abandon du suivi de fichier2
dac6945 Ajout du fichier fichier2.txt
b8f666c Suppression de test.txt du projet et du suivi
ebaa3b9 Ajout du fichier test.txt
86094b4 version initiale du projet
```

Annuler des actions et consulter l'historique: Consulter l'historique des modifications Git

- Pour annuler une validation (un commit), notamment lorsque la validation a été faite en oubliant des fichiers ou sur les mauvaises versions de fichiers, on utilise la commande `git commit` avec l'option `--amend`. Cela va pousser un nouveau commit qui va remplacer le précédent en l'écrasant.

1
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
\$ git log --oneline
fe974a2 (**HEAD -> master**) Ajout et modification fichier3
3a78e39 renommage du fichier
9c58c7f Ajout du fichier gitignore
121a1f9 Sauvegarde fichier2.txt
ffe4962 Abandon du suivi de fichier2
dac6945 Ajout du fichier fichier2.txt
b8f666c Suppression de test.txt du projet et du suivi
ebaa3b9 Ajout du fichier test.txt
86094b4 version initiale du projet

2
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
\$ git mv fichier3.txt fichierTestAnuulationCommit.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
\$ git status
On branch master
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
renamed: fichier3.txt -> fichierTestAnuulationCommit.txt

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: fichierTestAnuulationCommit.txt

3
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
\$ git commit --amend -m "Test annulation commit"
[master d58d04d] Test annulation commit
Date: Mon Apr 10 12:32:19 2023 +0200
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 fichierTestAnuulationCommit.txt

4
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/projet-git (master)
\$ git log --oneline
d58d04d (**HEAD -> master**) Test annulation commit
3a78e39 renommage du fichier
9c58c7f Ajout du fichier gitignore
121a1f9 Sauvegarde fichier2.txt
ffe4962 Abandon du suivi de fichier2
dac6945 Ajout du fichier fichier2.txt
b8f666c Suppression de test.txt du projet et du suivi
ebaa3b9 Ajout du fichier test.txt
86094b4 version initiale du projet

Création d'un compte github

Etapes de création

GitHub est un service Web d'hébergement de dépôts distants, utilisant git.

GitHub comporte des fonctionnalités supplémentaires destinées à la collaboration, telles que le suivi des bugs, les demandes d'ajout de fonctionnalités, ou la gestion de tâches.

A noter qu'il existe d'autres sites Web d'hébergement basés sur git, tels que GitLab, ou BitBucket.

1. Allez sur <https://github.com/> et suivez les instructions pour la création et l'activation de votre compte. Sur un terminal,
2. spécifiez l'adresse mail avec laquelle vous allez effectuer vos commits : `git config -- global user . email " votre . email@example .com "`

Search or jump to... / Pull requests Issues Codespaces Marketplace Explore

leilabousbia / tp3 Public Pin ⚠️ Couldn't load subscription status. Retry Fork

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

New repository Import repository New codespace New gist New organization

Set up GitHub Copilot Use GitHub's AI pair programmer to autocomplete suggestions as you code.

Invite collaborators Find people using their GitHub username or email address.

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

Set up in Desktop or HTTPS SSH git@github.com:leilabousbia/tp3.git

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

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

```
echo "# tp3" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git push -u origin main
```

https://github.com/new add origin git@github.com:leilabousbia/tp3.git

Etapes de création

1. Créez tout d'abord un compte sur le site GitHub sur ce lien <https://github.com/login>.
2. Créez un repository «project tic-1 E» en le configurant public.

Public?

Tout le monde pourra lire le code présent:

Nous pouvons par suite choisir d' initialiser le dépôt avec plusieurs fichiers :

- Le fichier README.md est l'équivalent du « lisez-moi », permettant de présenter le projet et d'y fournir des instructions pour installer/exécuter.
- Le fichier .gitignore permet de spécifier les dossiers et fichiers à ne pas considérer dans le dépôt Git. Par exemple, on ne souhaite pas mettre dans le dépôt Git un fichier qui contiendrait des mots de passe.
- Un fichier .gitignore est un fichier texte placé dans le dépôt git qui indique à git de ne pas suivre certains fichiers et dossiers que nous ne souhaitons pas télécharger dans notre dépôt maître. Il a de nombreuses utilisations et nous devrons presque toujours le configurer si nous configurons un nouveau dépôt.
- Le fichier LICENSE est utile dans le cas des dépôts publics avec des licences open-source.

Une fois sélectionné, nous pouvons créer le dépôt.

Create a new repository

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

Required fields are marked with an asterisk (*).

Owner *



Repository name *

projet tic-1q

⚠ Your new repository will be created as projet-tic-1q.

Great repository names are short and memorable. Need inspiration? How about [congenial-rotary-phone](#) ?

Description (optional)

Public

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

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file

This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

 .gitignore template: None ▾

Choose which files not to track from a list of templates. Learn more about ignoring files.

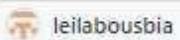
Choose a license

 License: None ▾

A license tells others what they can and can't do with your code. Learn more about licenses.

ⓘ You are creating a public repository in your personal account.

Owner *



leilabousbia

Repository name *

projet tic-1q

⚠ Your new repository will be created as projet-tic-1q.

Great repository names are short and memorable. Need inspiration? How about [congenial-rotary-phone](#) ?

Description (optional)



Public

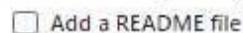
Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:



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

Add .gitignore



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

Choose a license



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

ⓘ You are creating a public repository in your personal account.

Create repository

projet-tic-1q Public

 Pin  Unwatch 1  Fork 0  Star 0



Set up GitHub Copilot

Use GitHub's AI pair programmer to autocomplete suggestions as you code.



Invite collaborators

Find people using their GitHub username or email address.

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

 Set up in Desktop

or

 HTTPS

 SSH

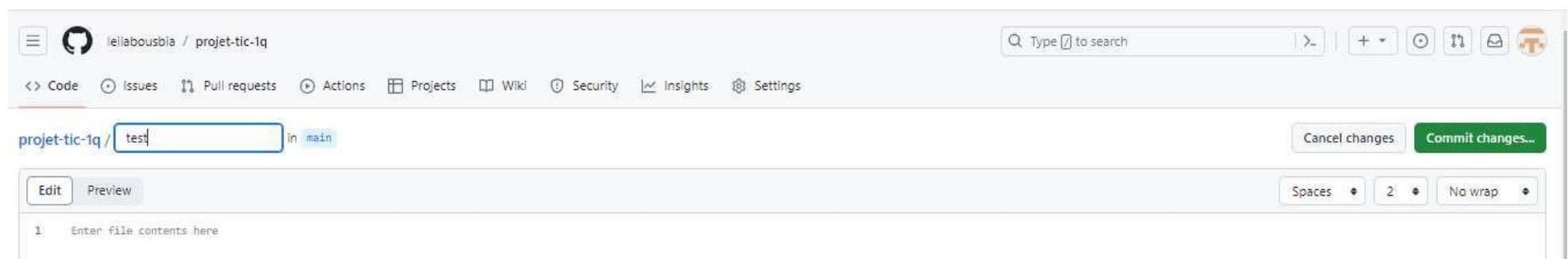
git@github.com:leilabousbia/projet-tic-1q.git 

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

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

```
echo "# projet-tic-1q" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:leilabousbia/projet-tic-1q.git
git push -u origin main 
```

Création d'un fichier dans un projet



Création d'un fichier dans un projet

The screenshot shows a GitHub repository page for 'projet-tic-1q'. The repository is owned by 'leilabousbia' and is public. The main interface includes a search bar at the top right, navigation links like 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. Below the header, the repository name 'projet-tic-1q' is displayed along with its status as 'Public'. A summary section shows 'main' branch, '1 branch', '0 tags', and a recent commit by 'leilabousbia' titled 'Create test' made 'now'. A prominent button 'Add a README' is visible. To the right, sections for 'About', 'Activity', 'Stars', 'Watching', and 'Forks' are shown, all currently at zero. Below these are sections for 'Releases' and 'Packages', both also showing zero entries.

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

projet-tic-1q Public

main 1 branch 0 tags

leilabousbia Create test 56b45c2 · in 30 seconds 1 commit

test Create test now

Add a README

About

No description, website, or topics provided.

Activity

0 stars

2 watching

0 forks

Releases

No releases published

Create a new release

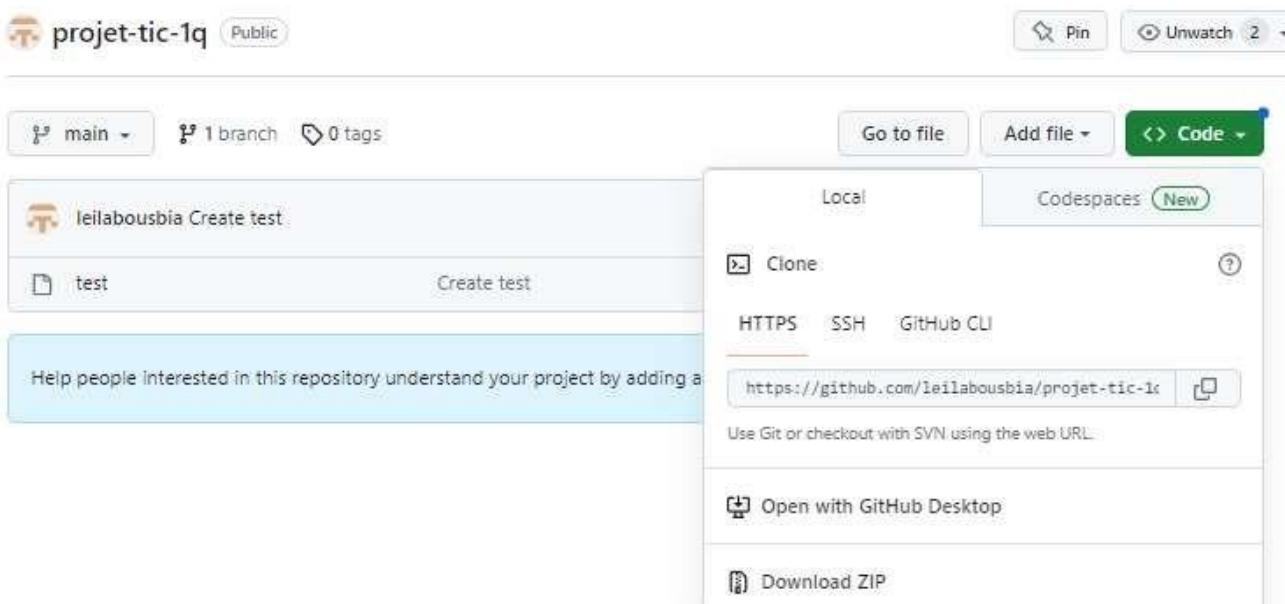
Packages

No packages published

Publish your first package

Récupération en local

Maintenant, nous allons récupérer ce dépôt en local. Pour cela, nous pouvons cliquer sur le bouton Code puis copier la commande git à exécuter dans un terminal.



Récupération en local

```
TEK-UP@DESKTOP-1EKN551 MINGW64 ~ (master)
$ git clone https://github.com/leilabousbia/projet-tic-1q.git
Cloning into 'projet-tic-1q'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.

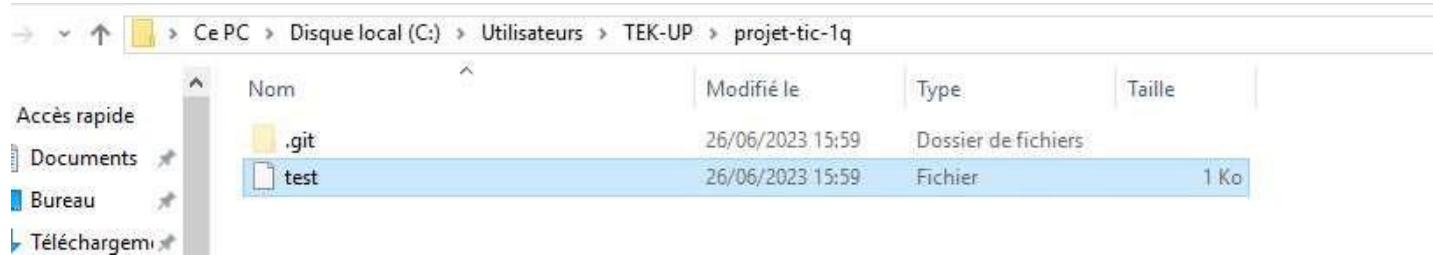
TEK-UP@DESKTOP-1EKN551 MINGW64 ~ (master)
$
```

Nous avons bien récupéré tous nos fichiers. À noter qu'il y a un dossier nommé .git : ce dernier contient tous les fichiers nécessaires au dépôt, aussi bien des copies de code que des journaux ou des informations diverses.

Récupération en local

Ce PC > Disque local (C:) > Utilisateurs > TEK-UP >				
	Nom	Modifié le	Type	Taille
★ Accès rapide	.idlerc	09/09/2020 18:44	Dossier de fichiers	
Documents	.ssh	13/04/2023 03:25	Dossier de fichiers	
Bureau	AppData	23/09/2019 12:27	Dossier de fichiers	
↓ Téléchargem	Bureau	26/06/2023 13:11	Dossier de fichiers	
Images	Cisco Packet Tracer 7.2.1	14/04/2020 08:45	Dossier de fichiers	
Cours et TD S	Cisco Packet Tracer 8.2.0	07/04/2023 23:28	Dossier de fichiers	
E:\	Cisco Packet Tracer 8.2.1	23/05/2023 11:54	Dossier de fichiers	
fev 2022	Contacts	26/09/2019 16:56	Dossier de fichiers	
juillet 2022	Documents	09/06/2023 09:12	Dossier de fichiers	
Nouveau dossier	Favoris	26/09/2019 16:56	Dossier de fichiers	
Nouveau dossier	git-merge-test	08/05/2023 21:18	Dossier de fichiers	
OneDrive - Perso	Images	08/06/2022 17:29	Dossier de fichiers	
Ce PC	Liens	26/09/2019 16:56	Dossier de fichiers	
Bureau	Musique	16/02/2020 10:49	Dossier de fichiers	
Documents	newproject	12/04/2023 15:50	Dossier de fichiers	
Images	Objets 3D	02/05/2020 00:52	Dossier de fichiers	
Musique	OneDrive	03/08/2022 17:05	Dossier de fichiers	
↓ Téléchargement	Parties enregistrées	26/09/2019 16:56	Dossier de fichiers	
Vidéos	projet-tic-1q	26/06/2023 15:59	Dossier de fichiers	
	Recherches	06/10/2021 12:26	Dossier de fichiers	
	repo56	14/04/2023 14:22	Dossier de fichiers	
	↓ Téléchargements	26/06/2023 09:40	Dossier de fichiers	

Récupération en local



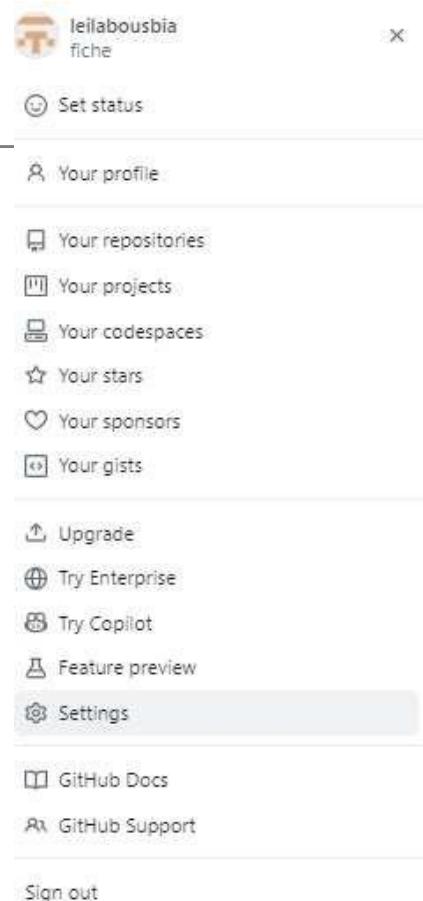


Créer dépôt GitHub (5)

Récap :

- ❖ Pour démarrer un projet, vous devez créer votre compte GitHub.
- ❖ Vous pouvez suivre vos différents projets facilement grâce au tableau de bord.
- ❖ Pour mettre votre projet sur GitHub, vous devez créer un repository.

Création d'un jeton d'accès personnel



Création d'un personal access token (classic)

1. Dans le coin supérieur droit d'une page, cliquez sur votre photo de profil, puis sur settings.



fiche (leilabousbia)
Your personal account

Public profile

Account

Appearance

Accessibility

Notifications

Access

Billing and plans

Emails

Password and authentication

Sessions

SSH and GPG keys

Organizations

Moderation

Code, planning, and automation

Repositories

Codespaces

Packages

Copilot

Pages

Création d'un jeton d'accès personnel

2. Dans la barre latérale gauche, cliquez sur developer setting.
3. Dans la barre latérale gauche, sous Personal access token , cliquez sur Jetons (classique) . 1. Sélectionnez Générer un nouveau jeton,

Saved replies

Security

Code security and analysis

Integrations

Applications

Scheduled reminders

Archives

Security log

Sponsorship log

Developer settings

The screenshot shows the GitHub developer settings interface. On the left, there's a sidebar with various options like GitHub Apps, OAuth Apps, Personal access tokens, and Fine-grained tokens. A green oval highlights the 'Personal access tokens' option. On the right, under the 'Personal access tokens' heading, there's a red box around the 'Tokens (classic)' button, which is labeled as a beta feature.

Création d'un jeton d'accès personnel

Cliquez sur Générer un nouveau jeton (classique) .

The screenshot shows the GitHub 'Personal access tokens (classic)' page. On the left, there's a sidebar with 'GitHub Apps', 'OAuth Apps', and 'Personal access tokens' selected. Under 'Personal access tokens', there are two tabs: 'Fine-grained tokens' (Beta) and 'Tokens (classic)', with 'Tokens (classic)' currently active. The main area displays four tokens:

Name	Scope	Last Used	Status	Action
key2023 — public access	public access	Never used	Never used	Delete
token1 — public access	public access	Never used	Never used	Delete
leilabousbiaa — repo:status, repo_deployment	repo:status, repo_deployment	Never used	Never used	Delete
Firstrtoken — repo:status, repo_deployment	repo:status, repo_deployment	Never used	Never used	Delete

Below the tokens, a note states: 'Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.'

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

Création d'un jeton d'accès personnel

4. Donnez à votre jeton un nom descriptif.
 5. Attribuer à votre jeton un délai d'expiration.
 6. Sélectionnez les étendues à attribuer à ce jeton.
- Pour utiliser votre jeton afin d'accéder aux dépôts à partir de la ligne de commande, sélectionnez dépôt.

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API](#) over Basic Authentication.

Note

key-tic-q2023

What's this token for?

Expiration *

30 days  The token will expire on Sun, Jul 30 2023

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input type="checkbox"/> admin:org	Full control of orgs and teams; read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership; read and write org projects
<input type="checkbox"/> read:org	Read org and team membership; read org projects
<input type="checkbox"/> manage_runners:org	Manage org runners and runner groups
<input type="checkbox"/> admin:public_key	Full control of user public keys

Création d'un jeton d'accès personnel

Après avoir générer token vous pouvez accéder à la page ci dessous



GitHub Apps

OAuth Apps

Personal access tokens

Fine-grained tokens Beta

Tokens (classic)

Personal access tokens (classic)

Generate new token ▾ Revoke all

Tokens you have generated that can be used to access the GitHub API.

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp_IQXYnSLVAujBst14hwZL6a60hRX7mA4P2Vm0 Copy	Delete
key2023 — public access	Never used
Expires on Wed, Jul 26 2023.	
token1 — public access	Never used
Expires on Wed, May 24 2023.	
leilabousbiaa — repo:status, repo_deployment	Never used
Expired on Sat, May 13 2023.	
Firstrtoken — repo:status, repo_deployment	Never used
Expired on Fri, May 12 2023.	

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

<input type="checkbox"/> delete_repo	Delete repositories	1
<input type="checkbox"/> write:discussion	Read and write team discussions	
<input type="checkbox"/> read:discussion	Read team discussions	
<input type="checkbox"/> admin:enterprise	Full control of enterprises	
<input type="checkbox"/> manage_runners:enterprise	Manage enterprise runners and runner groups	
<input type="checkbox"/> manage_billing:enterprise	Read and write enterprise billing data	
<input type="checkbox"/> read:enterprise	Read enterprise profile data	
<input type="checkbox"/> audit_log	Full control of audit log	
<input type="checkbox"/> read:audit_log	Read access of audit log	
<input type="checkbox"/> codespace	Full control of codespaces	
<input type="checkbox"/> codespace/secrets	Ability to create, read, update, and delete codespace secrets	
<input type="checkbox"/> project	Full control of projects	
<input type="checkbox"/> read:project	Read access of projects	
<input type="checkbox"/> admin:gpg_key	Full control of public user GPG keys	
<input type="checkbox"/> write:gpg_key	Write public user GPG keys	
<input type="checkbox"/> read:gpg_key	Read public user GPG Keys	
<input type="checkbox"/> admin:ssh_signing_key	Full control of public user SSH signing keys	
<input type="checkbox"/> write:ssh_signing_key	Write public user SSH signing keys	
<input type="checkbox"/> read:ssh_signing_key	Read public user SSH signing keys	

Création d'un jeton d'accès personnel

Après avoir générer token vous pouvez accéder à la page ci dessous



Personal access tokens (classic)

Generate new token ▾

Revoke all

Tokens you have generated that can be used to access the GitHub API.

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp_IQxYnSLVAujBst14hwZL6a60hRX7mA4P2Vm0

<input type="checkbox"/> delete_repo	Delete repositories	1
<input type="checkbox"/> write:discussion	Read and write team discussions	
<input type="checkbox"/> read:discussion	Read team discussions	
<input type="checkbox"/> admin:enterprise	Full control of enterprises	
<input type="checkbox"/> manage_runners:enterprise	Manage enterprise runners and runner groups	
<input type="checkbox"/> manage_billing:enterprise	Read and write enterprise billing data	
<input type="checkbox"/> read:enterprise	Read enterprise profile data	
<input type="checkbox"/> audit_log	Full control of audit log	
<input type="checkbox"/> read:audit_log	Read access of audit log	
<input type="checkbox"/> codespace	Full control of codespaces	
<input type="checkbox"/> codespace/secrets	Ability to create, read, update, and delete codespace secrets	
<input type="checkbox"/> project	Full control of projects	
<input type="checkbox"/> read:project	Read access of projects	
<input type="checkbox"/> admin:gpg_key	Full control of public user GPG keys	
<input type="checkbox"/> write:gpg_key	Write public user GPG keys	
<input type="checkbox"/> read:gpg_key	Read public user GPG keys	
<input type="checkbox"/> admin:ssh_signing_key	Full control of public user SSH signing keys	
<input type="checkbox"/> write:ssh_signing_key	Write public user SSH signing keys	
<input type="checkbox"/> read:ssh_signing_key	Read public user SSH signing keys	

Copiez et sauvez
garder votre
personnel token
(bloc notes)





Authentification et Connexion avec SSH (1)

Cependant, si vous avez configuré Git pour utiliser une clé SSH pour l'authentification, il est possible que Git n'ait pas besoin de vous demander de vous authentifier pour utiliser la commande git pull.

Dans ce cas, Git utilisera la clé SSH pour s'authentifier automatiquement lors de la récupération des modifications à partir du dépôt distant



Authentification et Connexion avec SSH (2)

- ❖ L'authentification par échange de clés SSH fonctionne en plaçant une **clé publique** sur l'ordinateur distant et en utilisant une **clé privée** depuis son ordinateur.
- ❖ Ces deux clés (publique et privée) sont liées l'une à l'autre.
- ❖ C'est seulement en présentant la clé privée à la clé publique qu'il est possible se connecter.
- ❖ Chaque clé se présente sous la forme d'une longue chaîne de caractères enregistrée dans un fichier.
- ❖ Pour plus de sécurité, vous pouvez également protéger la clé privée avec une phrase secrète.
Autrement dit, pour pouvoir utiliser la clé privée, il faudra saisir un mot de passe, ce qui renforce encore davantage la sécurité.
- ❖ Les clés SSH peuvent être créées avec différents algorithmes de chiffrement :
Dans notre cas, nous allons utiliser :
Ed25519 : l'algorithme à privilégier aujourd'hui.



Génération des clés SSH (1)

- ❖ Générez une paire de clés SSH avec la commande **ssh-keygen** en spécifiant l'algorithme de chiffrement désiré. Avec Ed25519 :

```
$ ssh-keygen -t ed25519
```

- ❖ Laissez l'emplacement par défaut en appuyant sur Entrée.

```
Generating public/private ed25519 key pair. Enter file in  
which to save the key (C:\Users\Pierre\.ssh\id_ed25519):
```

- ❖ Entrez une phrase secrète pour protéger votre clé privée :

```
Enter passphrase (empty for no passphrase): Enter same  
passphrase again:
```



Génération des clés SSH (2)

- ❖ Voilà, votre paire de clés SSH a bien été générée !

```
Your identification has been saved in C:\Users\Pierre\.ssh\id_ed25519. Your public key has
been saved in C:\Users\Pierre\.ssh\id\_ed25519.pub. The key fingerprint is:
SHA256:4Yjb63lZzyRw+ADKaZ6nwZDA7jBrtoVR4mkgXRGwN0 pierre@DESKTOP-VJAT016 The key's
randomart image is:
+--[ED25519 256]--+
| ..+=. . |
| o+o .. E |
| o..o.o ... |
| o.o+=o ++.. |
| oo.=o.+ S= |
| .+ .=+. + . |
| o o o+o o = |
| .o .. oo o |
| + .+. |
+-----[SHA256]-----
```



Génération des clés SSH (3)

- ❖ Vos clés SSH se trouvent dans le répertoire `~/.ssh/`.

```
$ cd <lien de ssh>
$ ls
id_ed25519      id_ed25519.pub

$ cat id_ed25519
$ cat id_ed25519.pub
```

- ➔ La paire de clés SSH (**publique et privée**) a bien été créée.
- ➔ Prochaine étape : copier la clé publique sur le serveur distant.

Copier la clé publique SSH sur le serveur distant (1)



- ❖ Pour copier une clé **publique** sur le compte d'un utilisateur distant, suivez ces instructions :
- ➔ Open the github settings > ssh and gpg keys
- ❖ Sélectionnez un nom pour votre clé **publique** et collez la clé déjà copiée.

A screenshot of the GitHub user settings interface. The left sidebar shows the user's profile picture and name 'fiche (leilabousbia) Your personal account'. The 'SSH and GPG keys' option is highlighted in the sidebar. The main area is titled 'SSH keys / Add new'. It has fields for 'Title' (empty), 'Key type' (set to 'Authentication Key'), and a large text area for the 'Key' containing the copied SSH public key. A green 'Add SSH key' button is at the bottom.

Copier la clé publique SSH sur le serveur distant (2)



- ❖ On tape la commande :

```
$ ssh -T git@github.com
TEK-UP@DESKTOP-1EKN551 MINGW64 ~/ssh (master)
$ ssh -T git@github.com
Hi leilabousbia! You've successfully authenticated, but GitHub does not provide
shell access.
```



- ❖ Donc, on peut envoyer et cloner nos fichiers vers ou depuis le dépôt distant en toute sécurité.
(création d'un dossier .git, cd ./ dossier, touch, git add, git commit -m ...)



L'authentification avec token :Récap

-
- Est une méthode d'authentification que vous pouvez utiliser pour vous connecter à GitHub.
 - Un token d'accès est un jeton unique qui est généré par GitHub et qui permet à un utilisateur d'accéder à des ressources spécifiques sur GitHub, telles que des dépôts, des problèmes, des demandes de fusion, etc.
 - Le token d'accès est utilisé pour identifier l'utilisateur et pour s'assurer que l'utilisateur a les autorisations nécessaires pour accéder aux ressources demandées.
 - Les tokens d'accès peuvent être utilisés pour accéder à GitHub via l'API REST de GitHub, les clients Git, les navigateurs web, etc.
 - Les tokens d'accès peuvent être créés et gérés dans les paramètres de sécurité de votre compte GitHub.

L'authentification par Token: Avantages & Inconvénients

Avantages de l'authentification par token :

Les tokens d'accès peuvent être limités à des autorisations spécifiques, ce qui permet un accès granulaire à GitHub.

Les tokens peuvent être facilement révoqués ou régénérés si nécessaire.

L'utilisation de tokens permet de garder vos informations d'identification GitHub en sécurité et de réduire le risque de fuite de vos informations de connexion.

Les tokens peuvent être utilisés avec des outils tiers pour automatiser des tâches liées à GitHub.

Inconvénients de l'authentification par token :

Les tokens peuvent être plus difficiles à configurer que SSH, surtout si vous utilisez des outils tiers qui nécessitent une configuration supplémentaire.

Si un token d'accès est compromis, un attaquant peut accéder à toutes les ressources pour lesquelles le token a été autorisé.

L'authentification avec SSH : Récap



-
- SSH est un protocole de communication sécurisé qui permet à un utilisateur de se connecter à un serveur distant de manière sécurisée.
 - En utilisant SSH, vous pouvez vous connecter à GitHub via une connexion cryptée et authentifier votre identité en utilisant une paire de clés SSH.
 - Pour utiliser SSH avec GitHub, vous devez d'abord générer une paire de clés SSH sur votre ordinateur, puis ajouter la clé publique à votre compte GitHub. Une fois cela fait, vous pouvez utiliser la clé privée pour vous connecter à GitHub via SSH à partir de votre ordinateur.

L'authentification avec SSH : Avantages & Inconvénients

Avantages de l'authentification SSH :

L'authentification SSH est souvent plus facile à configurer que l'authentification par token, surtout si vous avez déjà configuré une paire de clés SSH pour d'autres serveurs.

L'utilisation de clés SSH peut améliorer la sécurité en réduisant le risque de fuite d'informations d'identification.

Les connexions SSH sont cryptées de bout en bout, offrant une sécurité accrue.

Inconvénients de l'authentification SSH :

L'ajout de votre clé publique SSH à votre compte GitHub nécessite une étape supplémentaire de configuration, ce qui peut rendre le processus un peu plus difficile pour les débutants.

Les clés SSH sont liées à un seul ordinateur, ce qui peut rendre la connexion à GitHub depuis d'autres ordinateurs plus difficile. Vous devrez copier votre clé privée sur chaque ordinateur que vous utilisez pour vous connecter à GitHub via SSH, ce qui peut augmenter le risque de compromission de votre clé privée.



L'authentification par Token et SSH:Récap

En résumé, l'authentification par token est un moyen d'accéder à des ressources spécifiques sur GitHub en utilisant un jeton unique, tandis que l'authentification SSH permet une connexion sécurisée à GitHub via une clé privée publique. Les deux méthodes offrent des niveaux de sécurité similaires, mais chacune a ses propres avantages et inconvénients, en fonction de votre cas d'utilisation spécifique.



Récap De GitHub à Git et inversement

- ❖ GitHub permet de contribuer simplement à des projets.
- ❖ La plupart des gens qui utilisent GitHub vont cependant souvent préférer travailler hors ligne (en local; sur leur machine) plutôt que de devoir être constamment connecté à GitHub et de devoir passer par cette plateforme pour effectuer toutes les opérations.
- ❖ Pour travailler localement, il suffit de **cloner** le projet. Les différentes modifications apportées sur le projet seront effectuées sur la machine.
- ❖ Pour synchroniser les modifications faites depuis la machine avec le dépôt distant (dépôt GitHub), il suffit de faire un **git commit** depuis le dépôt local et de **push** (envoyer) les modifications sur le dépôt GitHub à l'aide de la commande **git push**.
- ❖ Taper **git push origin master** par exemple revient à envoyer les modifications situées dans ma branche master vers origin.
- ❖ Pour récupérer en local les dernières modifications du dépôt GitHub, on va utiliser la commande **git pull**.



L'authentification avec SSH :Récap

- SSH est un protocole de communication sécurisé qui permet à un utilisateur de se connecter à un serveur distant de manière sécurisée.
- En utilisant SSH, vous pouvez vous connecter à GitHub via une connexion cryptée et authentifier votre identité en utilisant une paire de clés SSH.
- Pour utiliser SSH avec GitHub, vous devez d'abord générer une paire de clés SSH sur votre ordinateur, puis ajouter la clé publique à votre compte GitHub. Une fois cela fait, vous pouvez utiliser la clé privée pour vous connecter à GitHub via SSH à partir de votre ordinateur.



Chapitre 2: **Git : Revenir en arrière (Rollback)**

Revenir en arrière (Rollback)

En Git, le "rollback" est une opération qui permet de revenir en arrière dans l'historique des modifications d'un dépôt Git. Il est également appelé "revert" ou "annulation". Lorsqu'un commit a été effectué dans un dépôt Git, il est possible de revenir en arrière en créant un nouveau commit qui annule les modifications apportées par le commit précédent. Cette opération crée un nouvel état dans l'historique du dépôt, qui représente l'état avant le commit annulé.

- L'un des principaux intérêts d'utiliser un logiciel de gestion de vision est de pouvoir travailler en "roll back", c'est-à-dire de pouvoir revenir à un état antérieur enregistré d'un projet.
- Après avoir utiliser un commit, nous pouvons continuer à travailler sur nos fichiers et à les modifier.
 - ➔ Parfois, certaines modifications ne vont pas apporter les comportements espérés et on voudra revenir à l'état du fichier du dernier instantané Git (c'est-à-dire au dernier état enregistré).
 - ➔ On va pouvoir faire ceci avec la commande générale `git checkout -- nom-du-fichier` ou la nouvelle commande spécialisée `git restore nom-du-fichier`.

Plan



1. Parlons de HEAD
2. Revenir en arrière
3. Correction des commits
4. Revenir en arrière (2) avec revert
5. TP 2



Parlons de HEAD

Rapport entre répertoire de travail et commits

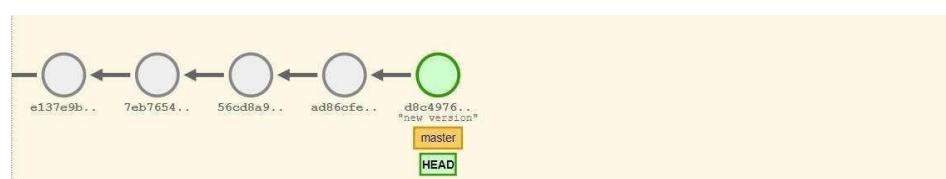
- Git considère votre répertoire de travail comme étant en fait composé de :
 - ✓ l'ensemble des fichiers *suivis*, **dans leur version correspondant à UN SNAPSHOT COURANT** (un commit).
 - ✓ **PLUS** les modifications existant sur ces fichiers par rapport à cette version spécifique.
 - ✓ **PLUS** des fichiers non suivis (pour lesquels on n'a pas effectué de `git add`)
- **HEAD** : une référence (c'est à dire, un pointeur vers un commit) pour désigner le '**SNAPSHOT COURANT**'.
- **master** : une référence qui pointe vers la tête de l'historique.

Parlons de HEAD

Rapport entre répertoire de travail et commits



- La commande `git commit` effectue deux choses :
 1. Elle enregistre un *snapshot* ayant un lien de parenté avec le commit référencé par HEAD.
 2. Elle déplace la référence HEAD et la référence master sur ce nouveau commit.





Parlons de HEAD

Manipulation de la référence HEAD

- `$ git checkout 56cd8a9` : la référence HEAD se déplace sur le commit 56cd8a9



- `git checkout HEAD^` : la référence HEAD se déplace d'un cran vers le gauche (HEAD[^] désigne le commit précédent HEAD)



- `git checkout master` : la référence HEAD se déplace vers la référence master, c'est-à-dire la tête de l'historique



La commande **git checkout** va spécifiquement déplacer le HEAD vers un commit que vous lui précisez.

En règle générale, elle ne peut être appliquée que si il n'y a pas de modifications en cours dans vos fichiers.



Parlons de HEAD

Influence sur le répertoire de travail

- Rappel : votre répertoire de travail est constitué des fichiers suivis *dans leur version correspondant au commit référencé par 'HEAD'*
- Par conséquent, `git checkout 56cd8a9`, par exemple, va déplacer 'HEAD' sur le commit 56cd8a9, puis va faire en sorte que votre répertoire de travail soit constitué des fichiers *dans leur version correspondant au nouveau 'HEAD'*.
- Les fichiers sont donc modifiés, ajoutés, ou supprimés, pour que votre répertoire de travail soit dans l'état qui était le sien au moment du commit 56cd8a9...



Revenir en arrière (1)

- 1. Supposons qu'on vient de supprimer un fichier et de Commiter cette suppression :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git commit -m "Suppression ProgPrincipal"
[master 888c1a4] Suppression ProgPrincipal
 1 file changed, 4 deletions(-)
 delete mode 100644 ProgPrincipal.py

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git log --oneline
888c1a4 (HEAD -> master) Suppression ProgPrincipal
2c9f609 1ere MAJ
03a51bd Ajout prog prncipal
427332e version 1
93393c9 version 0
```

- 2. On se rend compte qu'on a fait une mauvaise décision et on veut revenir à l'état précédent la suppression de ce fichier → sol : utiliser la commande git checkout id_commit_précédent (2c9f609)

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git checkout 2c9f609
Note: switching to '2c9f609'.
```



```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback ((2c9f609...))
$ ls
ProgPrincipal.py appFichierTicF.py file.txt notesModule.txt sms.txt.txt
```



Revenir en arrière (1)

- Si vous affichez l'historique, vous pouvez constater que tous les commits suivants ont *disparu* (tout du moins en apparence). Pour les *retrouver*, il faut replacer 'HEAD' sur la tête de l'historique enregistré dans le dépôt :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback ((2c9f609...))
$ git log --oneline
2c9f609 (HEAD) 1ere MAJ
03a51bd Ajout prog prncipal
427332e version 1
93393c9 version 0

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback ((2c9f609...))
$ git checkout master
Previous HEAD position was 2c9f609 1ere MAJ
Switched to branch 'master'

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git log --oneline
888c1a4 (HEAD -> master) Suppression ProgPrincipal
2c9f609 1ere MAJ
03a51bd Ajout prog prncipal
427332e version 1
93393c9 version 0
```

- À retenir : la commande git checkout id_commit permet de revenir en arrière mais sans pouvoir modifier.
- Pour récupérer le fichier ProgPrincipal.py, on doit taper :

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ ls
appFichierTicF.py file.txt notesModule.txt sms.txt.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git checkout 2c9f609 ProgPrincipal.py
Updated 1 path from a0027ec

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ ls
ProgPrincipal.py appFichierTicF.py file.txt notesModule.txt sms.txt.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ProgPrincipal.py

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git commit -m "restauration ProgPrincipal.py"
[master 8b63f89] restauration ProgPrincipal.py
 1 file changed, 4 insertions(+)
 create mode 100644 ProgPrincipal.py

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git log --oneline
8b63f89 (HEAD -> master) restauration ProgPrincipal.py
888c1a4 Suppression ProgPrincipal
2c9f609 1ere MAJ
03a51bd Ajout prog prncipal
427332e version 1
93393c9 version 0
```



Correction des commits

- Si on décide d'annuler notre dernier commit, il faut le supprimer de l'historique. Cette opération peut être réalisée avec la commande **\$ git reset HEAD^**.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git log --oneline
8b63f89 (HEAD -> master) restauration ProgPrincipal.py
888c1a4 Suppression ProgPrincipal
2c9f609 1ere MAJ
03a51bd Ajout prog prncipal
427332e version 1
93393c9 version 0
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git reset HEAD^
```

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/desktop/TPs_GIT/TPRollback (master)
$ git log --oneline
888c1a4 (HEAD -> master) Suppression ProgPrincipal
2c9f609 1ere MAJ
03a51bd Ajout prog prncipal
427332e version 1
93393c9 version 0
```



Correction des commits

Annuler plusieurs commits

git reset permet aussi de **supprimer plusieurs commits**. Pour cela il y a plusieurs possibilités :

- Pour supprimer les 'n' derniers commits, il suffit de suffixer le caractère '^' n fois à la référence 'HEAD'. Par exemple pour supprimer les 5 derniers commits :

```
$ git reset HEAD^^^^^
```

- Notation compressée :

```
$ git reset HEAD~5
```

- Pour supprimer tous les commits après le commit ayant l'id 'f1fa39c' :

```
$ git reset f1fa39c
```

- Pour supprimer le commit ayant l'id 'f1fa39c' et ceux qui le suivent (et donc revenir *avant* le commit 'f1fa39c') :

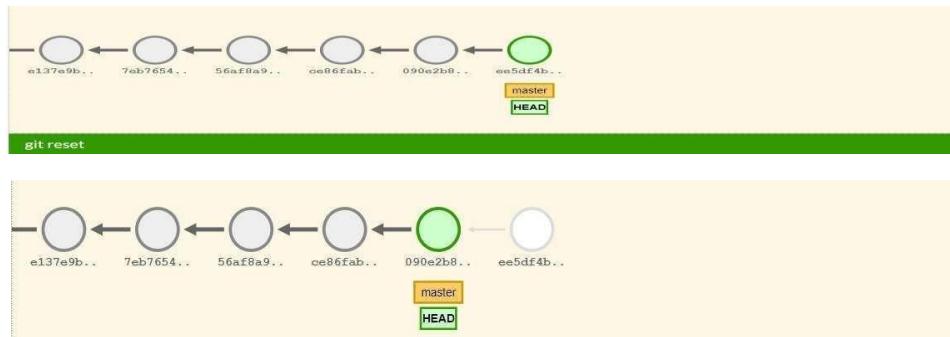
```
$ git reset f1fa39c^
```



- **git checkout HEAD[^]** : déplacement de la référence **HEAD** mais sans *toucher* à l'historique enregistré (et **synchronisation** du répertoire de travail)
- **git reset HEAD[^]** : déplacement de la référence de la tête de l'historique (**master**), ce qui force un déplacement de **HEAD** (mais **sans modification** du répertoire de travail)



`git reset` va donc déplacer en arrière la référence qui *pointe* sur le dernier commit de l'historique (le **master**). C'est comme si ce commit n'avait jamais été effectué



Correction des commits

- **git checkout** agit sur le répertoire de travail : le répertoire de travail est restauré dans l'état enregistré dans le commit sélectionné.

- ✓ Pour cela, Git peut modifier vos fichiers, et en ajouter ou supprimer d'autres (Git ne touche cependant pas aux fichiers non suivis).
- ✓ Cette commande ne modifie pas l'historique enregistré dans le dépôt (la référence **master** n'est pas modifiée). On peut donc revenir à la dernière version avec `git checkout master`.

- **git reset** agit sur l'historique : la référence de tête (**master**) est déplacée, ce qui supprime des commits de l'historique.



Correction des commits

[Annuler des commits](#)

- **git reset** agit sur l'historique : la référence de tête (**master**) est déplacée, ce qui supprime des commits de l'historique.

• **Avec l'option --soft** : le contenu du répertoire de travail n'est pas modifié : les différences entre le répertoire de travail et le nouveau commit de tête sont considérées comme des modifications placées dans l'index, mais non encore validées.



Résultat : on revient juste au moment où l'on avait fait le git add mais sans avoir encore fait le commit.

• **Dans sa version de base (option --mixed par défaut)** : le contenu du répertoire de travail n'est toujours pas modifié : les différences entre le répertoire de travail et le nouveau commit de tête sont considérées comme non indexées.

Résultat : on revient un cran plus tôt, juste au moment où l'on avait modifié les fichiers mais sans avoir encore fait le git add.

• **Dans sa version --hard** : le répertoire de travail est également synchronisé avec le nouveau commit de tête.

Résultat : on revient un cran de plus en arrière, avant le moment où l'on avait modifié les fichiers.



Correction des commits

Récapitulons

- **git checkout <commit_id>** est une commande utilisée très régulièrement lorsque l'on travaille avec des *branches*, comme vous le verrez bientôt. C'est l'usage principal de cette commande.
- **git checkout <un_fichier>** est occasionnellement utilisée pour supprimer des modifications que l'on vient d'ajouter à un fichier.
- **git reset** est rarement utilisée. Elle ne sert, à priori, que lorsque l'on décide d'abandonner un travail en cours et de revenir en arrière.

ATTENTION : Comme pour toute commande modifiant l'historique, git reset ne doit pas être utilisé sur des historiques partagés avec d'autres contributeurs.

Revenir en arrière avec revert



- **git revert** : cette commande est utilisée pour annuler les modifications partagées en créant de nouveaux commits. Les nouveaux commits inversent les modifications précédentes, de sorte que l'historique de commits reste cohérent. Cela signifie que la commande **git revert** est une opération sûre et ne supprime pas les modifications précédentes.
- En résumé, si vous souhaitez annuler des modifications non partagées, utilisez **git reset**. Si vous souhaitez annuler des modifications partagées sans supprimer l'historique des commits, utilisez **git revert**.



Revenir en arrière (1)

- Supposons que vous ayez un projet Git avec trois commits, numérotés 1, 2 et 3, comme suit: Commit 1 -> Commit 2 -> Commit 3
- Supposons également que vous avez modifié le contenu du projet et avez effectué un nouveau commi : Commit 1 -> Commit 2 -> Commit 3 -> **Commit 4**
- Maintenant, supposons que vous souhaitez annuler le dernier commit (**Commit 4**) en raison d'une erreur. Si vous utilisez **git reset**, vous pouvez supprimer le commit 4 et revenir à l'état précédent du projet en utilisant la commande suivante: `git reset --hard HEAD~1`
→ Cela supprimera le dernier commit et réinitialisera l'état du projet au commit 3. Cependant, cette commande supprime définitivement le commit 4 et toutes les modifications qui y sont associées. Si vous avez partagé ce commit avec d'autres personnes, cela peut causer des problèmes.

Revenir en arrière (1)

- Si vous utilisez **git revert**, vous pouvez annuler le commit 4 en créant un nouveau commit qui annule les modifications apportées dans le commit 4 en utilisant la commande suivante: **git revert HEAD**
- Cela créera un nouveau commit (numéroté 5 dans cet exemple) qui annule les modifications apportées dans le commit 4, tout en préservant l'historique des commits. **Cela signifie que si d'autres personnes ont déjà récupéré le commit 4, elles pourront toujours le récupérer, mais verront que le commit 5 l'annule.**
- En résumé, utilisez **git reset** si vous souhaitez supprimer définitivement des commits et **git revert** si vous souhaitez annuler des modifications tout en préservant l'historique des commits.



Rolback : Récap

supposons que vous avez effectué un commit sur la branche principale et que vous souhaitez annuler ce commit :



1. Tout d'abord, utilisez la commande `git log` pour afficher l'historique des commits et trouver l'identifiant du commit que vous souhaitez annuler : `git log`
2. Copiez l'identifiant du commit que vous souhaitez annuler.
3. Utilisez la commande `git revert` suivie de l'identifiant du commit pour créer un nouveau commit qui annule les changements apportés par le commit précédent : `git revert <identifiant-commit>`
4. Git ouvrira l'éditeur de texte par défaut pour vous permettre de saisir un message de commit expliquant l'annulation. Une fois que vous avez saisi votre message de commit, enregistrez et quittez l'éditeur de texte. Votre annulation est maintenant effectuée et un nouveau commit est créé pour refléter l'annulation.
5. Utilisez la commande `git push` pour pousser les modifications vers votre dépôt distant si nécessaire :`git push`

TP

CHAPITRE 3

NOTION DE BRANCHES

Git : Notion de branches

Branche de production **Main ou Master**

Le principal avantage de Git est **son système de branches**.

- C'est sur ces branches que repose toute les avantages de Git !
- Les différentes branches correspondent à des copies de votre code principal à un instant T, où vous pourrez tester toutes les modifications sans impacter le code principal.
- Sous Git, la branche principale est appelée la **branche main**, ou **master** pour les dépôts créés.
- GitHub a opéré un **changement** de terme de “**master**” à “**main**” en octobre 2020 (les deux termes sont retenus).
- Il faut voir les branches comme autant des dossiers différents.

Git : Notion de branches

Branche de production Main ou Master

- ❖ La création de branches est une fonctionnalité disponible dans la plupart des systèmes de contrôle de version.
- ❖ Le fait de créer une branche à part consiste à créer un **pointeur virtuel pointant sur toutes les opérations effectuées au niveau de cette branche**.
- ❖ Par exemple, si un développeur voulait développer un feature, il le fait ressortir dans une branche à part.
- ❖ Ainsi, le feature sera codé et testé au niveau de la branche sans toucher à la branche principale.
- ❖ Après avoir assuré qu'il fonctionne correctement, on pourra le fusionner dans la branche principale (main, master).

Git : Notion de branches

Une branche est une succession de commits, avec une référence nommée désignant la tête de cette succession.

Lorsque que l'on initialise un dépôt Git, la branche par défaut se nomme master. Lorsqu'on veut faire évoluer notre projet, cela peut avoir des effets de bord non désirés et créer des bugs.

- ✓ Garder intacte la version stable courante tant que nous n'avons pas testé correctement notre nouvelle version.

Avoir la possibilité de pouvoir enregistrer notre travail en cours dans une succession de commits, et donc une branche, qui soit gérée séparément de la version stable.

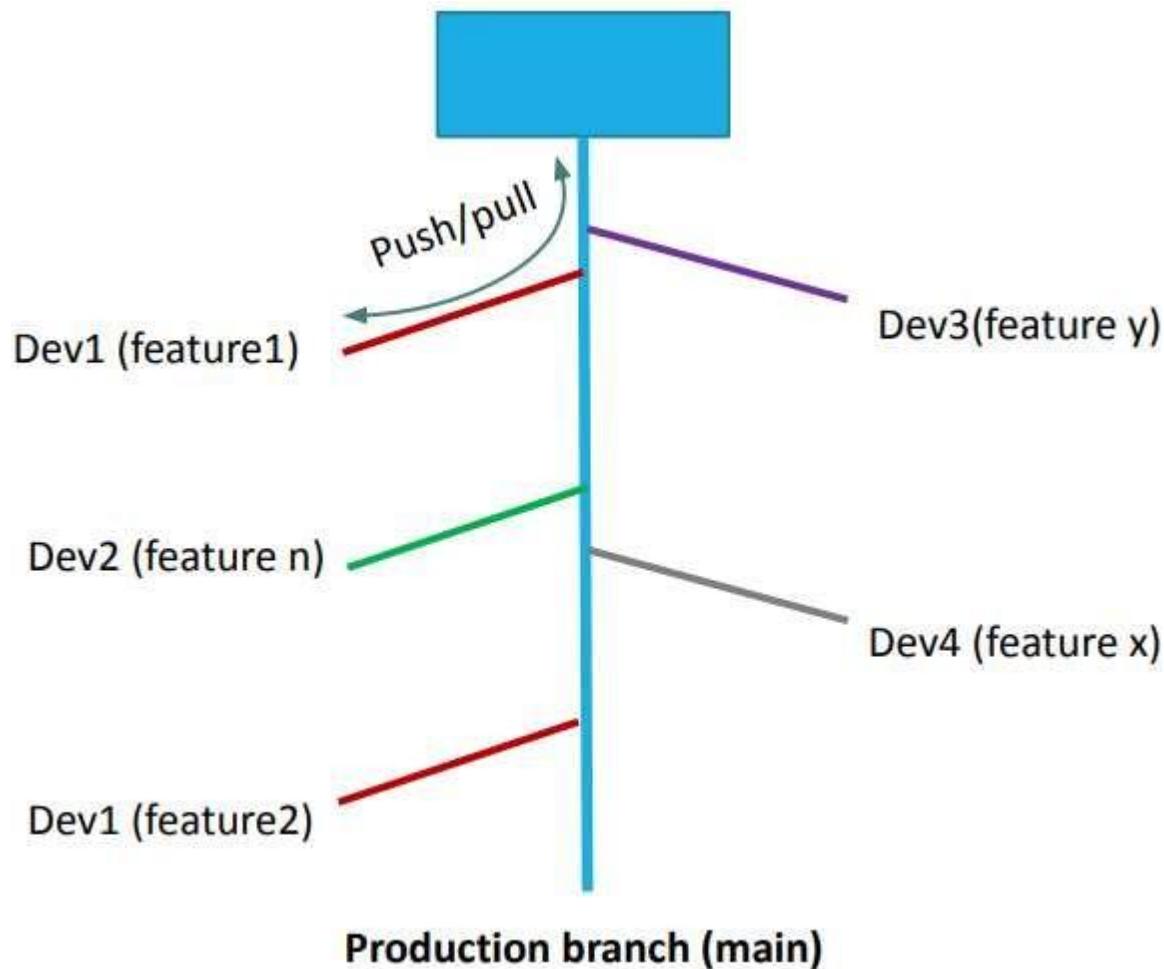
Git : Notion de branches

Une bonne pratique est ainsi de créer une nouvelle branche pour chaque évolution, que ce soit une nouvelle fonctionnalité, une amélioration ou une correction de bug.

- ✓ Une grande souplesse et tranquillité d'esprit. Si l'évolution s'avère inutile ou même catastrophique, ce n'est pas grave, il suffira d'abandonner la branche et de la supprimer sans que cela ait d'impact sur la branche contenant la version stable.
- ✓ Git offre des outils puissants permettant de gérer les évolutions sous forme de branches.
- ✓ Une branche est rattachée à un commit donné, point de départ du travail sur l'évolution. Ce commit peut faire partie de la branche master ou de toute autre branche.

Git : Notion de branches

Branche de production **Main ou Master**



Git : Notion de branches

Comment fonctionne

- ❖ Une branche représente une ligne de développement indépendante.
- ❖ Elle peut être considérée comme un moyen de créer un tout nouveau répertoire de travail, une zone de préparation et un historique du projet.
- ❖ Les **nouveaux commits** sont enregistrés dans **l'historique de la branche actuelle**, ce qui entraîne un embranchement dans l'historique du projet.
- ❖ La commande `git branch` nous permet de créer, répertorier, renommer et supprimer des branches.
- ❖ Il ne nous permet pas de basculer entre les branches.
- ❖ Pour cette raison, une commande `git checkout` et `git merge` seront utiles.

!!git merge vous permet de sélectionner les lignes de développement indépendantes créées avec `git branch` et de les intégrer à une seule branche

Git : Notion de branches

Créer des branches

- ❖ Il est important de comprendre que les branches ne sont que des pointeurs vers des commits.
- ❖ Lorsque nous créons une branche, tout ce que Git a à faire est de créer un nouveau pointeur, sans changer le dépôt.
- ❖ Nous démarrons avec un dépôt qui ressemble à ceci :



Git : Notion de branches

Branche de production **Main ou Master**

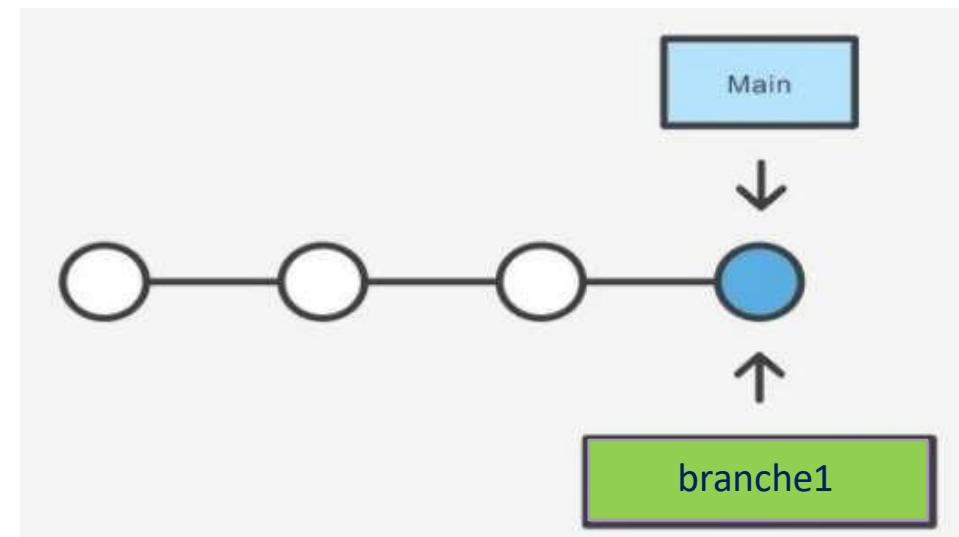
- ❖ Ensuite, nous allons créer une branche en utilisant la commande :

```
$ git branch branche1
```

L'historique du dépôt reste inchangé. Tout ce que nous obtenons est un nouveau pointeur vers le commit

- ❖ Notons que cela ne crée que la nouvelle branche.

- ❖ Pour commencer à y ajouter des commits,
nous devons le sélectionner avec **git checkout**,
puis créer des modifications et utiliser les
commandes standard **git add** et **git commit**.



Git : Notion de branches

Git Checkout

- ❖ Dans Git, un "checkout" permet de basculer entre différentes versions d'une entité ciblée.
 - ❖ La commande `git checkout` opère sur trois entités distinctes : les fichiers, les validations (commits) et les branches.
 - ❖ Dans le sujet **Annulation des modifications**, nous avons vu comment `git checkout` affiche les anciens commits.
 - ❖ L'extraction des branches est similaire à l'extraction des anciens commits et fichiers dans la mesure où le répertoire de travail est mis à jour pour correspondre à la branche (révision sélectionnée).
- ➔ Ce pendant, les nouvelles modifications sont enregistrées dans l'historique du projet, c'est-à-dire qu'il ne s'agit pas d'une opération en lecture seule.

Git : Notion de branches

Vérification des branches

- ❖ La commande `git checkout` permet de naviguer entre les branches créées par `git branch`.
- ❖ L'extraction d'une branche met à jour les fichiers du répertoire de travail pour correspondre à la version stockée dans cette branche, et indique à Git d'enregistrer tous les nouveaux commits sur cette branche.
- ❖ La commande `git checkout` peut parfois être confondue avec la commande `git clone`.
- ❖ La différence entre ces deux commandes est que `clone` fonctionne pour **récupérer le code** d'un dépôt distant, alternativement `checkout` fonctionne pour **basculer entre les versions de code** déjà sur le système local.

Git : Notion de branches

Utilisation : Branches existantes

- ❖ En supposant que le dépôt dans lequel nous travaillons contient des branches préexistantes, nous pouvons basculer entre ces branches à l'aide de `git checkout`.
- ❖ Pour savoir quelles branches sont disponibles et quel est le nom de la branche actuelle, nous pouvons exécuter `git branch`.

```
$ git branch
```

```
main
```

```
Branche1
```

```
$ git checkout branche1
```

- ❖ Cet exemple montre comment afficher une liste des branches disponibles en exécutant la commande `git branch` et passer à une branche spécifiée, dans ce cas, le `branche1`

Git : Notion de branches

Création de branche distante

- ❖ Les exemples précédents ont tous démontré des opérations de branches locales.
- ❖ La commande `git branch` fonctionne également sur les branches distantes.
- ❖ Pour opérer sur des branches distantes, un dépôt distant doit être configuré et ajouté à la configuration du dépôt local.
- ❖ Si vous clonez un dépôt, **le dépôt distant est automatiquement ajouté sous le nom « origin ».**
- ❖

```
$ git remote origin https://github.com/leilabousbia/projet-tic-1q.git
```

```
# Ajouter un dépôt distant à la configuration du dépôt local
```

```
$ git push <origin> branche1~ # pushes la branche branche1 vers origin
```

→ Cette commande poussera une copie de la branche locale `branche1` vers le dépôt distant `<remote>`.

Git : Notion de branches

Création de branche distante

La commande git remote origin <URL> est utilisée pour ajouter ou modifier la configuration de votre dépôt Git en lui associant un dépôt distant, généralement hébergé sur une plateforme telle que GitHub.

Dans l'exemple donné, la commande git remote origin https://github.com/leilabousbia/projet-tic-1q.git indique à Git d'ajouter ou de mettre à jour le paramètre origin pour qu'il pointe vers le dépôt distant situé à l'URL https://github.com/leilabousbia/projet-tic-1q.git.

Plus précisément:

git remote est une commande Git utilisée pour gérer les dépôts distants associés à votre dépôt local.
origin est le nom donné à la configuration du dépôt distant. origin est un nom couramment utilisé pour désigner le dépôt distant principal auquel votre dépôt local est lié, mais vous pouvez choisir un autre nom si vous le souhaitez.
https://github.com/leilabousbia/projet-tic-1q.git est l'URL du dépôt distant sur GitHub. Cette URL est spécifique au dépôt que vous souhaitez associer à votre dépôt local. Assurez-vous d'utiliser l'URL correcte de votre propre dépôt distant.

Une fois que vous avez exécuté cette commande avec succès,

vous pouvez utiliser des commandes Git telles que git push pour envoyer vos commits vers le dépôt distant associé à origin. Par exemple, vous pouvez utiliser git push origin main pour pousser vos modifications vers la branche principale (main) du dépôt distant origin.

En résumé, la commande git remote origin <URL> établit une connexion entre votre dépôt local et un dépôt distant spécifié par l'URL donnée, permettant ainsi de synchroniser vos modifications entre les deux.

Git : Notion de branches

Quelques commandes utiles

\$ git branch --list // Répertoriez toutes les branches de dépôt.

\$ git branch <branch> // Créer une nouvelle branche appelée <branch>.

\$ git branch -d <branch_name> // Supprimer la branche spécifiée, Git vous empêche de supprimer la branche si elle contient des modifications non fusionnées

\$ git branch -D <branch_name> // Forcer la suppression de la branche spécifiée, même si elle contient des modifications non fusionnées.

\$ git branch -m <branch> // Renommer la branche actuelle en <branch>.

\$ git branch -m old_name new_name // Renommer la branche.

\$ git branch -a // Répertorier toutes les branches distantes (Pour lister toutes les branches).

\$ git checkout branch_name // Pour passer à une branche spécifique

Git : Notion de branches

Quelques commandes utiles

`$ git push origin --delete name_branch // Pour supprimer une branche distante.`

Ou bien :

`$ git push origin :name_branch // Pour supprimer une branche distante.`

→ Cela enverra un signal de suppression au dépôt origin distant qui déclenchera une suppression de la branche distante.

`$ git checkout -b "branch_name" // créer et switcher dans une branche à la fois`

Git : Notion de branches

1. \$ git branch // Pour voir la branche par défaut (**master**)

Manipulation des étapes de branche :

2. \$ git branch feature1 // Pour créer une nouvelle branche (**feature1**)

3. \$ git branch // Pour voir les branches (master, feature1)

4. \$ git checkout feature1 // Pour basculer de : master → feature1

5. \$ git branch // Pour voir où il s'est pointé (**feature1**)

→ Pour créer une branche et y basculer directement, nous pouvons utiliser :

\$ git checkout -b feature2

6. Essayons de créer différents fichiers dans ces branches et comparons-les avec la branche principale

→ On cherche à envoyer cette nouvelle branche, nous utilisons \$ git push origin feature

→ Mais, avant d'envoyer cette branche à distance, \$ git branch --set-upstream-to=origin/feature

Git : Notion de branches

6. Essayons de créer différents fichiers dans ces branches et **Manipulation des étapes de branche :** comparons-les avec la branche principale

- On cherche à envoyer cette nouvelle branche, nous utilisons `$ git push origin feature`
- Mais, avant d'envoyer cette branche à distance, `$ git branch --set-upstream-to=origin/feature`
- `git branch --set-upstream-to=progit/master`

TP3



Chapitre 4

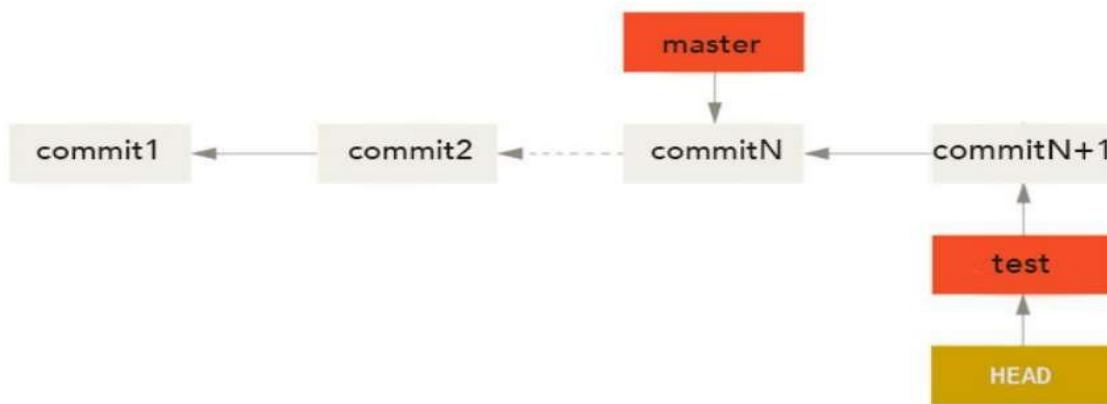
Fusion et rebasage

- Dans Git, Les branches permettent de travailler sur de nouvelles fonctionnalités sans impacter la branche principale.
- Les fonctionnalités sont développées sur des branches connexes et testées avant d'être réintégrées dans la branche principale.
- Pour réintiquer les fonctionnalités, il faut rapatrier le contenu des branches dans la branche principale.
- Git propose deux méthodes pour cela : la fusion des branches ou le rebasage.

Fusionner des branches

Exemple 1 :

- Commençons avec un cas plus simple et imaginons que notre projet soit dans cet état :



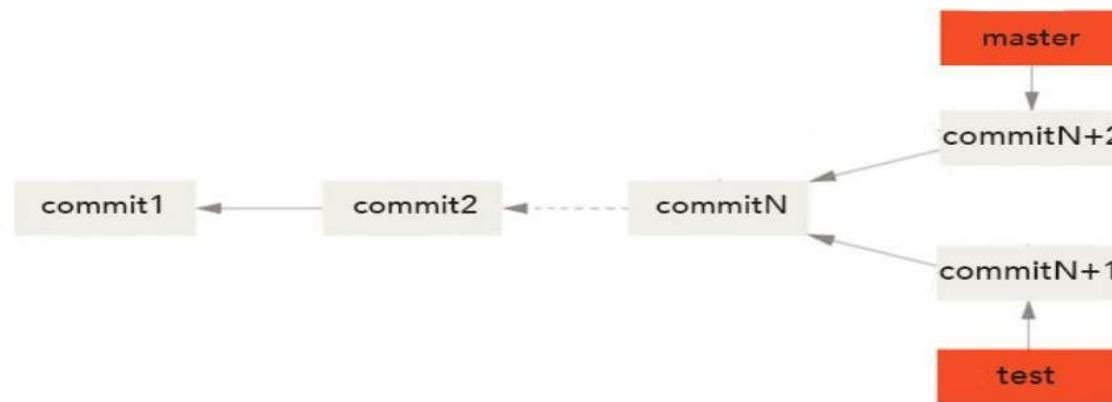
Ici, on a une branche **test** qui pointe sur un commit **commitN+1** et une branche **master** qui pointe sur un commit **commitN**. **commitN** est l'ancêtre direct de **commitN+1** et il n'y a donc pas de problème de divergence.

- Pour fusionner nos deux branches, on va se placer sur **master** avec une commande `git checkout` puis taper une commande `git merge` avec le nom de la branche qu'on souhaite fusionner avec **master**.
- Dans ce cas, “fusionner” nos deux branches revient finalement à faire avancer **master** au niveau du commit pointé par **test**. C'est exactement ce que fait Git et c'est ce qu'on appelle un “fast forward” (avance rapide).
- Il ne nous reste alors plus qu'à effacer notre branche **test**. On peut faire cela en utilisant la commande `git branch -d`.

Fusionner des branches

Exemple 2 :

Reprendons maintenant la situation précédente avec deux branches **dont les historiques divergent**. On peut représenter cette situation comme cela :



- Pour fusionner deux branches ici on va à nouveau se placer dans la branche dans laquelle on souhaite fusionner puis effectuer un git merge.
1
3
9
- Dans cet exemple, Git réalise une fusion en utilisant 3 sources : le dernier commit commun aux deux branches et le dernier commit de chaque branche.

Fusionner des branches

Exemple 2 :

- Git crée automatiquement un nouvel instantané dont le contenu est le résultat de la fusion ainsi qu'un commit qui pointe sur cet instantané.
- Ce commit s'appelle **un commit de fusion** et est spécial puisqu'il possède plusieurs parents.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git branch
* master
  test

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git merge test
Merge made by the 'ort' strategy.
 file4.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file4.txt

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git log --oneline
618eeac (HEAD -> master) Merge branch 'test'
b38db4b nouveau fichier file5 ajouté
5003d1d (test) nouveau fichier file4 ajouté
82c6ade 1ère maj
068c083 commit 3
64fb884 commit 2
e746a25 commit 1
```

Répertoire de travail		
Nom	Modifié le	Type
.git	01/05/2023 17:25	Dossier
file1	28/04/2023 16:43	Document
file2	28/04/2023 16:42	Document
file3	28/04/2023 16:42	Document
file4	01/05/2023 17:25	Document
file5	01/05/2023 16:30	Document

Fusionner des branches

- La commande "**git merge --squash**" fusionne une branche dans une autre branche sans créer de commit de fusion.
- Toutes les modifications de la branche fusionnée sont rassemblées en un seul commit sur la branche de destination.
- Vous pouvez inclure un message de commit personnalisé pour le commit de fusion en utilisant la commande "git commit".
- La commande "git merge --squash" est utile lorsque vous ne voulez pas inclure l'historique complet de la branche source dans la branche de destination.
- La commande "git merge --squash" ne supprime pas la branche source après la fusion¹, vous devez la supprimer manuellement en utilisant la commande "git branch -d".

Fusionner des branches

- Notez que dans le cas d'une fusion à trois sources, il se peut qu'il y ait des conflits. Cela va être notamment le cas si une même partie d'un fichier a été modifiée de différentes manières dans les différentes branches. Dans ce cas, lors de la fusion, **Git nous alertera du conflit et nous demandera de le résoudre avant de terminer la fusion des branches.**
- **Exemple :** Imaginons que nos deux branches possèdent un fichier LISEZMOI.txt et que les deux fichiers LISEZMOI.txt possèdent des textes différents. Git va automatiquement “fusionner” les contenus des deux fichiers en un seul qui va en fait contenir les textes des deux fichiers de base à la suite l'un de l'autre avec des indicateurs de séparation.
- On peut alors ouvrir le fichier à la main et choisir ce qu'on conserve (en suppri¹₂mant les parties qui ne nous intéressent pas par exemple). Dès qu'on a terminé l'édition, on va taper une commande **git add** pour marquer le conflit comme résolu. On n'aura alors plus qu'à effectuer un **git commit** pour terminer le commit de fusion.

Rebaser des branches

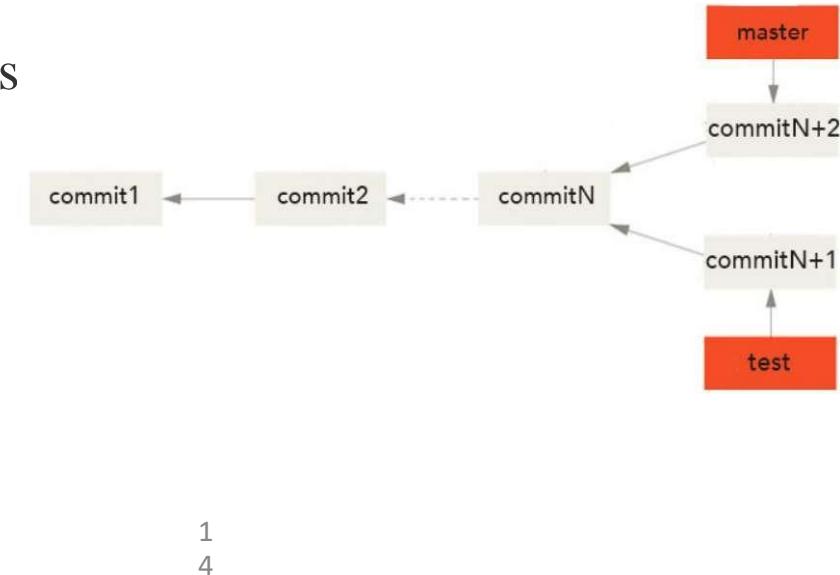
- Git nous fournit deux moyens de rapatrier le travail effectué sur une branche vers une autre : on peut fusionner ou rebaser.
- Reprenons notre exemple précédent avec nos deux branches

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git log --oneline
618eeac (HEAD -> master) Merge branch 'test'
b38db4b nouveau fichier file5 ajouté
5003d1d (test) nouveau fichier file4 ajouté
82c6ade 1ère maj
068c083 commit 3
64fb884 commit 2
e746a25 commit 1

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git branch
* master
  test

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git reset --hard b38db4b
HEAD is now at b38db4b nouveau fichier file5 ajouté

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git log --oneline
b38db4b (HEAD -> master) nouveau fichier file5 ajouté
82c6ade 1ère maj
068c083 commit 3
64fb884 commit 2
e746a25 commit 1
```



1
4
Ici, pour revenir à cet état, on a exécuté la commande `$git reset --hard b38db4b` (dernier commit de la branche master avant l'étape de fusion == on vient d'annuler l'opération de fusion)

Rebaser des branches

- Plutôt que d'effectuer une fusion à trois sources, on va pouvoir rebaser les modifications validées dans commitN+1 dans notre branche **master**. On utilise la commande `git rebase` pour récupérer les modifications validées sur une branche et les rejouer sur une autre.
- Dans ce cas, Git part à nouveau **du dernier commit commun aux deux branches** (l'ancêtre commun le plus récent) puis récupère les modifications effectuées sur la branche qu'on souhaite rapatrier et les applique sur la branche vers laquelle on souhaite rebaser notre travail **dans l'ordre dans lequel elles ont été introduites**.

```
TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git log --oneline
b38db4b (HEAD -> master) nouveau fichier file5 ajouté
82c6ade 1ère maj
068c083 commit 3
64fb884 commit 2
e746a25 commit 1

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git rebase test
Successfully rebased and updated refs/heads/master.

TEK-UP@DESKTOP-5ILQ865 MINGW64 ~/Desktop/tpCoursBranches (master)
$ git log --oneline
98aabd3 (HEAD -> master) nouveau fichier file5 ajouté
5003d1d (test) nouveau fichier file4 ajouté
82c6ade 1ère maj
068c083 commit 3
64fb884 commit 2
e746a25 commit 1
```

Rebaser rejoue les modifications d'une ligne de commits sur une autre dans l'ordre d'apparition.

Rebaser des branches

la commande "git rebase" est utilisée pour récupérer les modifications validées sur une branche et les rejouer sur une autre branche. Voici comment cela fonctionne :

Supposons que vous ayez deux branches : la branche source (branch A) et la branche cible (branch B).

1. Assurez-vous que vous êtes sur la branche cible (branch B) en utilisant la commande "git checkout B".
2. Exécutez la commande "git rebase A" pour démarrer le processus de rebase. Cela signifie que vous souhaitez récupérer les modifications de la branche source (branch A) et les appliquer sur la branche cible (branch B).
3. Git va maintenant parcourir l'historique des commits de la branche cible (branch B) depuis le point où elle a divergé de la branche source (branch A). Il identifie les commits spécifiques qui ne sont pas encore présents dans la branche cible.
4. Git déplace ensuite les commits de la branche cible (branch B) temporairement hors de la branche et les met de côté.
5. Ensuite, Git applique un par un les commits de la branche source (branch A) sur la branche cible (branch B), un par un, dans l'ordre chronologique. Cela signifie que chaque commit de la branche source est réappliqué sur la branche cible.
6. Si des conflits surviennent pendant le processus de rebase, Git vous avertira et marquera les fichiers conflictuels. Vous devrez résoudre ces conflits manuellement en éditant les fichiers concernés.
7. Une fois que tous les commits de la branche source ont été rejoués sur la branche cible avec succès, Git termine le processus de rebase.
8. Vous pouvez maintenant continuer à travailler sur la branche cible avec les modifications récupérées de la branche source.

Comparaison

- La principale différence entre "git merge" et "git rebase" est que "git merge" fusionne deux branches en créant un nouveau commit de fusion, tandis que "git rebase" réapplique l'historique de commit de la branche source à la branche de destination en réécrivant l'historique des commits.
- "git merge" conserve l'historique de commit original de chaque branche, tandis que "git rebase" réécrit l'historique des commits de la branche source.
- "git merge" est utilisé pour fusionner des branches avec des modifications importantes et distinctes, tandis que "git rebase" est utilisé pour intégrer des modifications mineures dans une branche principale, pour réorganiser l'historique de commit ou pour maintenir un historique de commit linéaire.
- "git merge" peut introduire des conflits de fusion qui doivent être résolus manuellement¹, tandis que "git rebase" peut introduire des conflits de réécriture d'historique qui peuvent également nécessiter une résolution manuelle.

git merge vs git rebase

- Le choix entre "git merge" et "git rebase" dépend des besoins du projet et des préférences de l'équipe de développement.
- Voici quelques critères à considérer :
 - Si vous voulez conserver l'historique de commit original de chaque branche, utilisez "git merge".
 - Si vous voulez maintenir un historique de commit linéaire et plus facile à suivre, utilisez "git rebase".
 - Si vous voulez fusionner des branches avec des modifications importantes et distinctes, utilisez "git merge".
 - Si vous voulez intégrer des modifications mineures dans une branche principale, utilisez "git rebase".
 - Si vous voulez réorganiser l'historique de commit, utilisez "git rebase".
 - Si vous voulez éviter les conflits de fusion, utilisez "git rebase".
 - Si vous voulez éviter de réécrire l'historique de commit, utilisez "git merge".
- En fin de compte, il n'y a pas de réponse unique pour le choix entre "git merge" et "git rebase". Le choix dépend des besoins du projet, de la préférence de l'équipe de développement et de la façon dont vous voulez structurer l'historique de commit.

Git Merge

L'option git merge --squash

- ❖ La commande `git merge --squash` est une option de fusion qui nous permet de condenser l'historique Git des branches thématiques lorsque nous remplissons une demande d'extraction.
- ❖ Au lieu que chaque commit sur la branche thématique soit ajouté à l'historique de la branche par défaut, une fusion squash ajoute toutes les modifications de fichier à un seul nouveau commit sur la branche par défaut.
- ❖ Le commit de Merge Squash n'a pas de référence à la branche thématique, il produira un **nouveau commit** contenant toutes les modifications de la branche thématique.
- ❖ De plus, il est recommandé de supprimer la branche thématique pour éviter toute confusion.

Git Merge

La commande git merge --squash est utilisée pour fusionner les modifications d'une branche dans une autre branche, tout en condensant tous les commits en un seul commit. Cela permet d'avoir une historique de commits plus propre et plus concis.

Voici comment fonctionne la commande git merge --squash :

1. Tout d'abord, assurez-vous d'être sur la branche dans laquelle vous souhaitez fusionner les modifications. Par exemple, si vous voulez fusionner les modifications d'une branche de fonctionnalité dans la branche principale, passez à la branche principale en utilisant la commande git checkout main.
1. Exécutez la commande git merge --squash suivie du nom de la branche que vous souhaitez fusionner. Par exemple, si la branche de fonctionnalité s'appelle "feature-branch", la commande serait :

```
git merge --squash feature-branch
```

Git va fusionner toutes les modifications de la branche spécifiée, mais au lieu de créer plusieurs commits, il les regroupera en un seul commit.

Git Merge

L'option git merge --squash

À ce stade, vous pouvez modifier le message de commit pour le commit condensé. Git vous fournira un message de commit par défaut avec un résumé des modifications de la branche fusionnée.

Enregistrez et fermez le fichier de message de commit pour finaliser la fusion.

Git créera un nouveau commit avec toutes les modifications de la branche fusionnée en tant que commit unique, incorporant les modifications dans la branche sur laquelle vous étiez lorsque vous avez exécuté la commande.

Il est important de noter que git merge --squash ne crée pas un commit de fusion avec l'historique complet comme le ferait une fusion normale. Au lieu de cela, il crée un seul commit contenant les modifications combinées. Cela peut être utile pour maintenir un historique de commits propre et concis, en particulier lors de la fusion de branches de fonctionnalités de longue durée.

N'oubliez pas de pousser les modifications vers le dépôt distant après avoir effectué la fusion par écrasement (squash merge) si vous souhaitez partager les modifications avec les autres.

Git Merge

git rebase - git merge - git merge --squash

L'option git merge --squash

- ❖ La fusion conserve l'historique des commits tel quel.
- ❖ Donc la branche dans laquelle nous avions exécuté la fusion sera mal organisé.

\$ git merge feature --squash

\$ git status

\$ git commit –m “commit groupant tous les commits de la branche feature”

Git log --oneline --graph --decorate

La commande `git log --oneline --graph --decorate` est utilisée pour afficher l'historique des commits dans un projet Git de manière concise et visuelle.

`git log` : Cela indique à Git que vous souhaitez afficher l'historique des commits.

`--oneline` : Cela spécifie que vous souhaitez afficher chaque commit sur une seule ligne pour une vue plus concise.

`--graph` : Cela indique à Git de dessiner une représentation graphique de l'historique des commits, ce qui permet de voir facilement les branches et les fusions.

`--decorate` : Cela ajoute des informations supplémentaires à la sortie, telles que les noms des branches et les tags.

En utilisant cette commande, vous pouvez visualiser l'historique des commits de votre projet de manière plus facile à comprendre, ce qui peut vous aider à suivre les différentes branches et les fusions et à comprendre l'historique des modifications apportées à votre projet.

CHAPITRE 5

GIT MERGE CONFLIT

Git Merge conflit

D'où vient les conflits de merge

- ❖ Les **conflits** surviennent généralement lorsque :
 - si **deux** personnes ont **modifié les mêmes lignes** dans un fichier,
 - si un développeur **a supprimé** un fichier alors qu'un autre développeur **le modifiait**.
- ❖ Dans ces cas, Git **ne peut pas déterminer** automatiquement la **version correcte**.
- ❖ Les conflits n'affectent que le développeur qui effectue le merge, les autres membres de l'équipe ne sont pas conscients du conflit.
- ❖ Git **marquera le fichier** comme étant en **conflit** et **arrêtera le processus de merge**.
- ❖ Il **impose** alors aux développeurs de **résoudre le conflit**.

Git Merge conflit

→ Git ne parvient pas à lancer le merge

Types des conflits de merge

❖ Un merge peut entrer un état de conflit en deux points distincts ;

- Au début d'un processus de merge
- Au cours d'un processus de merge

❖ Le démarrage du merge échouera tant que Git verra qu'il y a **des changements dans le répertoire de travail ou la zone d'indexe du projet en cours.**

❖ Git ne parvient pas à lancer le merge, parce que ces changements pourraient être remplacés par les commits mergés.

→ Le cas **échéant** n'est pas dû à **des conflits avec d'autres développeurs**, mais bien **avec des changements locaux en cours.**

→ L'échec d'un merge au démarrage renverra le message d'erreur suivant :

`error: Entry '<fileName>' not uptodate. Cannot merge. (Changes in working directory)`

Git Merge conflit

Types des conflits de merge

→ Git rencontre un problème durant le merge

- ❖ Un échec pendant un merge indique un **conflit** entre la branche **locale actuelle** et la branche **mergée**.
- ❖ Cela indique un **conflit avec le code d'autres développeurs.**
- ❖ Git fera de son mieux pour merger les fichiers, mais vous laissera le soin de résoudre manuellement les conflits dans les fichiers concernés.
- ❖ Un échec en cours de merge renverra le message d'erreur suivant :

error: Entry '<fileName>' would be overwritten by merge. Cannot merge. (Changes in staging area)

Git Merge conflit

→ Afin de nous familiariser réellement avec les conflits de merge, nous allons produire un conflit à examiner et à le résoudre ultérieurement.

- ❖ Créez un répertoire `git-merge-test`, basculez vers ce répertoire, puis initialisez-le `git init`.
 - ❖ Créez un fichier texte `merge.txt` avec du contenu.
 - ❖ Ajoutez et commitez `merge.txt` au dépôt.
- Nous disposons à présent d'un nouveau dépôt avec une branche (master ou main) et un fichier `merge.txt` avec du contenu.

Création d'un conflit de merge

```
user@DESKTOP-RRVCC4I MINGW64 ~
$ mkdir git-merge-test

user@DESKTOP-RRVCC4I MINGW64 ~
$ cd git-merge-test

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test
$ git init
Initialized empty Git repository in C:/Users/user/git-merge-test/.git/

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ echo "contenu à supprimer" > merge.txt

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ cat merge.txt
contenu à supprimer

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git add merge.txt
warning: in the working copy of 'merge.txt', LF will be replaced by CRLF the next time Git touches it

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   merge.txt

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git commit -am "validation du contenu initial"
[master (root-commit) 17d7357] validation du contenu initial
 1 file changed, 1 insertion(+)
 create mode 100644 merge.txt
```

Git Merge conflit

Création d'un conflit de merge

- ❖ Nous allons maintenant créer une branche à utiliser en tant que merge conflictuel.
- ❖ Créer une branche nommée `new_branch_to_merge_later` et en faire un check-out
- ❖ Remplacer le contenu dans le fichier `merge.txt`
- ❖ Faire un commit du nouveau contenu
- ❖ Avec cette nouvelle branche : `new_branch_to_merge_later`, nous avons créé un commit qui remplace le contenu du fichier `merge.txt`.

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git checkout -b new_branch_to_merge_later
Switched to a new branch 'new_branch_to_merge_later'

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (new_branch_to_merge_later)
$ echo "contenu totalement différent à fusionner plus tard" > merge.txt

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (new_branch_to_merge_later)
$ git commit -am"modifié le contenu de merge.txt pour provoquer un conflit"
warning: in the working copy of 'merge.txt', LF will be replaced by CRLF the next time Git touches it
[new_branch_to_merge_later b2eae78] modifié le contenu de merge.txt pour provoquer un conflit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git Merge conflit

Création d'un conflit de merge

- ❖ Cette chaîne de commandes fait un check-out de la branche (master), ajoute du contenu au fichier `merge.txt` et le commite.
- ❖ Notre exemple de dépôt se retrouve ainsi à un état où nous avons deux nouveaux commits. Un dans la branche (master) et un dans la branche `new_branch_to_merge_later`.

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (new_branch_to_merge_later)
$ git checkout master
Switched to branch 'master'

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ echo "contenu à ajouter" >> merge.txt

user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git commit -a -m "contenu ajouté à merge.txt"
warning: in the working copy of 'merge.txt', LF will be replaced by CRLF the next time Git touches it
[master 8fed885] contenu ajouté à merge.txt
 1 file changed, 1 insertion(+)
```

Git Merge conflit

Création d'un conflit de merge

- À présent, exécutons la commande `git merge new_branch_to_merge_later` pour voir ce qui se produit !

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git merge new_branch_to_merge_later
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- Un conflit apparaît.

Git Merge conflit

Comment identifier les conflits de merge

- Comme nous l'avons vu dans l'exemple de la procédure, Git générera une sortie descriptive nous permettant de savoir qu'un CONFLIT s'est produit.
- Pour mieux interpréter, nous pouvons exécuter la commande `git status`.

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified: merge.txt

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

- La sortie de la commande `git status` indique que des chemins n'ont pas été mergés en raison d'un conflit.
- Le fichier `merge.txt` apparaît désormais à un état modifié.

Git Merge conflit

Comment identifier les conflits de merge

❖ Examinons le contenu du fichier en utilisant la commande `cat` et découvrons ce qui a été modifié.

❖ Nous pouvons voir certains ajouts étranges.

•<<<<< HEAD

•=====

•>>>>> new_branch_to_merge_later

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master|MERGING)
$ cat merge.txt
<<<<< HEAD
contenu à supprimer
contenu à ajouter
=====
contenu totalement différent à fusionner plus tard
>>>>> new_branch_to_merge_later
```

❖ Considérez ces nouvelles lignes comme des « séparateurs de conflit ».

❖ La ligne ===== est le « centre » du conflit.

❖ Tout le contenu entre le centre et la ligne <<<<< HEAD est dans la branche principale actuelle vers laquelle pointe la réf HEAD.

❖ Autrement, tout le contenu entre le centre et >>>>> new_branch_to_merge_later est présent dans notre branche de merge.

Git Merge conflit

Comment résoudre les conflits de merge avec la ligne de commande ?

- ❖ La manière la plus directe de résoudre un conflit de merge consiste à modifier le fichier conflictuel.
- ❖ Ouvrez le fichier **merge.txt** dans notre éditeur.
- ❖ Dans notre exemple, supprimons simplement tous les séparateurs de conflit.
- ❖ Le contenu du fichier **merge.txt** modifié devrait ressembler à ceci :

```
merge - Bloc-notes
Fichier Edition Format Affichage Aide
<<<<< HEAD
contenu à supprimer
contenu à ajouter
=====
contenu totalement différent à fusionner plus tard
>>>> new_branch_to_merge_later
```

```
*merge - Bloc-notes
Fichier Edition Format Affichage Aide
contenu à supprimer
contenu à ajouter
contenu totalement différent à fusionner plus tard |
```

Git Merge conflit

Comment résoudre les conflits de merge avec la ligne de commande ?

- ❖ Une fois le fichier modifié, utilisez `git add merge.txt` pour indexer le nouveau contenu mergé.
- ❖ Pour finaliser le merge, créez un commit :

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master|MERGING)
$ git commit -a -m "merger et résoudre le conflit de merge.txt"
[master fe87c67] merger et résoudre le conflit de merge.txt
```

- ❖ Git voit que le conflit a été résolu et crée un commit de merge pour finaliser le merge.
- ❖ Ajoutant la commande `git log --oneline --graph --decorate`

```
user@DESKTOP-RRVCC4I MINGW64 ~/git-merge-test (master)
$ git log --oneline --graph --decorate
* fe87c67 (HEAD -> master) merger et résoudre le conflit de merge.txt
|\ 
| * b2eae78 (new_branch_to_merge_later) modifié le contenu de merge.txt pour provoquer un conflit
* | 8fed885 contenu ajouté à merge.txt
|| 
* 17d7357 validation du contenu initial
```

Git Merge conflit

Les commandes Git qui peuvent aider à résoudre des conflits de merge

Outils généraux

\$ git status // permet d'identifier les fichiers en conflit durant un merge.

\$ git log --merge // --merge génère un journal contenant une liste de commits en conflit entre les branches de merge.

\$ git diff // permet de trouver les différences entre les états d'un dépôt/de fichiers. Cette commande est utile pour prévoir et éviter les conflits de merge.

\$ git checkout // permet d'annuler les changements apportés à des fichiers ou de modifier des branches

Git Merge conflit

Les commandes Git qui peuvent aider à résoudre des conflits de merge

Outils utiles en cas de conflits Git durant un merge

\$ git merge --abort // l'ajout de --abort mettra fin au processus de merge et réinitialisera la branche à son état antérieur au merge.

\$ git reset // reset peut être utilisée durant un conflit de merge pour réinitialiser les fichiers en conflit à un état fonctionnel connu.

Git Merge conflit

Les commandes Git qui peuvent aider à résoudre des conflits de merge

Stashing ou cachette

git stash stocke (ou stashe) temporairement les changements apportés à votre copie de travail :

- pour que vous puissiez effectuer d'autres tâches,
- puis revenir et les réappliquer par la suite.

Le stashing est pratique si vous avez besoin de changer rapidement de contexte et de travailler sur autre chose, mais que vous êtes en plein dans un changement de code et que n'êtes pas tout à fait prêt à commiter.

Notez que le stash est local pour votre dépôt Git. Les stashes ne sont pas transférés au serveur lors d'un push.

Git Merge conflit

Les commandes Git qui peuvent aider à résoudre des conflits de merge

Stashing ou cachette

\$ git stash // enregistre juste l'état de la zone de stash (d'indexe)

\$ git stash -a // enregistre l'état de la zone d'indexe et de la zone non suivie

\$ git stash list //afficher les caches

\$ git stash apply //appliquer le dernier stash et enregistré dans le stash

\$ git stash pop //appliquer mais effacez-le des caches enregistrées

Cherry-pick

\$ git Cherry-pick id_of_commit_of_the_branch_we.want_to_copy_from // Cherry-pick

est la copie d'un commit spécifique utile d'une branche à une autre