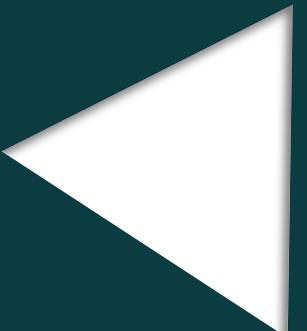


2025-2026



Git

Introduction



La problématique

Les problématiques

- Historique des modifications
 - Travail d'équipe
 - Sauvegarde distante

Solutions : gestionnaire de version



CVS



SVN



GIT



Mercurial



Bazaar

Les gestionnaires de versions

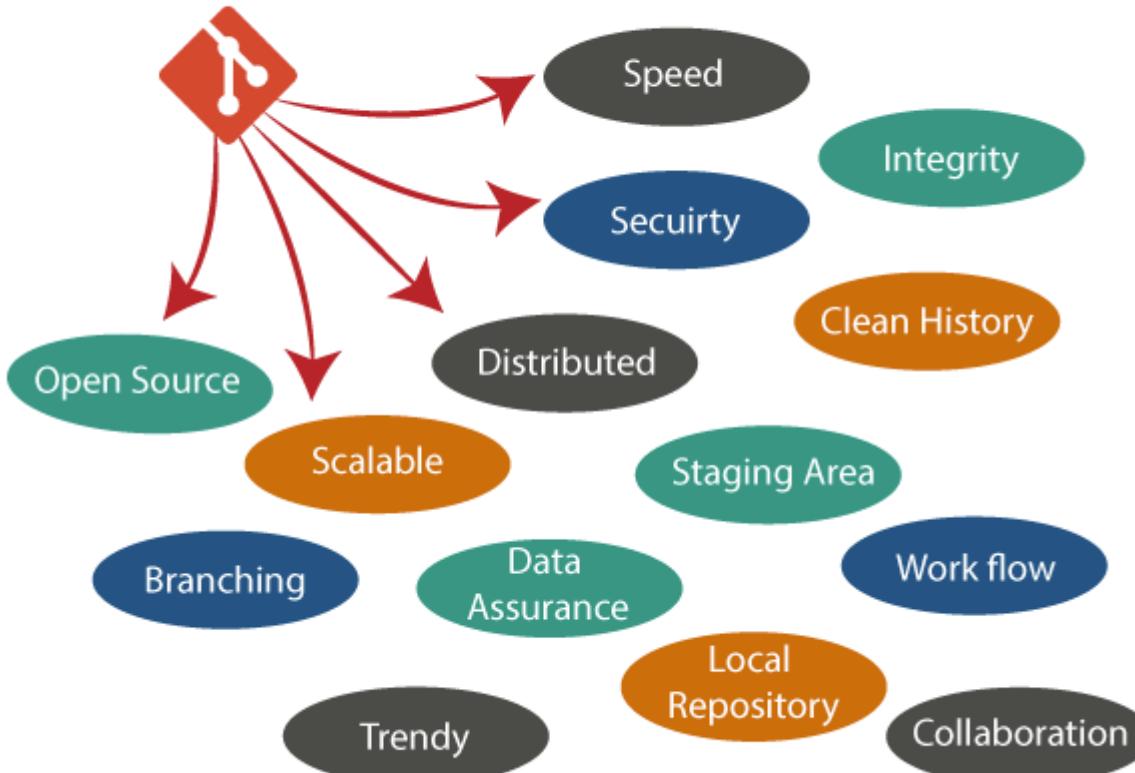
Caractéristique	CVS	SVN	Git	Mercurial	Bazaar
Type de gestion	Centralisé	Centralisé	Décentralisé	Décentralisé	Décentralisé
Branche (Branching)	Oui	Oui	Oui	Oui	Oui
Fusion (Merging)	Oui	Oui	Oui	Oui	Oui
Performance	Dépend de la taille	Bonne	Rapide	Rapide	Rapide
Facilité d'utilisation	Moins convivial	Convivial	Convivial	Convivial	Convivial
Support de fichiers binaires	Limité	Oui	Oui	Oui	Oui
Outils graphiques intégrés	Limités	Oui	Oui	Oui	Oui
Intégration avec IDE	Limitée	Bonne	Excellent	Bonne	Limitée
Popularité/Communauté	Diminuée	Grande	Très grande	Grande	Petite à Moyenne
Adoption dans l'industrie	En déclin	Grande	Très grande	Moyenne	Petite à Moyenne

Une solution se distingue : Git



Les avantages de Git

Why Git?



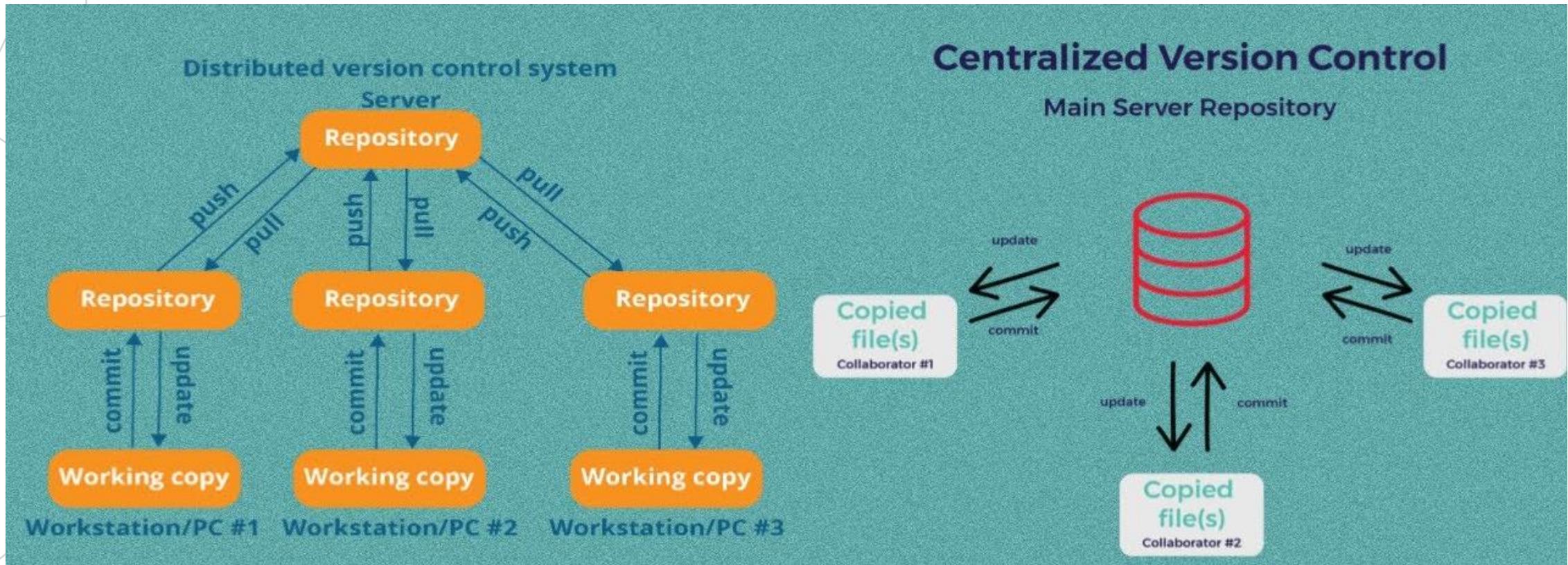


Centralisé ou pas

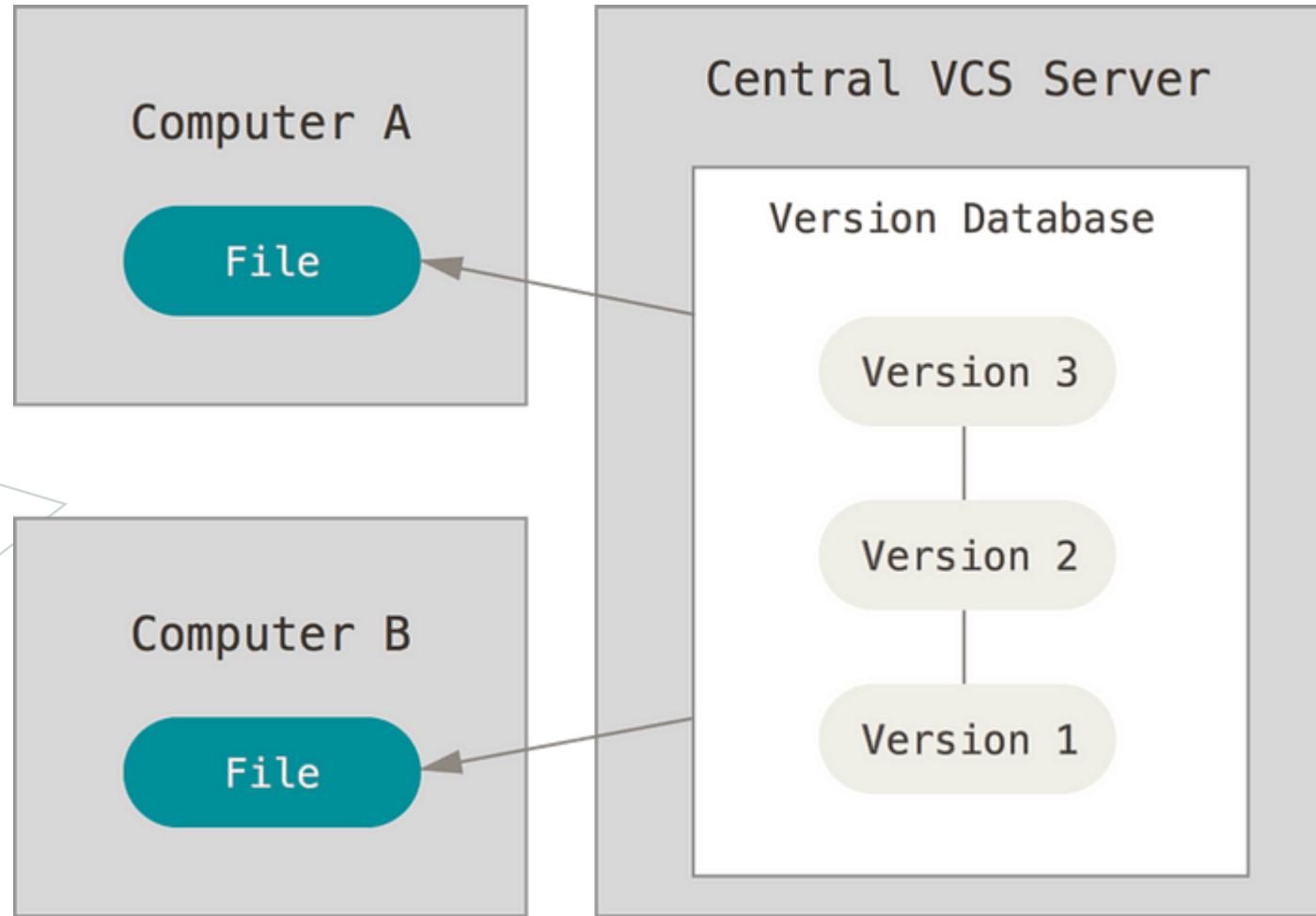
Les gestionnaires de versions

Caractéristique	CVS	SVN	Git	Mercurial	Bazaar
Type de gestion	Centralisé	Centralisé	Décentralisé	Décentralisé	Décentralisé
Branche (Branching)	Oui	Oui	Oui	Oui	Oui
Fusion (Merging)	Oui	Oui	Oui	Oui	Oui
Performance	Dépend de la taille	Bonne	Rapide	Rapide	Rapide
Facilité d'utilisation	Moins convivial	Convivial	Convivial	Convivial	Convivial
Support de fichiers binaires	Limité	Oui	Oui	Oui	Oui
Outils graphiques intégrés	Limités	Oui	Oui	Oui	Oui
Intégration avec IDE	Limitée	Bonne	Excellent	Bonne	Limitée
Popularité/Communauté	Diminuée	Grande	Très grande	Grande	Petite à Moyenne
Adoption dans l'industrie	En déclin	Grande	Très grande	Moyenne	Petite à Moyenne

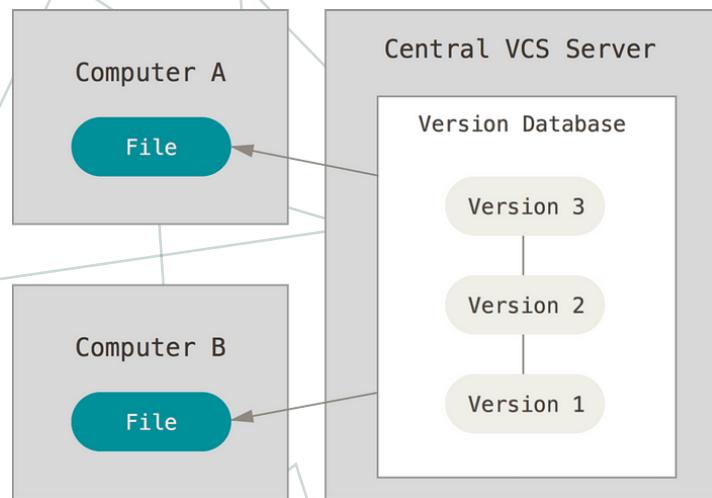
Décentralisation du code



Centralisé

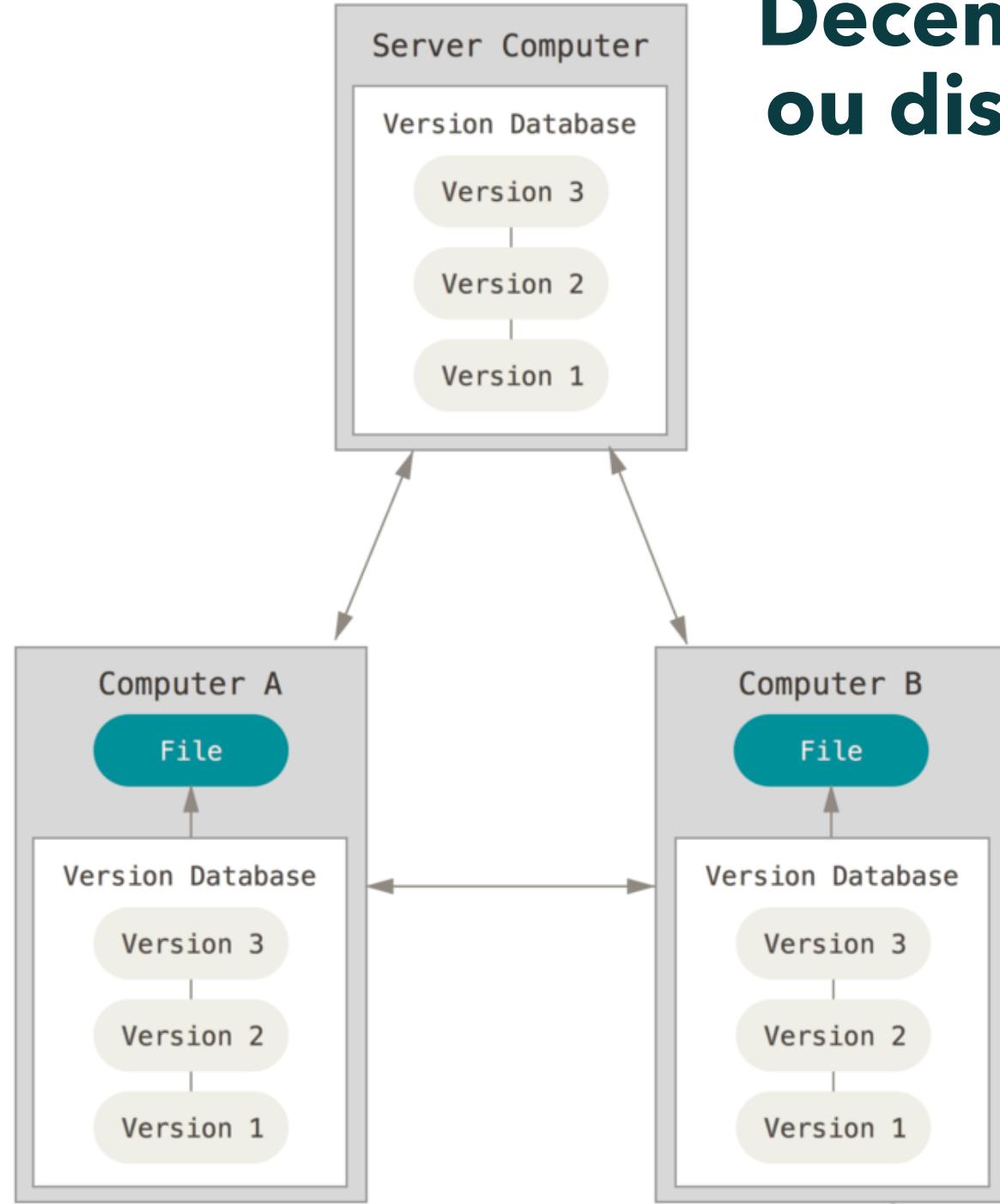


Centralisé



<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

Décentralisé ou distribué





Les commandes de base

Les commandes de bases

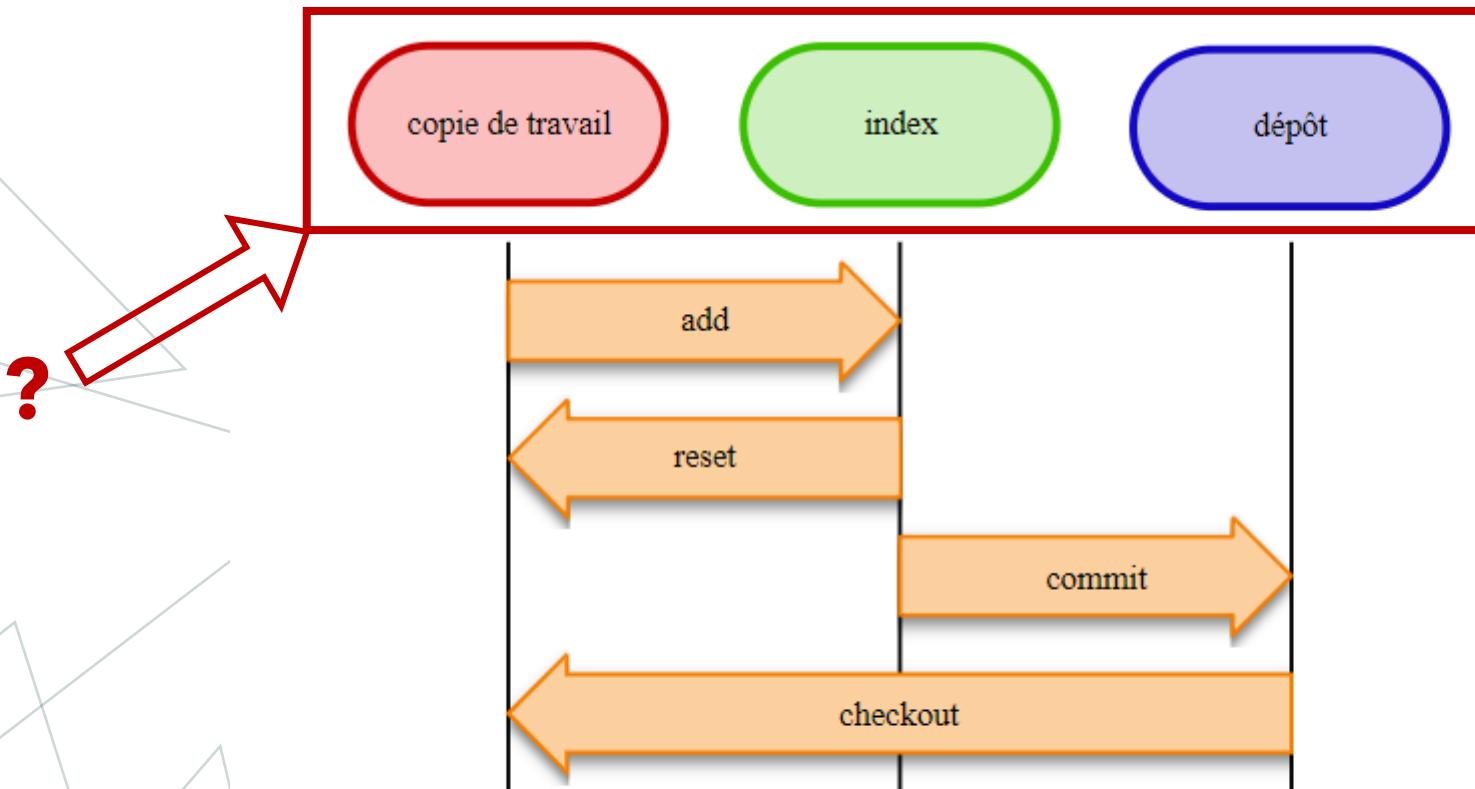


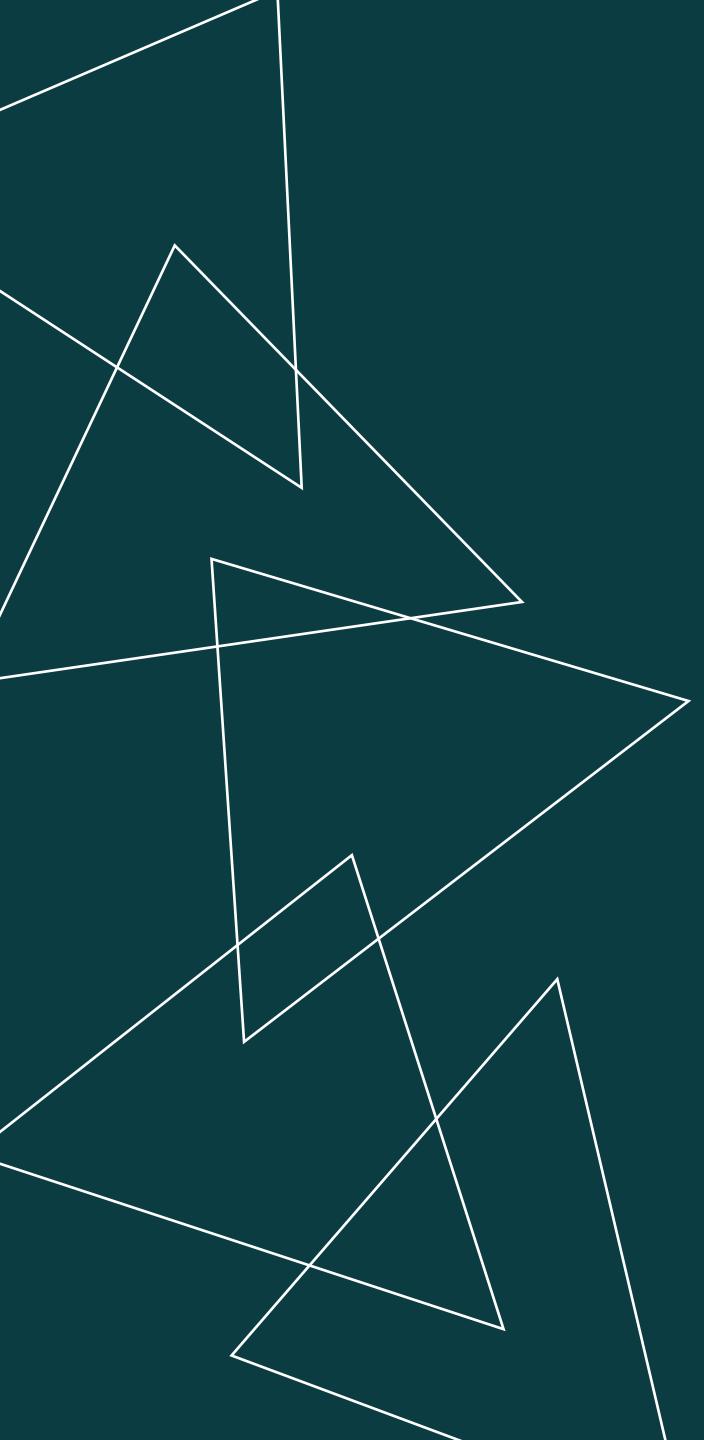
Figure inspirée de git-scm.org.

Git

Avant de commencer ...

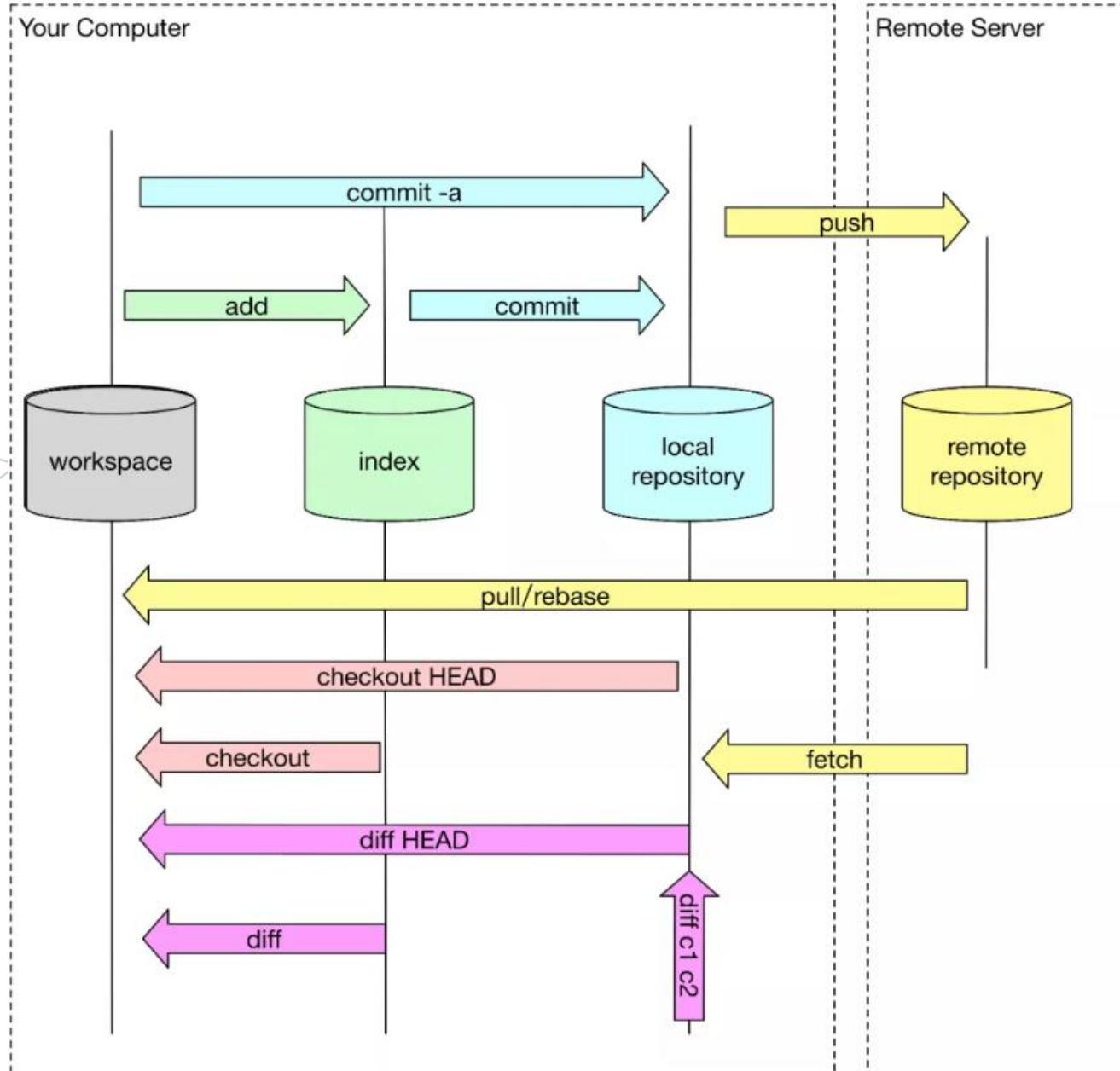
Vocabulaire

Terme Git	Définition
Repository (Dépôt)	Un endroit où Git stocke des versions de votre projet et ses historiques. Cela peut être local sur votre machine ou distant sur un serveur.
Commit (Validation)	Une "capture" d'un état spécifique de vos fichiers et dossiers. Un commit représente une nouvelle version de votre projet.
Branch (Branche)	Une copie indépendante de votre code qui peut évoluer séparément des autres branches. Cela permet le développement parallèle de fonctionnalités ou correctifs.
Merge (Fusion)	La combinaison de modifications provenant de différentes branches pour créer une version unique et cohérente de votre code.
Push (Pousser)	L'envoi de vos modifications locales vers un dépôt distant. Cela met à jour le dépôt distant avec vos dernières modifications.
Pull (Tirer)	Récupérer les modifications d'un dépôt distant et les fusionner dans votre copie locale.
Clone (Cloner)	Copier un dépôt distant sur votre machine locale, créant une version locale complète de ce dépôt.
Fork (Fourche)	Créer une copie indépendante d'un dépôt, généralement utilisée dans les projets open source pour contribuer sans avoir directement accès au dépôt d'origine.
Remote (Distant)	Un nom associé à un dépôt distant, généralement utilisé pour référencer un dépôt sur un serveur distant.
Stash (Cachette)	Temporairement sauvegarder des modifications non validées, vous permettant de passer à une autre tâche sans compromettre votre travail en cours.
Tag (Étiquette)	Un pointeur statique vers un commit spécifique, souvent utilisé pour marquer des versions stables de votre projet.



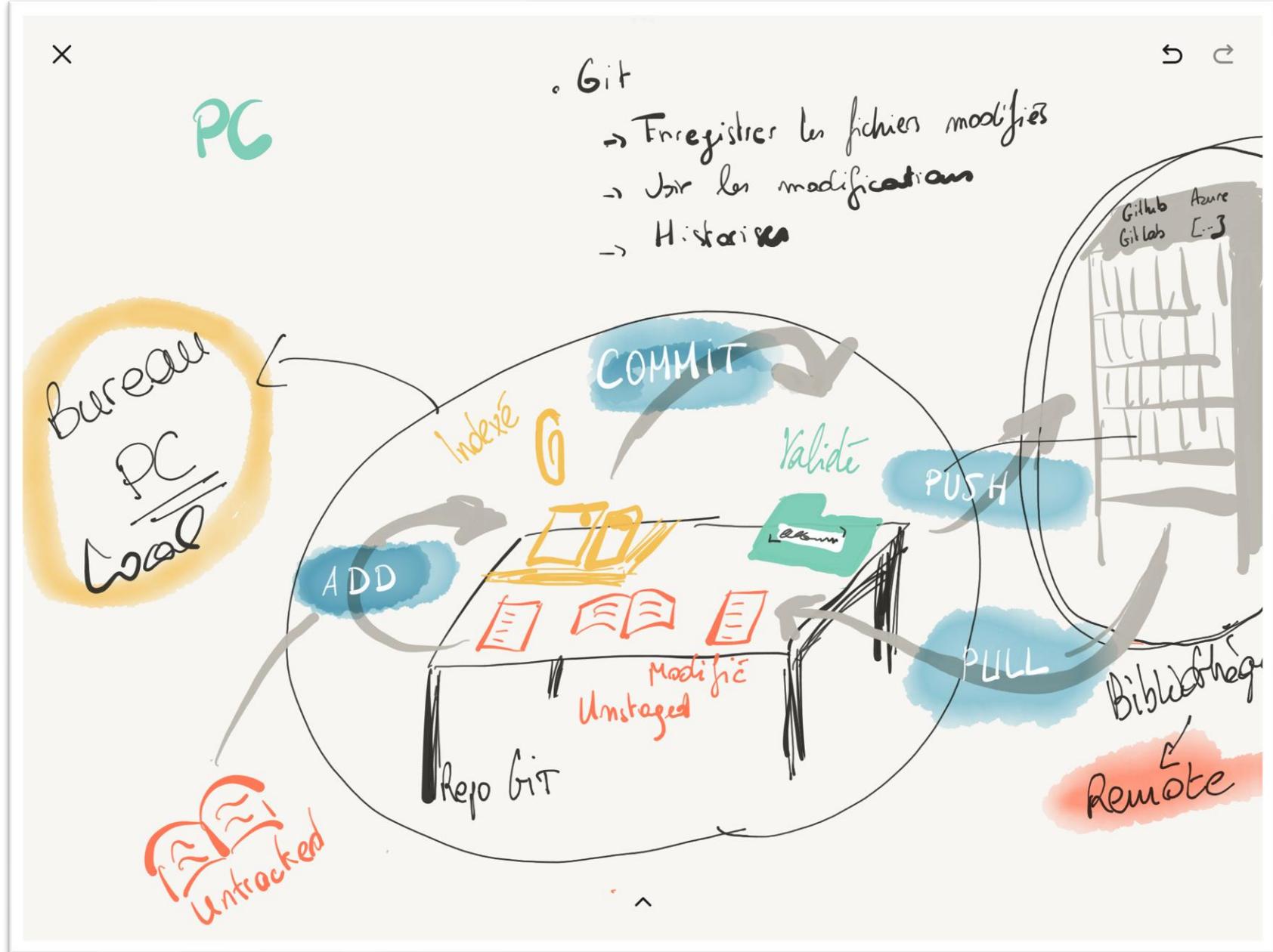
Structure d'un dépôt

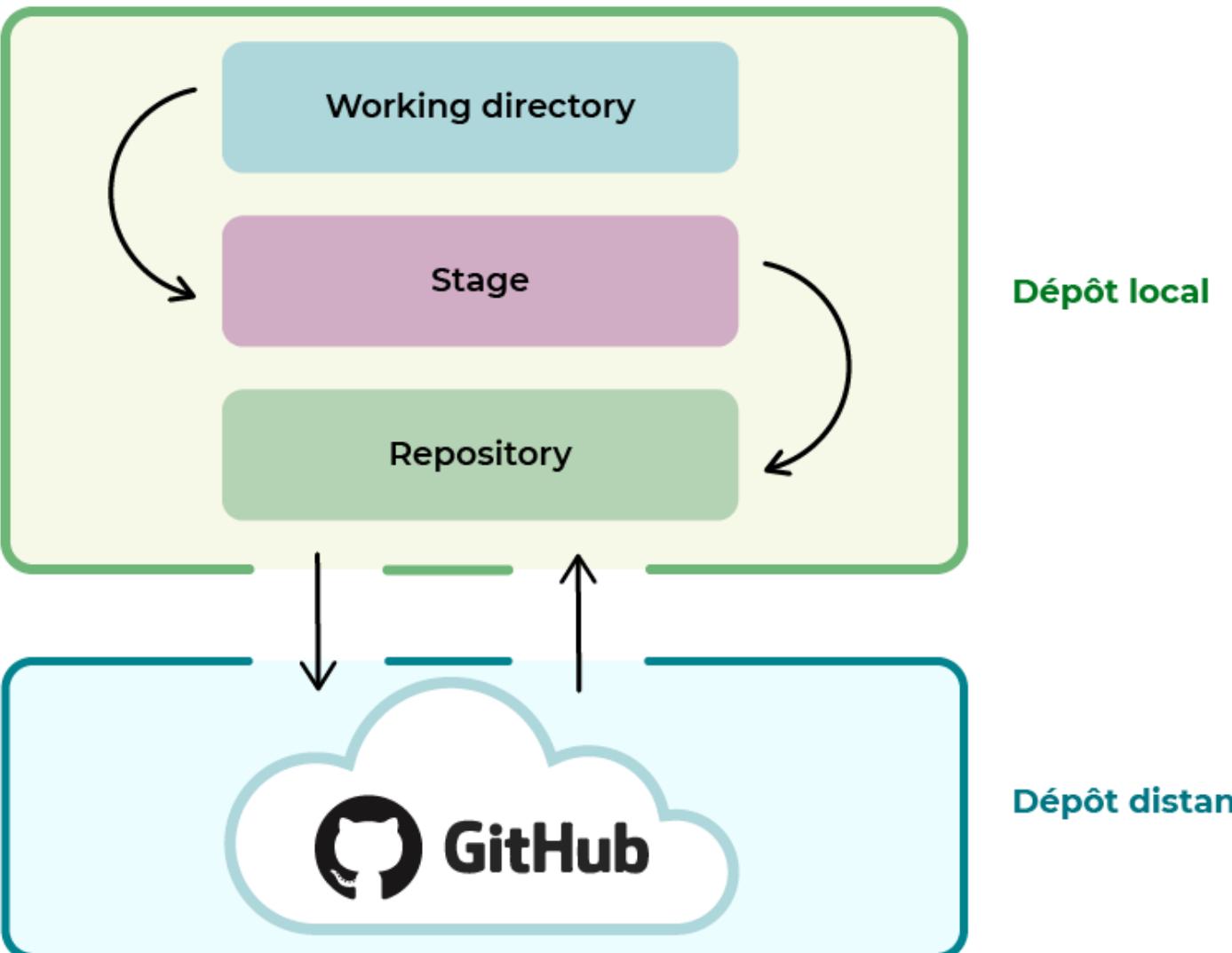
Structure



Description de cette structure

- Workspace
- Index
- Local repository
- Remote repository





<https://openclassrooms.com/fr/courses/7162856-gerez-du-code-avec-git-et-github/7165726-travaillez-depuis-votre-depot-local-git>

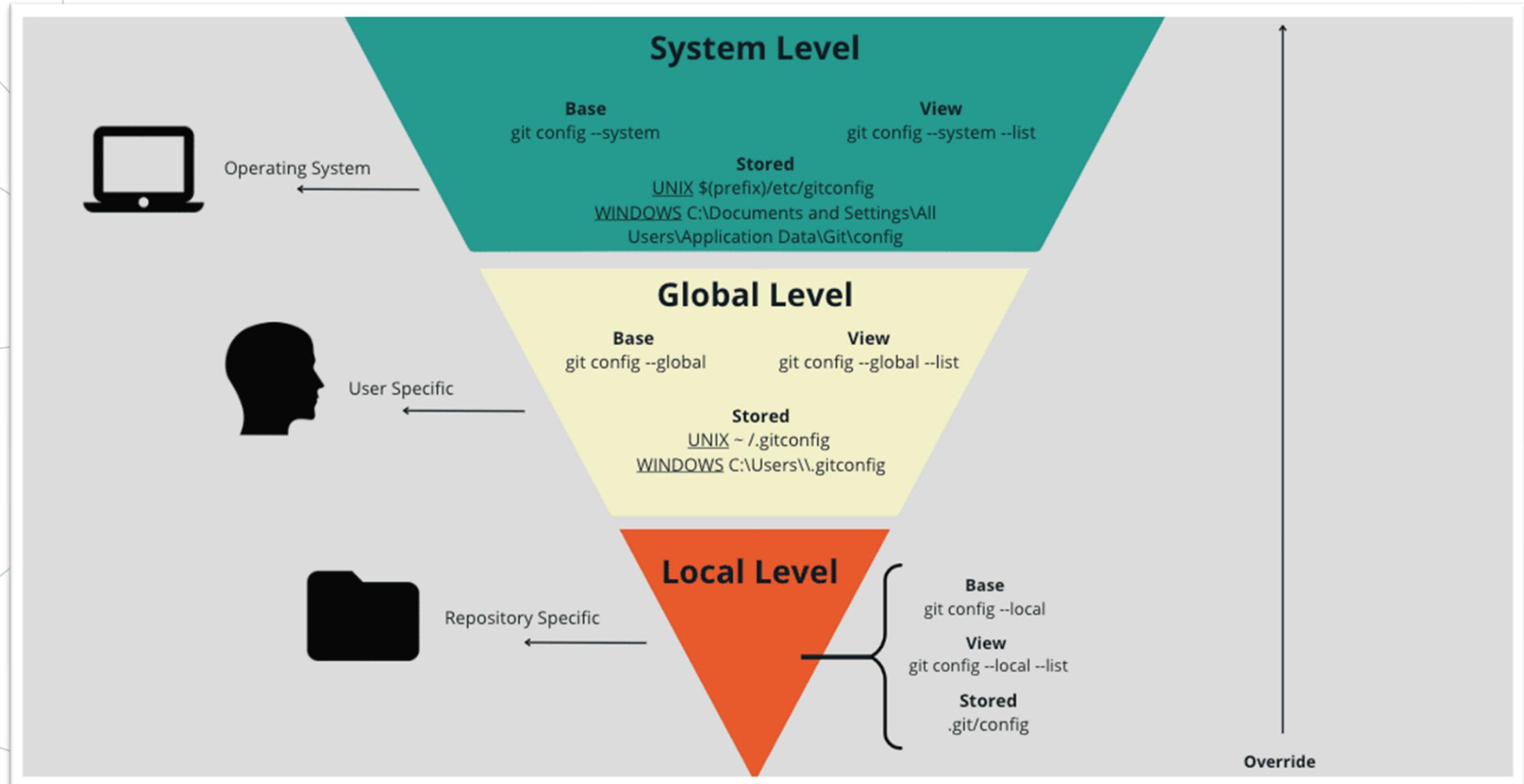


Installation & Configuration

Installation de git (windows)

- <https://git-scm.com/>
- Téléchargez l'installateur pour Windows à partir du site officiel de Git : Git for Windows.
- Exédez le fichier téléchargé.
- Suivez les instructions de l'assistant d'installation en laissant les options par défaut, sauf si vous avez une raison spécifique de les modifier.
- Sélectionnez l'éditeur de texte que vous préférez pour les commits (par exemple, Nano, Vim, ou Notepad++).

Configuration de git



Fondamental

- Configurer git pour votre utilisation

```
git config init.defaultbranch
```

```
git config --global user.name "Votre Nom"  
git config --global user.email "votre@email.com"
```



Git

Les commandes de base

Les commandes de base

- Créer un dépôt local
- Ajout et versioning de fichiers
- Supprimer, déplacer et renommer des fichiers
 - Analyser l'historique des révisions
- Obtenir des informations sur l'état de la copie de travail
 - Ignorer des fichiers

Créer un dépôt local

- Nouveau : init
- Existant : clone

Clone and Existing Repository

```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repository

```
$ git init
```

Voir l'état du dépôt

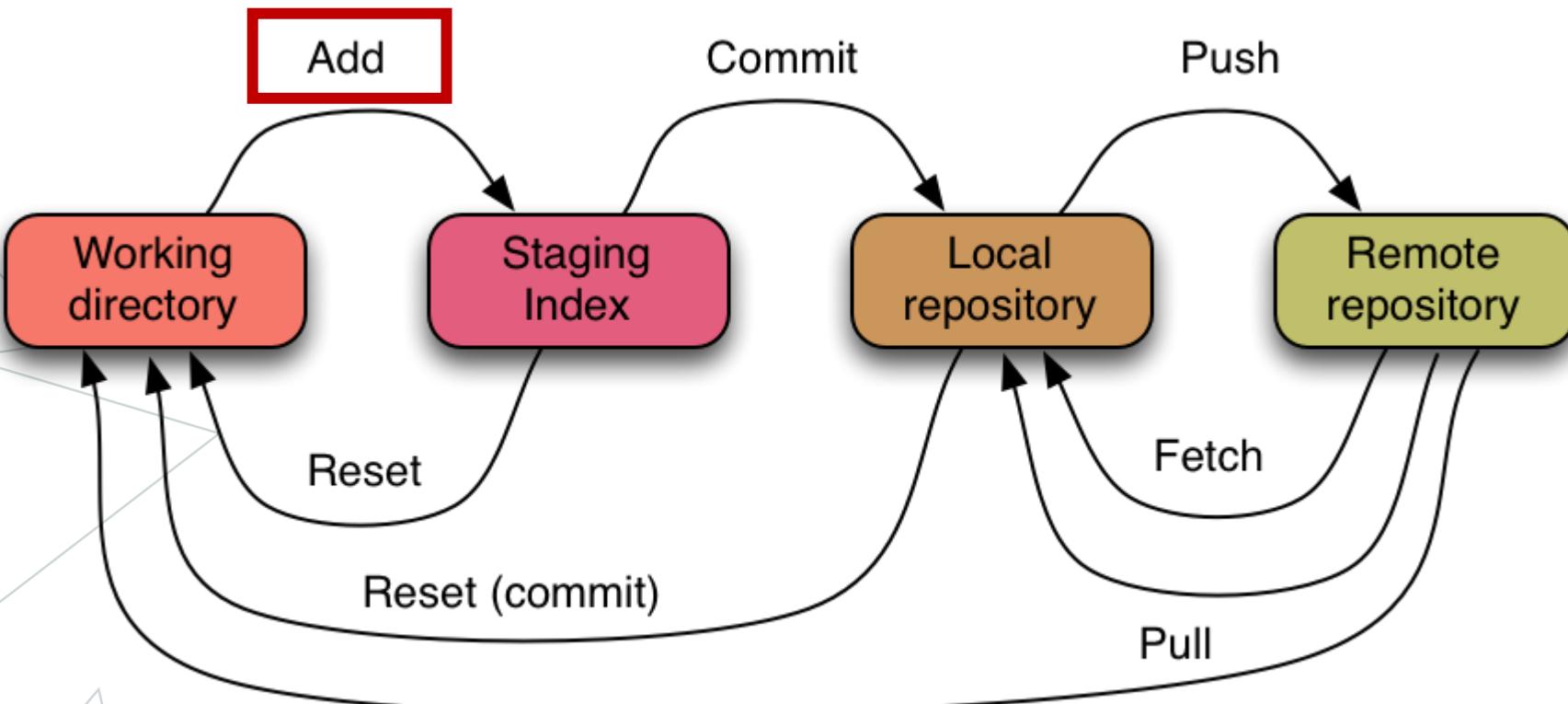
- git status

```
C:\Users\Thomas BDC\source\repos\NetCore092023>git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   NetCore092023.sln
    modified:   NetCore092023/Controllers/LiveController.cs
    modified:   NetCore092023/Data/ApplicationDbContext.cs
    modified:   NetCore092023/Data/Migrations/ApplicationDbContextModelSnapshot.cs
    modified:   NetCore092023/Models/LiveModel.cs
    modified:   NetCore092023/NetCore092023.csproj
    modified:   NetCore092023/Program.cs

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    LinqExercices/
    NetCore092023/Controllers/LiveEFController.cs
    NetCore092023/Data/Migrations/20230927091246_AddLives.Designer.cs
    NetCore092023/Data/Migrations/20230927091246_AddLives.cs
    NetCore092023/Data/Migrations/20230927094245_ADDNPARTICIPANT.Designer.cs
    NetCore092023/Data/Migrations/20230927094245_ADDNPARTICIPANT.cs
    NetCore092023/Views/LiveEF/

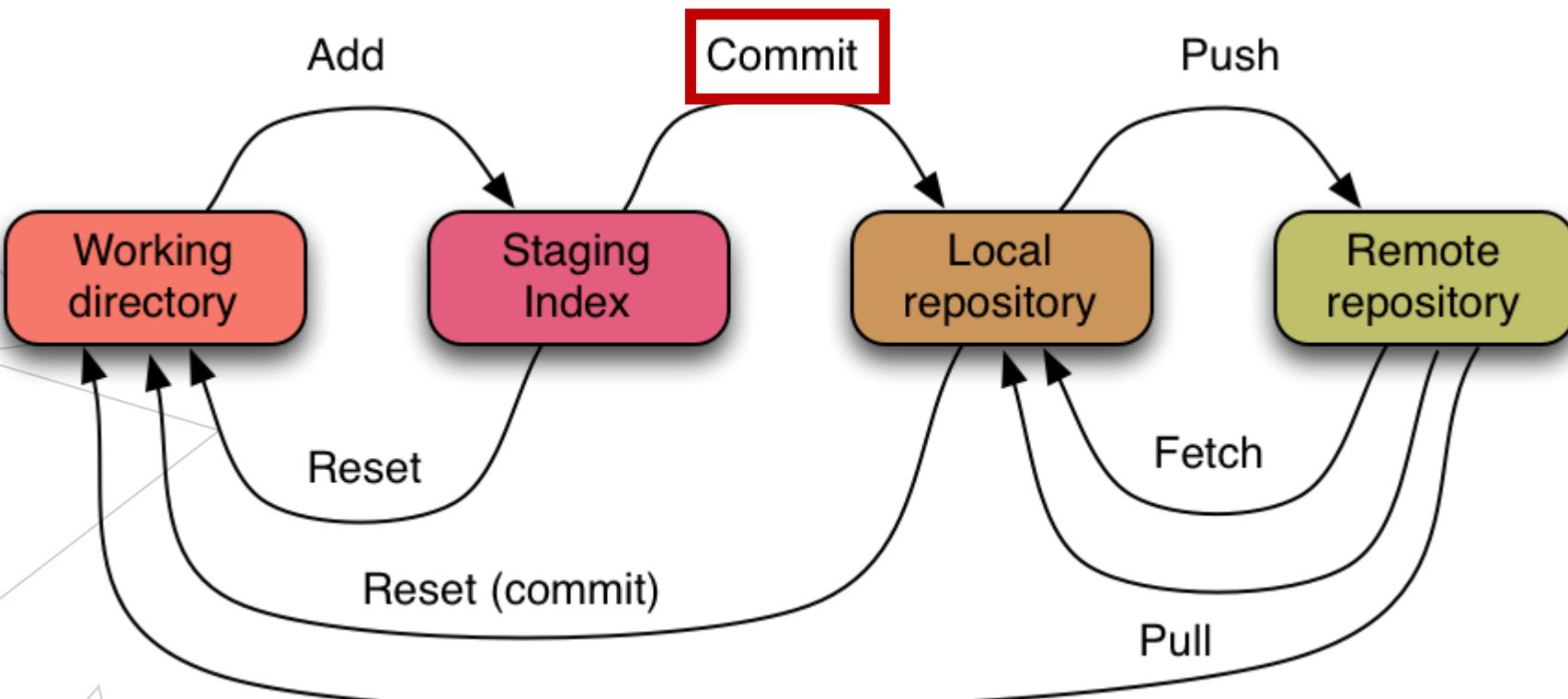
no changes added to commit (use "git add" and/or "git commit -a")
```



Ajout et versioning de fichiers

- git add <filename>
- git add .





Ajout et versioning de fichiers



```
$ git commit -m "Commit message here"
```



Ajout et versioning de fichiers

```
git commit
```

Commitez l'instantané stagé. Cette action lancera un éditeur de texte qui vous demandera un message de commit. Après avoir saisi un message, enregistrez le fichier et fermez l'éditeur pour créer le commit réel.

```
git commit -a
```

Commitez un instantané de tous les changements apportés au répertoire de travail. Celui-ci inclut uniquement les changements dans les fichiers suivis (ceux qui ont été ajoutés grâce à `git add` à un moment donné dans l'historique).

```
git commit -m "commit message"
```

Une commande de raccourci qui crée immédiatement un commit avec un message de commit transmis. Par défaut, `git commit` ouvre l'éditeur de texte configuré en local et invite à saisir un message de commit. Lorsque vous transmettez l'option `-m`, vous renoncez à l'invite d'éditeur de texte au profit d'un message contextuel.

```
git commit -am "commit message"
```

Une commande de raccourci pour les utilisateurs intensifs, qui combine les options `-a` et `-m`. Cette combinaison crée immédiatement un commit de tous les changements stagés et insère un message de commit contextuel.

```
git commit --amend
```

Cette option ajoute un autre niveau de fonctionnalité à la commande de commit. Lorsque vous transmettez cette option, le dernier commit sera modifié. Au lieu de créer un commit, les changements stagés sont ajoutés au commit précédent. Cette commande ouvre l'éditeur de texte configuré pour le système et invite à modifier le message de commit précédemment indiqué.

Historique

- git log

```
$ git log
commit 83af15b1b70b4e4c2d530b324939d4158daaf423 (HEAD -> main)
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 14:15:40 2023 +0100

    Ajout d'un gitignore

commit 0e1243303ead865b47ac1bab0863a57efb1c401a
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 13:55:46 2023 +0100

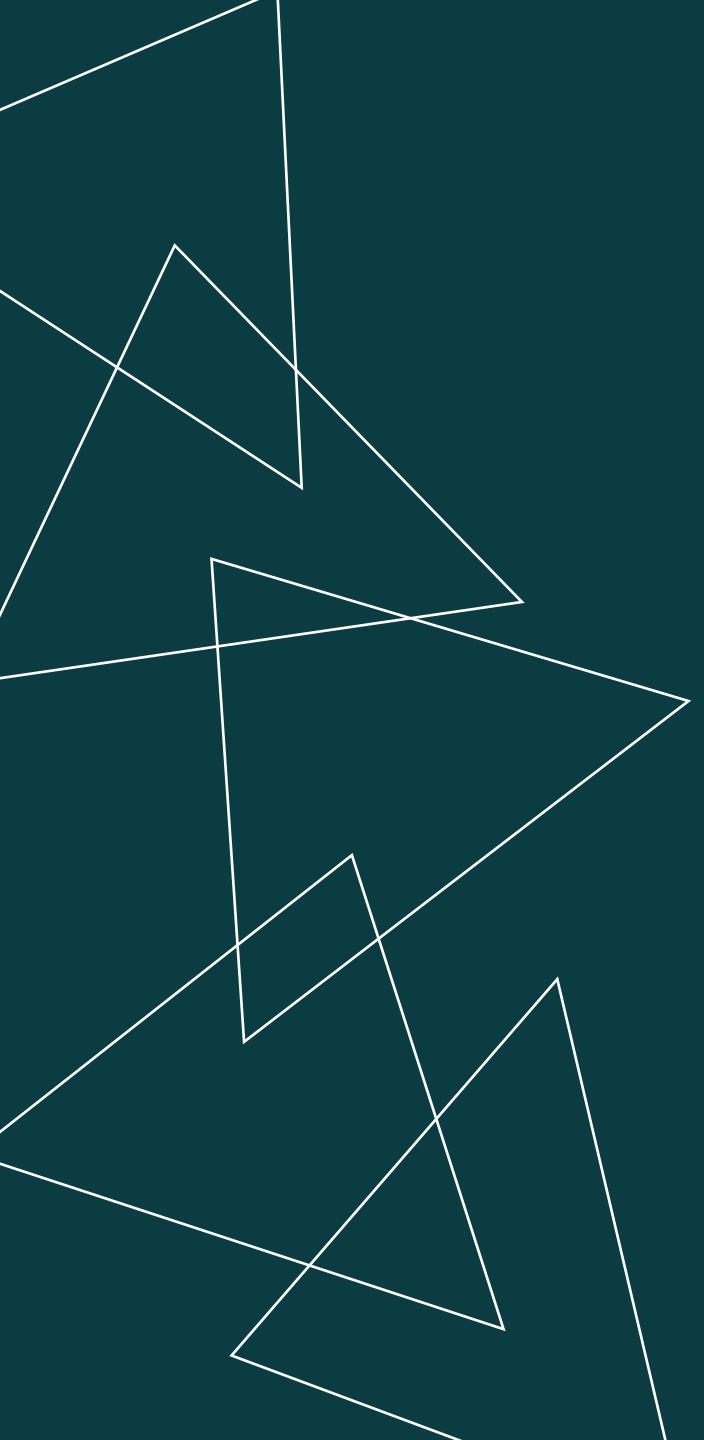
    Modification du fichier1

commit ca0ea33f286e819c12fe974e59675fff448bb4cd
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 13:49:41 2023 +0100

    Premier commit Fichier 1 et Fichier 2
```

Résumé

- On a le fonctionnement routinier
- Voyons donc quelques outils pratiques
 - Diff
 - Stash
 - Gitignore



Des aides pour développer

A faire

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble
$ git init
Initialized empty Git repository in C:/Users/Thomas BDC/source/repos/coursGitPrepaGrenoble/.git/

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$
echo "Contenu du fichier 1" > fichier1.txt
echo "Contenu du fichier 2" > fichier2.txt

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    fichier1.txt
    fichier2.txt

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

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git add .

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:  fichier1.txt
    new file:  Fichier2.txt
```

A faire

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git commit -m "Premier commit Fichier 1 et Fichier 2"
[main (root-commit) ca0ea33] Premier commit Fichier 1 et Fichier 2
 2 files changed, 2 insertions(+)
  create mode 100644 fichier1.txt
  create mode 100644 fichier2.txt
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ echo "Nouveau contenu pour fichier1.txt" > fichier1.txt

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
      modified:   fichier1.txt

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

- Mon fichier est modifié.
- C'est quoi qui est modifié ?

Git diff

- Le diff prend deux ensembles de données et génère une sortie révélant les changements entre eux. git diff est une commande Git multi-usage qui exécute une fonction de différenciation sur des sources de données Git. Ces sources de données peuvent être des commits, des branches, des fichiers, et bien plus. Ce document décrit les appels les plus courants de git diff et les modèles de workflow de différenciation. La commande git diff est souvent utilisée avec les commandes git status et git log pour analyser l'état actuel d'un dépôt Git.

Git diff

- `diff --git a/fichier1.txt b/fichier1.txt`: Cette ligne indique le chemin du fichier avant (`a/fichier1.txt`) et après (`b/fichier1.txt`) les modifications. Git affiche ces chemins pour indiquer les différences entre les versions.
 - `index ce64904..9b8c337 100644`: Ces informations sont les identifiants des versions du fichier avant et après. Ils identifient spécifiquement les états des fichiers. En général, ces identifiants sont des codes de hachage générés par Git pour chaque version du fichier.
 - `--- a/fichier1.txt`: Cela indique le début des différences pour le `fichier1.txt` avant les modifications.
 - `+++ b/fichier1.txt`: Cela indique le début des différences pour le `fichier1.txt` après les modifications.
 - `@@ -1 +1 @@`: Cette ligne indique la portée des lignes affectées par les modifications. Ici, `-1` fait référence à la première ligne du fichier d'origine, et `+1` à la première ligne du fichier modifié.
- `-Contenu du fichier 1`: Cette ligne indique le contenu de la première version du `fichier1.txt`.
- `+Nouveau contenu pour fichier1.txt`: Cette ligne indique le nouveau contenu introduit dans le `fichier1.txt`.

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git diff
diff --git a/fichier1.txt b/fichier1.txt
index ce64904..9b8c337 100644
--- a/fichier1.txt
+++ b/fichier1.txt
@@ -1 +1 @@
-Contenu du fichier 1
+Nouveau contenu pour fichier1.txt
```

Pas très clair

- Nos IDE savent afficher cette information de façon lisible

The screenshot shows a terminal window with the following content:

```
≡ fichier1.txt (Working Tree) M X
≡ fichier1.txt
1- Contenu du fichier 1
2 → 1+ Nouveau contenu pour fichier1.txt
2
```

The first line shows the file path and status. The second line shows the file name. The third line contains the old content of the file. The fourth line shows an arrow pointing to the new content, which is highlighted in green. The fifth line shows the new content.

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
on branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   fichier1.txt

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

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git add .

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git commit -m "Modification du fichier1"
[main 0e12433] Modification du fichier1
 1 file changed, 1 insertion(+), 1 deletion(-)

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ mkdir logs
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ cd logs

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/logs (main)
$ echo "error one line 94" > 21121997.txt
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/logs (main)
$ cd ..

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
on branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    logs/

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

- Après ça, modifier le fichier2.txt à la main

- Si je fais 'git add .', je vais tracker les fichiers de logs, or, je ne veux pas les tracker
- Je vais avoir ce message « untracked file », pour un fichier que je ne veux pas tracker

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
on branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   fichier2.txt

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

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



gitignore

Git considère chaque fichier de votre copie de travail comme l'une des trois choses suivantes :

- 1. suivi - un fichier qui a été précédemment mis en scène (staged) ou validé (commité);
- 2. non-suivi - un fichier qui n'a pas été mis en scène ou validé;
- 3. ignoré - un fichier que Git a explicitement été informé d'ignorer.

Les fichiers ignorés sont généralement des artefacts de construction (build artifacts) et des fichiers générés par la machine qui peuvent être dérivés de la source de votre dépôt ou qui ne devraient pas être validés. Quelques exemples courants sont :

- - caches de dépendances, tels que le contenu de /node_modules ou /packages
- - code compilé, comme les fichiers .o, .pyc et .class
- - répertoires de sortie de build, tels que /bin, /out ou /target
- - fichiers générés pendant l'exécution, tels que .log, .lock ou .tmp
- - fichiers système cachés, tels que .DS_Store ou Thumbs.db
- - fichiers de configuration personnelle de l'IDE, tels que .idea/workspace.xml

Les fichiers ignorés sont suivis dans un fichier spécial nommé `.gitignore` qui est placé à la racine de votre dépôt. Il n'y a pas de commande `git ignore` explicite : à la place, le fichier `.gitignore` doit être modifié et validé manuellement lorsque vous avez de nouveaux fichiers que vous souhaitez ignorer. Les fichiers `.gitignore` contiennent des motifs qui sont comparés aux noms de fichiers dans votre dépôt pour déterminer s'ils doivent être ignorés ou non.

- Notre dossier logs n'est plus traqué

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ echo "logs" > .gitignore

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
on branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   fichier2.txt

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

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

Templates

Code Blame 52 lines (41 loc) · 799 Bytes

```
1 # Cache and logs (Symfony2)
2 /app/cache/*
3 /app/logs/*
4 !app/cache/.gitkeep
5 !app/logs/.gitkeep
6
7 # Email spool folder
8 /app/spool/*
9
10 # Cache, session files and logs (Symfony3)
11 /var/cache/*
12 /var/logs/*
13 /var/sessions/*
14 !var/cache/.gitkeep
15 !var/logs/.gitkeep
16 !var/sessions/.gitkeep
17
18 # Logs (Symfony4)
19 /var/log/*
20 !var/log/.gitkeep
21
22 # Parameters
23 /app/config/parameters.yml
24 /app/config/parameters.ini
25
26 # Managed by Composer
```

<https://github.com/github/gitignore>

Syntaxe

Syntaxe	Description
`#`	Utilisé pour commenter une ligne dans le fichier `. gitignore `. Les lignes débutant par `#` sont ignorées et servent à ajouter des commentaires pour la lisibilité.
`**`	Correspond à tout caractère ou séquence de caractères. Par exemple, `*.txt` ignore tous les fichiers avec l'extension .txt.
`/logs`	Ignore le répertoire nommé "logs" à la racine du dépôt.
`logs/*`	Ignore tous les fichiers ou répertoires nommés "logs" à partir du répertoire où se trouve le fichier `. gitignore `.
`!important.txt`	N'ignore pas le fichier `important.txt`, même s'il est précédé par des règles d'ignorance générale.
`**/logs`	Ignore le répertoire "logs" quel que soit l'endroit où il se trouve dans la hiérarchie des répertoires.
`*.log`	Ignore tous les fichiers ayant l'extension `.log`.
`config*.txt`	Ignore tous les fichiers commençant par "config" et ayant l'extension `.txt`.
`logs/*.log`	Ignore tous les fichiers avec l'extension `.log` dans le répertoire "logs".
`/logs/debug/*`	Ignore le répertoire "debug" dans le répertoire "logs" situé à la racine du dépôt.
`/*.tmp`	Ignore tous les fichiers avec l'extension `.tmp` à la racine du dépôt.
`/vendor/**/*.dll`	Ignore tous les fichiers `*.dll` dans tous les sous-répertoires de `vendor`.

Continuons...

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
on branch main
changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   fichier2.txt

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

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

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git add .

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git commit -m "Ajout d'un gitignore"
[main 83af15b] Ajout d'un gitignore
 2 files changed, 2 insertions(+)
 create mode 100644 .gitignore
```

Continuons ...

- Mkdocs build

Building the site

That's looking good. You're ready to deploy the first pass of your `MkLorum` documentation. First build the documentation:

```
mkdocs build
```

This will create a new directory, named `site`. Take a look inside the directory:

```
$ ls site
about  fonts  index.html  license  search.html
css    img    js          mkdocs   sitemap.xml
```

Ignorer le tout

Notice that your source documentation has been output as two HTML files named `index.html` and `about/index.html`. You also have various other media that's been copied into the `site` directory as part of the documentation theme. You even have a `sitemap.xml` file and `mkdocs/search_index.json`.

If you're using source code control such as `git` you probably don't want to check your documentation builds into the repository. Add a line containing `site/` to your `.gitignore` file.

```
echo "site/" >> .gitignore
```

Ignorer des fichiers déjà tracké

To stop tracking a file, we must remove it from the index:

```
git rm --cached <file>
```

To remove a folder and all files in the folder recursively:

```
git rm -r --cached <folder>
```

The removal of the file from the head revision will happen on the next commit.

WARNING: While this will not remove the physical file from your local machine, it will remove the files from other developers' machines on their next `git pull`.

A faire

- Créer un dossier « web »
- Ajouter index.html dans ce dossier
 - Ajouter une ligne de code

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    web/
nothing added to commit but untracked files present (use "git add" to track)
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git add web

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   web/index.html
    new file:   web/test.html

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:  .gitignore
```

Attention

- On vous demande de faire une modification en urgence sur fichier1.txt
- Mais vous ne voulez pas perdre ce que vous avez déjà fait

Git stash



- 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.

Commande Git	Description
<code>`git stash save "message"` or `git stash`</code>	Enregistre les modifications locales dans un stash (cachette) avec un message descriptif. Ceci enregistre les modifications non validées de la copie de travail.
<code>`git stash list`</code>	Affiche la liste des stashes actuellement enregistrés.
<code>`git stash apply`</code>	Applique le dernier stash (le plus récent) sur la copie de travail sans le supprimer de la liste des stashes.
<code>`git stash apply stash@{n}`</code>	Applique un stash spécifique identifié par son index `n` sur la copie de travail sans le supprimer de la liste des stashes.
<code>`git stash pop`</code>	Applique et supprime le dernier stash (le plus récent) de la liste des stashes.
<code>`git stash pop stash@{n}`</code>	Applique et supprime un stash spécifique identifié par son index `n` de la liste des stashes.
<code>`git stash drop`</code>	Supprime le dernier stash (le plus récent) de la liste des stashes sans l'appliquer sur la copie de travail.
<code>`git stash drop stash@{n}`</code>	Supprime un stash spécifique identifié par son index `n` de la liste des stashes sans l'appliquer sur la copie de travail.
<code>`git stash clear`</code>	Supprime tous les stashes de la liste des stashes.

git stash	Enregistre les modifications locales dans un stash (cachette) avec un message descriptif. Ceci enregistre les modifications non validées de la copie de travail.
(unstash)	Applique et supprime le dernier stash (le plus récent) de la liste des stashes.

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   web/index.html
    new file:   web/test.html

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ dir
fichier1.txt  fichier2.txt  logs  web

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git stash
Saved working directory and index state WIP on main: 83af15b Ajout d'un gitignore

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ dir
fichier1.txt  fichier2.txt  logs

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git stash pop
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   web/index.html
    new file:   web/test.html

Dropped refs/stash@{0} (a531698441caa07272406bb720ec852d43a9a21c)

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   web/index.html
    new file:   web/test.html
```

Git

**Aller plus loin avec les
commandes basiques**

Allons plus loin...

- Filtrage avancé du log
- Staging de portions de fichiers
- Affichage de versions antérieures sans checkout
 - Utiliser git-reset

Approfondissons git log

Commande Git	Description
`git log`	Affiche l'historique des commits.
`git log --oneline`	Affiche chaque commit sur une seule ligne avec l'identifiant de hachage et le message.
`git log --graph`	Affiche un graphique ASCII représentant l'historique des branches et des fusions.
`git log --decorate`	Affiche les noms des branches et les tags à côté des commits.
`git log --stat`	Affiche des statistiques de modification pour chaque commit.
`git log --pretty=format:"..."`	Utilise un format spécifique pour l'affichage des commits. (Vous pouvez personnaliser le format.)
`git log --since="2 weeks ago"`	Affiche les commits effectués depuis les deux dernières semaines.
`git log --until="2023-01-01"`	Affiche les commits jusqu'à la date spécifiée.
`git log -n 5`	Limite l'affichage aux 5 derniers commits.
`git log --max-count=3`	Limite l'affichage à 3 commits.
`git log --author="John"`	Affiche les commits effectués par l'auteur spécifié.
`git log --committer="Jane"`	Affiche les commits commis par le committer spécifié.
`git log --grep="motif"`	Affiche les commits avec des messages contenant le motif spécifié.
`git log <chemin-du-fichier>`	Affiche l'historique des commits pour un fichier spécifique.
`git log <commit-A>..<commit-B>`	Affiche les commits entre deux références de commit.

Git log est mieux avec une extension

The screenshot shows a Git commit history interface with the following details:

Graphique	Description	Date	Auteur	Validation
	Cette révision n'existe pas encore, elle :	14 nov. 2023 04:00	*	
	main Ajout d'un gitignore	14 nov. 2023 02:11	Thomas BUREAU DU COLOMBIER 83af15b	
	Modification du fichier1	14 nov. 2023 01:55	Thomas BUREAU DU COLOMBIER < 0e12433	
	Premier commit Fichier 1 et Fichier 2	14 nov. 2023 01:49	Thomas BUREAU DU COLOMBIER < ca0ea33	

Below the commit history, there is a detailed view of the commit "Ajout d'un gitignore". The commit message is "Ajout d'un gitignore". The commit details are as follows:

Validation : 83af15b1b70b4e4c2d530b324939d4158daaf423 [83af15b]
Parents : 0e1243303e
Auteur : Thomas BUREAU DU COLOMBIER <sio.bureauducolumbier@gmail.com>
Date : mardi 14 novembre 2023 14:15:40
Auteur : Thomas BUREAU DU COLOMBIER

On the right side, there is a file viewer for ".gitignore" showing the content: "+ logs".

At the bottom left, there is a sidebar with two items: "fichier2.txt" and ".gitignore".

Continuons la pratique

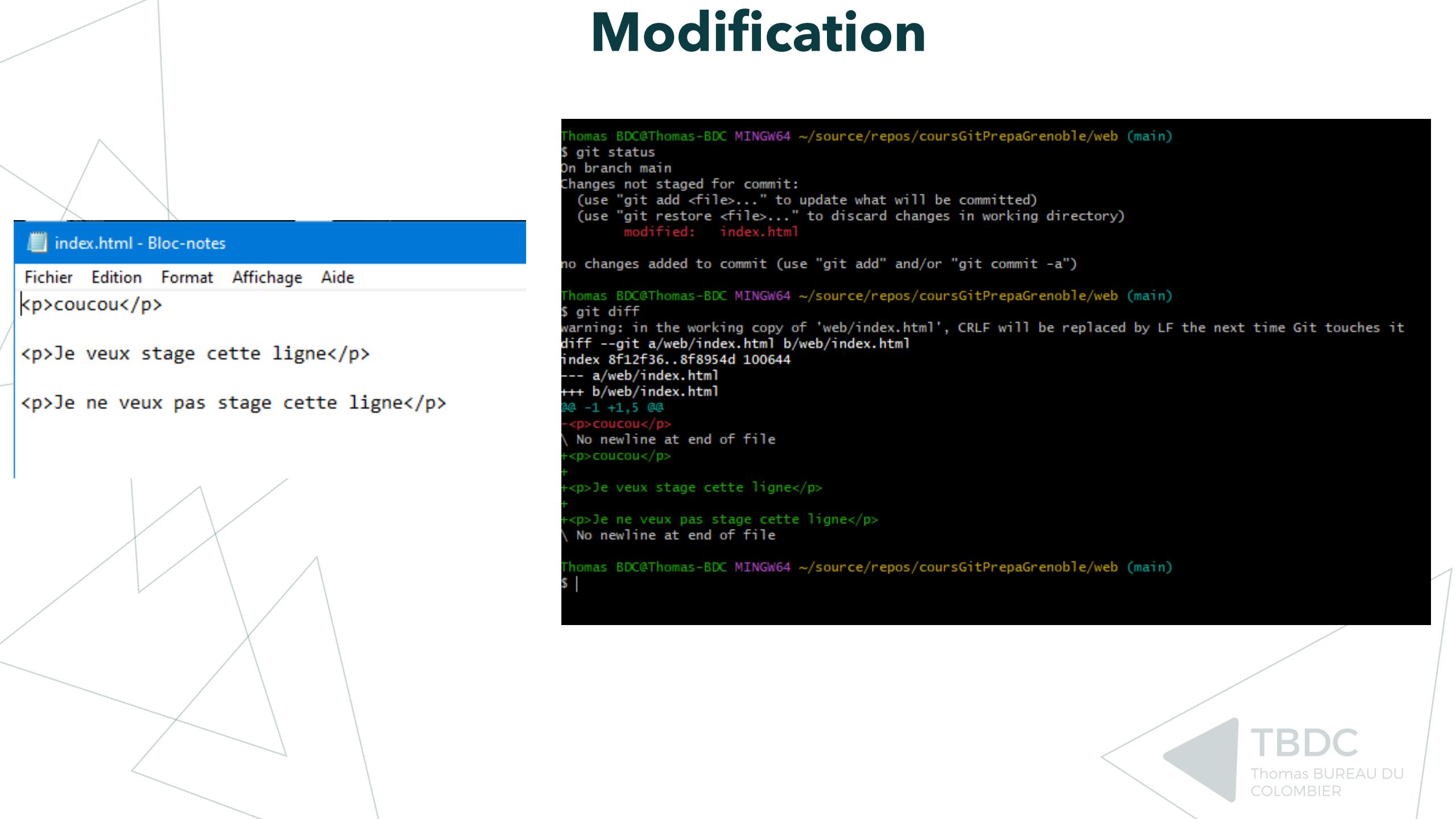
```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   web/index.html
    new file:   web/test.html

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git add .

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   web/index.html
    new file:   web/test.html

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git commit -m "Ajout des fichiers web"
[main 312acb5] Ajout des fichiers web
 2 files changed, 1 insertion(+)
 create mode 100644 web/index.html
 create mode 100644 web/test.html
```

Modification



index.html - Bloc-notes

Fichier Edition Format Affichage Aide

```
<p>coucou</p>

<p>Je veux stage cette ligne</p>

<p>Je ne veux pas stage cette ligne</p>
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   index.html

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

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git diff
warning: in the working copy of 'web/index.html', CRLF will be replaced by LF the next time Git touches it
diff --git a/web/index.html b/web/index.html
index 8f12f36..8f8954d 100644
--- a/web/index.html
+++ b/web/index.html
@@ -1 +1,5 @@
-<p>coucou</p>
\ No newline at end of file
+<p>coucou</p>
+
+<p>Je veux stage cette ligne</p>
+
+<p>Je ne veux pas stage cette ligne</p>
\ No newline at end of file

Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ |
```

Staging de portions de fichiers

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/cou
$ git add -p index.html
warning: in the working copy of 'web/index.html'
warning: in the working copy of 'web/index.html'
diff --git a/web/index.html b/web/index.html
index 8f12f36..8f8954d 100644
--- a/web/index.html
+++ b/web/index.html
@@ -1 +1,5 @@
-

coucou</p>
\ No newline at end of file
+<p>coucou</p>
+
+<p>Je veux stage cette ligne</p>
+
+<p>Je ne veux pas stage cette ligne</p>
\ No newline at end of file
(1/1) Stage this hunk [y,n,q,a,d,e,?]?


```

- `git add -p <filename>`
 - y : Stage (mettre en scène) cette partie des modifications.
 - n : Ne pas mettre en scène cette partie des modifications.
 - q : Quitter ; terminer le processus interactif.
 - a : Mettre en scène toutes les parties restantes automatiquement.
 - d : Ne pas mettre en scène cette partie ni les parties restantes.
 - / : Rechercher un motif dans le hunk (la partie modifiée).
 - j : Choisir la partie suivante sans poser de questions.
 - J : Choisir la partie suivante sans poser de questions, même si le hunk actuel n'est pas terminé.
 - g : Aller au début du hunk.
 - s : Diviser le hunk en parties plus petites.
 - e : Modifier le hunk manuellement.
 - ? : Afficher l'aide, qui répertorie toutes ces options.

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos
$ git add -p index.html
warning: in the working copy of 'web/index.html'
warning: in the working copy of 'web/index.html'
diff --git a/web/index.html b/web/index.html
index 8f12f36..8f8954d 100644
--- a/web/index.html
+++ b/web/index.html
@@ -1 +1,5 @@
-<p>coucou</p>
\ No newline at end of file
+<p>coucou</p>
+
+<p>Je veux stage cette ligne</p>
+
+<p>Je ne veux pas stage cette ligne</p>
\ No newline at end of file
(1/1) Stage this hunk [y,n,q,a,d,e,?] ? s
Sorry, cannot split this hunk
@@ -1 +1,5 @@
-<p>coucou</p>
\ No newline at end of file
+<p>coucou</p>
+
+<p>Je veux stage cette ligne</p>
+
+<p>Je ne veux pas stage cette ligne</p>
\ No newline at end of file
(1/1) Stage this hunk [y,n,q,a,d,e,?] ? e
```

```
Users > Thomas BDC > source > repos > coursGitPrepaGrenoble > .git > addp-hunk-edit.diff
# Manual hunk edit mode -- see bottom for a quick guide.
@@ -1 +1,5 @@
-<p>coucou</p>
\ No newline at end of file
+<p>coucou</p>
+
+<p>Je veux stage cette ligne</p>
+
+<p>Je ne veux pas stage cette ligne</p>
\ No newline at end of file
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
# If the patch applies cleanly, the edited hunk will immediately be marked for staging.
# If it does not apply cleanly, you will be given an opportunity to
# edit again. If all lines of the hunk are removed, then the edit is
# aborted and the hunk is left unchanged.
```

Supprimer cette ligne

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified: index.html

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

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git diff index.html
warning: in the working copy of 'web/index.html', CRLF will be replaced by LF the next time Git tou
diff --git a/web/index.html b/web/index.html
index 83307a7..8f8954d 100644
--- a/web/index.html
+++ b/web/index.html
@@ -1,3 +1,5 @@
<p>coucou</p>

<p>Je veux stage cette ligne</p>
+
+<p>Je ne veux pas stage cette ligne</p>
\ No newline at end of file
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git commit -m "commit une partie de fichier"
[main 23e0eca] commit une partie de fichier
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Résultat

- On a commit les lignes voulues
- La ligne non voulue reste en staged

The screenshot shows a Git commit interface with two main panes:

Left Pane (Staged Changes):

- Validation :** 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4 [23e0eca]
- Parents :** 312acb5c95
- Auteur :** Thomas BUREAU DU COLOMBIER <sio.bureauducolumbier@gmail.com>
- Date :** mercredi 15 novembre 2023 11:37:05
- Auteur :** Thomas BUREAU DU COLOMBIER
- commit une partie de fichier

Right Pane (Unstaged Changes):

web/index.html

		Bloc 1 : lignes 1-3	Restaurer le fragment
1		-<p>coucou</p> \ No newline at end of file	
2		+<p>coucou</p>	
3		+ +<p>Je veux stage cette ligne</p>	

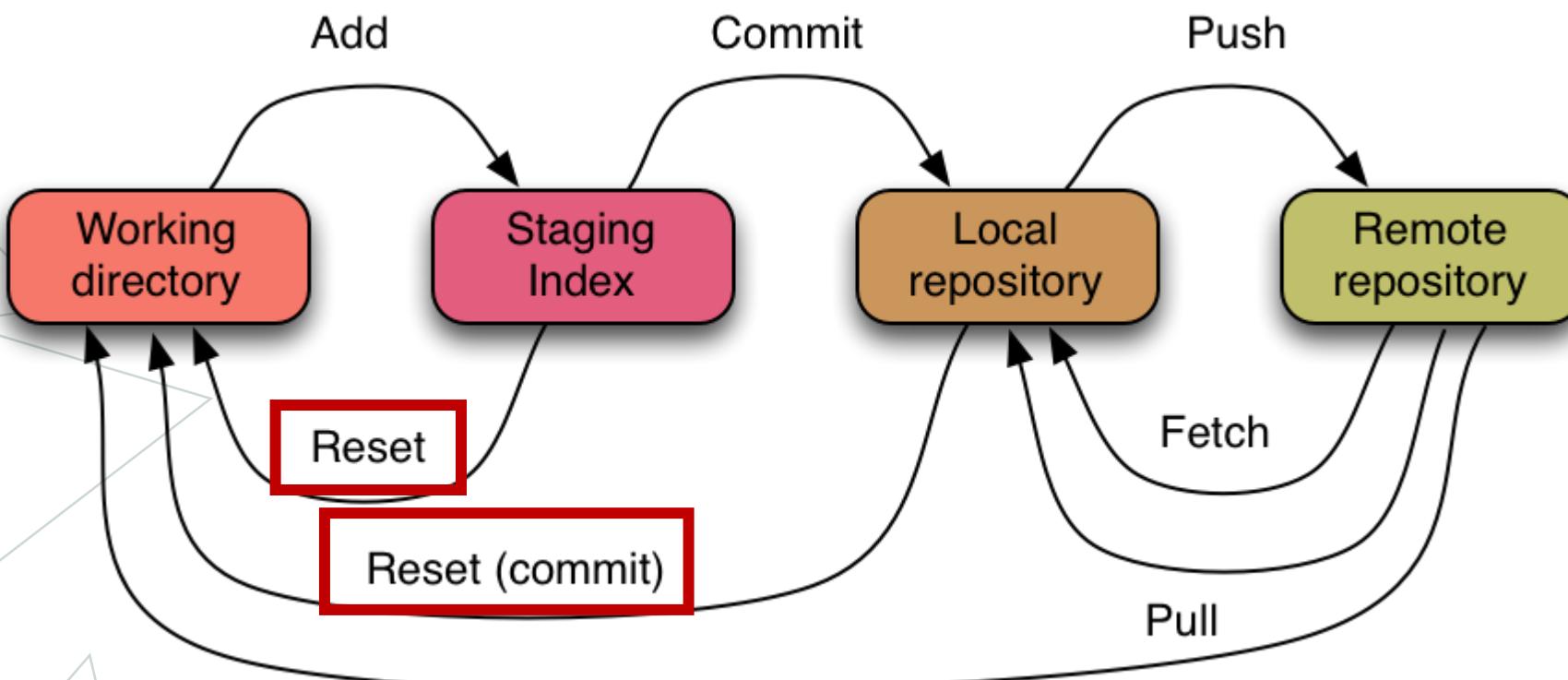
Bottom Panel (Index Status):

- Fichiers marqués dans l'index :** Tout désindexer | Désindexer la sélection
- Fichiers dans l'arbre de travail :** Tout indexer | Indexer la sélection
- web/index.html**

Bottom Right Detail:

		Bloc 1 : lignes 1-5	Marquer le fragment	Rejeter le bloc
1	1	...<p>coucou</p>		
2	2	...		
3	3	...<p>Je veux stage cette ligne</p>		
4		+		
5		+<p>Je ne veux pas stage cette ligne</p> \ No newline at end of file		

Annuler des changements



git restore

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   index.html

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

git restore

Les différentes commandes

Il existe trois commandes avec des noms similaires : git reset, git restore et git revert.

- **git revert** : Crée un nouveau commit qui annule les modifications apportées par d'autres commits.
- **git restore** : Restaure des fichiers dans l'arborescence de travail à partir soit de l'index soit d'un autre commit. Cette commande ne met pas à jour votre branche. La commande peut également être utilisée pour restaurer des fichiers dans l'index à partir d'un autre commit.
- **git reset** : Met à jour jour de votre branche. Il déplace l'extrémité afin d'ajouter ou de supprimer des commits de la branche. Cette opération modifie l'historique des commits.
 - git reset peut également être utilisé pour restaurer l'index, chevauchant ainsi l'action de git restore.

Commande	Cas d'utilisation	Fonctionnalité principale	Effet sur l'historique	Précautions à prendre
<code>'git revert'</code>	Annuler des changements spécifiques	Crée un nouveau commit annulant des changements	Crée de nouveaux commits	Utile pour annuler des modifications sans altérer l'historique, mais peut entraîner une multiplication des commits.
<code>'git restore'</code>	Restaurer des fichiers	Restaure des fichiers à partir du dernier commit ou d'un autre commit	Ne modifie pas l'historique	Restaure les fichiers sans altérer l'historique des commits. Utilisé pour remettre les fichiers dans l'état d'un commit spécifique.
<code>'git reset'</code>	Gestion des commits et de l'historique	Déplace la branche HEAD vers un commit spécifique	Modifie l'historique	Utiliser avec précaution pour éviter la perte de données. Utile pour réinitialiser l'historique des commits ou déplacer la branche.

déplacer la branche
ou ajouter des commits
à un commit existant

Restore (pour l'index et les changements en cours)

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   index.html
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git restore --staged index.html
```

Enlève le fichier de l'index

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   index.html

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

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git restore index.html
```

Annule les modifications

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git status
On branch main
nothing to commit, working tree clean
```

Annuler un commit

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git log
commit 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4 (HEAD -> main)
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Wed Nov 15 11:37:05 2023 +0100

    commit une partie de fichier

commit 312acb5c9521590cead0d68df802506c65cde382 ←
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 16:04:15 2023 +0100

    Ajout des fichiers web

commit 83af15b1b70b4e4c2d530b324939d4158daaf423
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 14:15:40 2023 +0100

    Ajout d'un gitignore

commit 0e1243303ead865b47ac1bab0863a57efb1c401a
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 13:55:46 2023 +0100

    Modification du fichier1

commit ca0ea33f286e819c12fe974e59675fff448bb4cd
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 13:49:41 2023 +0100

    Premier commit Fichier 1 et Fichier 2
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble
$ git revert 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4
hint: Waiting for your editor to close the file... |
```

```
> Users > Thomas BDC > source > repos > coursGitPrepaGrenoble > .git > COMMIT_EDITMSG
1  Revert "commit une partie de fichier"
2
3  This reverts commit 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4.
4
5  # Please enter the commit message for your changes. Lines starting
6  # with '#' will be ignored, and an empty message aborts the commit
7  #
8  # On branch main
9  # Changes to be committed:
10 #   modified:   web/index.html
11 #
12
13 Annulation du commit
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git revert 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4
[main 958841d] Revert "commit une partie de fichier"
 1 file changed, 1 insertion(+), 3 deletions(-)
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble/web (main)
$ git log
commit 958841dd6e974e7f8039d05a38ad0df505cf6bc1 (HEAD -> main)
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Wed Nov 15 14:02:15 2023 +0100

    Revert "commit une partie de fichier"

    This reverts commit 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4.

    Annulation du commit

commit 23e0ecac9f6f209250e4177d2dc550ca31fdf4d4
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Wed Nov 15 11:37:05 2023 +0100

    commit une partie de fichier

commit 312acb5c9521590cead0d68df802506c65cde382
Author: Thomas BUREAU DU COLOMBIER <sio.bureau du colombier@gmail.com>
Date:   Tue Nov 14 16:04:15 2023 +0100

    Ajout des fichiers web
```



Git

BRANCHES, FUSION ET CONFLIT

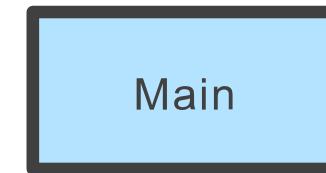
Les branches

- Fonctionnalité majeure de Git
- Actuellement, nous travaillons sur une branche.

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

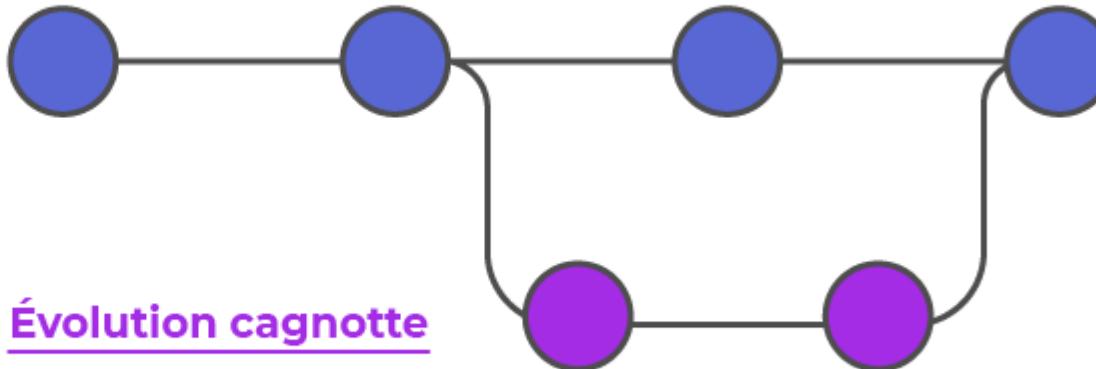


Pour rappel, la société GitHub a opéré un changement de terme de “master” à “main” en octobre 2020. Vous serez amené à travailler avec ces deux termes.



A quoi ça sert

Application en production



Évolution cagnotte

git branch

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git branch
* main
```

1. **Lister les branches :** Si vous exécutez simplement `git branch`, cela affichera la liste de toutes les branches présentes dans votre dépôt Git. L'étoile (*) indique la branche actuellement active.

bash

 Copy code

```
git branch
```

2. **Créer une nouvelle branche :** Pour créer une nouvelle branche à partir de votre branche actuelle, utilisez la commande suivante :

bash

 Copy code

```
git branch <nom_de_la_nouvelle_branche>
```

Par exemple, pour créer une nouvelle branche appelée "feature/nouvelle-fonctionnalite" à partir de votre branche actuelle :

bash

 Copy code

```
git branch feature/nouvelle-fonctionnalite
```



3. **Changer de branche**: Pour basculer vers une autre branche, vous pouvez utiliser la commande `git checkout` suivie du nom de la branche :

bash

 Copy code

```
git checkout <nom_de_la_branche>
```

À partir de Git version 2.23, vous pouvez également utiliser `git switch` pour changer de branche :

bash

 Copy code

```
git switch <nom_de_la_branche>
```

A faire

- Créez une branche, et ajoutez un fichier dans cette branche
 - Faire un commit de ces modifications
- Analyser ce qui se passe sur l'explorateur de fichier windows lorsqu'on change de branche sur git

4. **Supprimer une branche** : Pour supprimer une branche, vous pouvez utiliser la commande suivante :

bash

 Copy code

```
git branch -d <nom_de_la_branche>
```

Si vous souhaitez supprimer une branche même si des modifications n'ont pas été fusionnées, vous pouvez utiliser ` -D` :

bash

 Copy code

```
git branch -D <nom_de_la_branche>
```



5. **Renommer une branche** : Pour renommer une branche, vous pouvez la renommer en utilisant ` -m` avec `git branch` :

bash

 Copy code

```
git branch -m <nouveau_nom_de_branche>
```

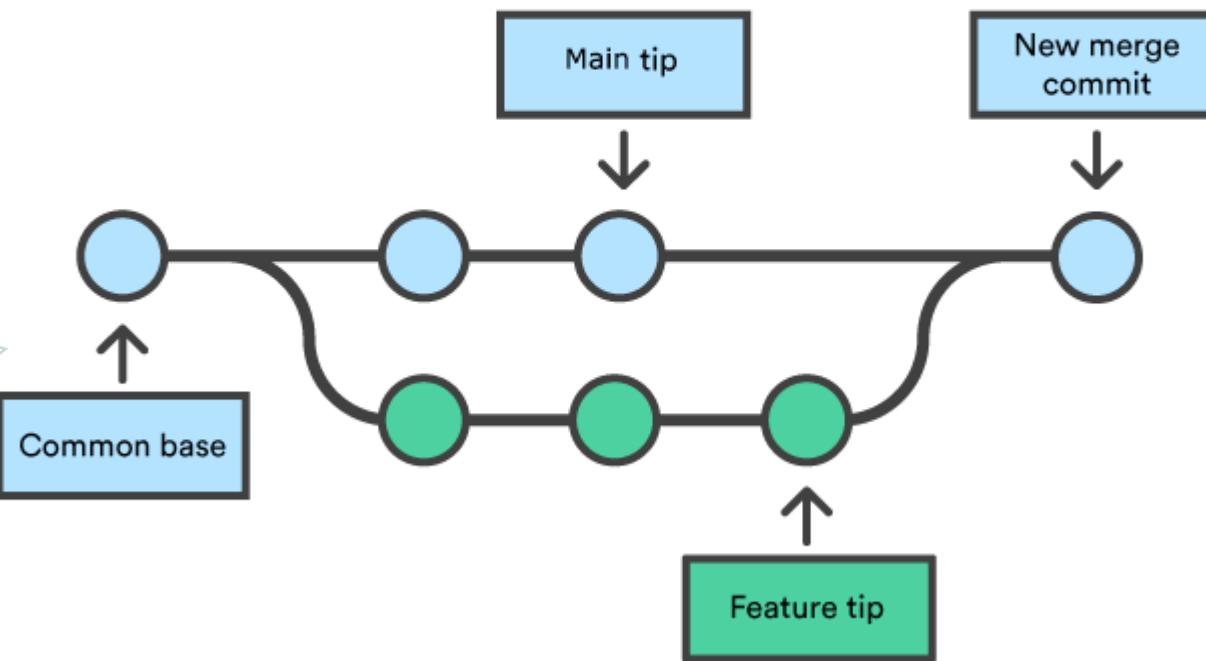
Les raisons

- 1. Isolation des fonctionnalités** : Les branches permettent à plusieurs développeurs de travailler sur des fonctionnalités distinctes en parallèle sans interférer les unes avec les autres. Chaque fonctionnalité peut être développée dans sa propre branche et fusionnée ultérieurement dans la branche principale.
- 2. Correction de bogues (bugfix)** : Les branches sont souvent utilisées pour isoler et résoudre des bogues spécifiques. Les développeurs peuvent créer une branche dédiée pour travailler sur le correctif du bogue sans affecter le reste du code.
- 3. Tests expérimentaux** : Les branches peuvent être utilisées pour expérimenter de nouvelles idées ou fonctionnalités sans risquer la version principale du code. Une fois que ces idées sont validées, elles peuvent être fusionnées dans la branche principale.
- 4. Versionnement du logiciel** : Les branches sont également utiles pour gérer les différentes versions d'un logiciel. Chaque version majeure ou mineure peut être gérée dans sa propre branche, permettant aux développeurs de travailler sur des mises à jour spécifiques à cette version.

Merge

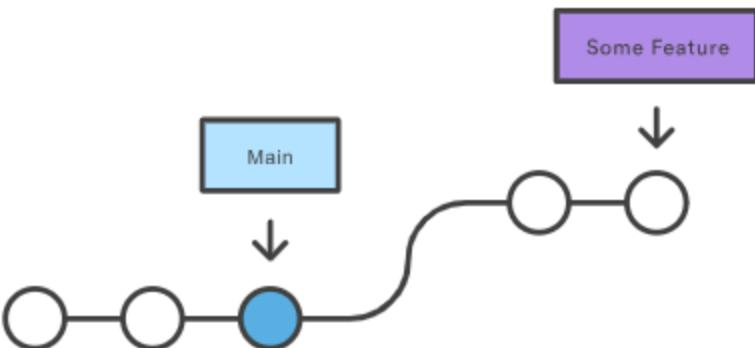
Merge ou Fusion

- Un "merge" (fusion en français) en Git est un processus qui combine les modifications de deux branches distinctes pour créer une nouvelle branche ou mettre à jour une branche existante. Cela permet d'intégrer les changements effectués sur une branche avec une autre branche.

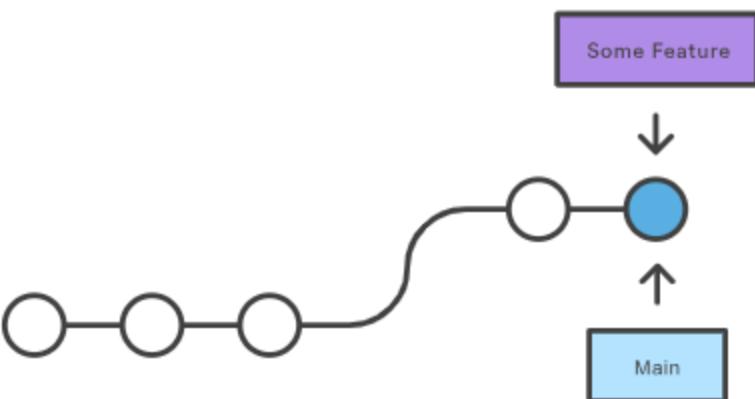


Fast forward

Before Merging



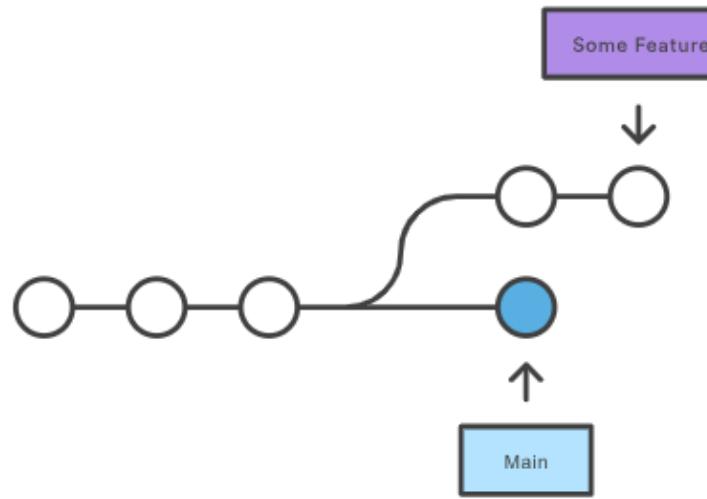
After a Fast-Forward Merge



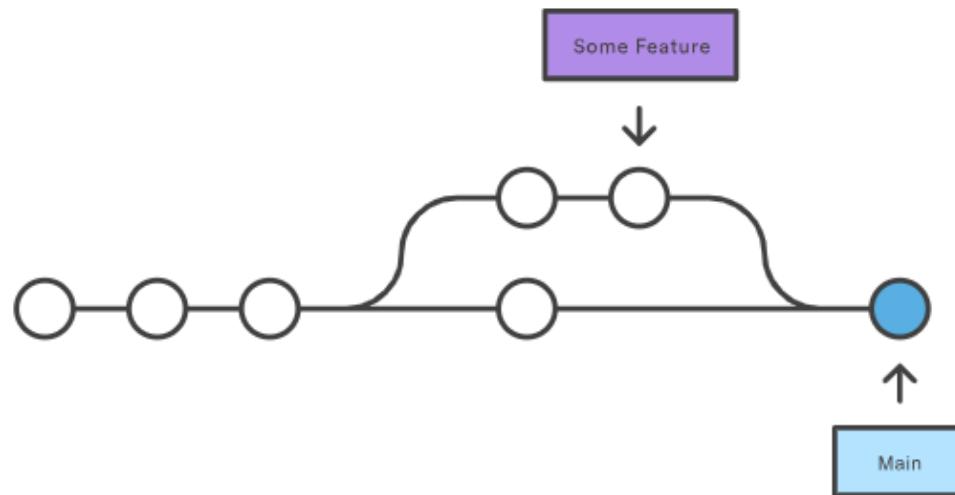
- Merge "Fast-forward" : Ce type de merge intervient lorsque les branches n'ont pas divergé. Git peut simplement déplacer le pointeur de la branche vers l'avant car l'historique des commits peut être tracé en ligne droite. Cela se produit souvent lorsqu'une branche a été créée après la branche parente et que aucun autre commit n'a été ajouté à la branche parente depuis.

A faire

```
# Start a new feature
git checkout -b new-feature main
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Merge in the new-feature branch
git checkout main
git merge new-feature
git branch -d new-feature
```



After a 3-way Merge



- Merge "Non Fast-forward": Ce type de merge intervient lorsque Git ne peut pas simplement déplacer le pointeur de la branche. Cela se produit lorsque les branches ont des commits distincts, et Git doit créer un nouveau commit de fusion pour intégrer les changements des deux branches.

A faire

```
Start a new feature
git checkout -b new-feature main
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Develop the main branch
git checkout main
# Edit some files
git add <file>
git commit -m "Make some super-stable changes to main"
# Merge in the new-feature branch
git merge new-feature
git branch -d new-feature
```

Conflits

A faire

- Créer une branche « fix-bug »
- Modifier le fichier index.html à la ligne 1
 - Faire un commit
- Checkout sur la branche main
- Modifier le fichier index.html à la ligne 1
 - Faire un commit
- Essayer de merge
- Que se passe-t-il ?

Créer un conflit

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git branch fix-bug
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git add .
```

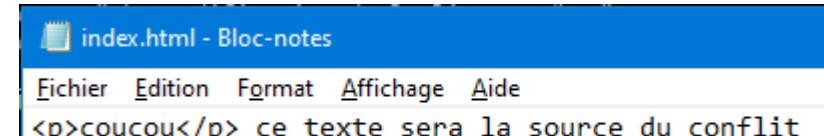
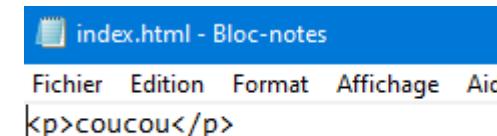
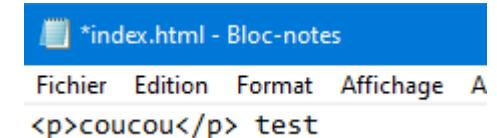
```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git commit -m "commit sur main"
[main 3f69bea] commit sur main
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (main)
$ git checkout fix-bug
Switched to branch 'fix-bug'
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (fix-bug)
$ git add web/index.html
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (fix-bug)
$ git commit -m "commit pour le merge"
[fix-bug 7d1c39b] commit pour le merge
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (fix-bug)
$ git merge main
Auto-merging web/index.html
CONFLICT (content): Merge conflict in web/index.html
Automatic merge failed; fix conflicts and then commit the result.
```





index.html - Bloc-notes

Fichier Edition Format Affichage Aide

<<<<< HEAD

<p>coucou</p> ce texte sera la source du conflit

=====

<p>coucou</p> test

>>>>> main

Lorsque Git rencontre un conflit au cours d'un merge, il éditera le contenu des fichiers affectés avec des indicateurs visuels qui marquent les deux côtés du contenu en conflit. Ces marqueurs visuels sont les suivants : <<<<<, ===== et >>>>>. Il s'avère utile de rechercher ces indicateurs dans un projet au cours d'un merge pour savoir où des conflits doivent être résolus.

```
here is some content not affected by the conflict
<<<<< main
this is conflicted text from main
=====
this is conflicted text from feature branch
>>>>> feature branch;
```

En principe, le contenu situé avant le marqueur ===== représente la branche cible et la partie après ce marqueur représente la branche mergée.



Lorsque vous avez identifié les sections en conflit, vous pouvez faire le nécessaire pour résoudre le problème. Lorsque vous êtes prêt à terminer le merge, il vous suffit d'exécuter `git add` sur le(s) fichier(s) en conflit pour signaler à Git que le problème est résolu. Ensuite, lancez la commande `git commit` normalement pour générer le commit de merge. Le processus est exactement le même que pour commiter un instantané classique. Ainsi, les développeurs n'auront aucune difficulté à gérer leurs merges.

Notez que des conflits de merge se produiront également en cas de merge à trois branches. Il n'est pas possible d'avoir des changements en conflit dans un fast-forward merge.

A faire

- Corriger le conflit
- Merge enfin sur main
- Supprimer la branche

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGreco
(fix-bug|MERGING)
$ git status
On branch fix-bug
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:  web/index.html

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

index.html > .

Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes

<<<<< HEAD (Current Change)

```
<p>coucou</p> ce texte sera la source du conflit
```

=====

```
<p>coucou</p> test
```

>>>>> main (Incoming Change)

index.html M X

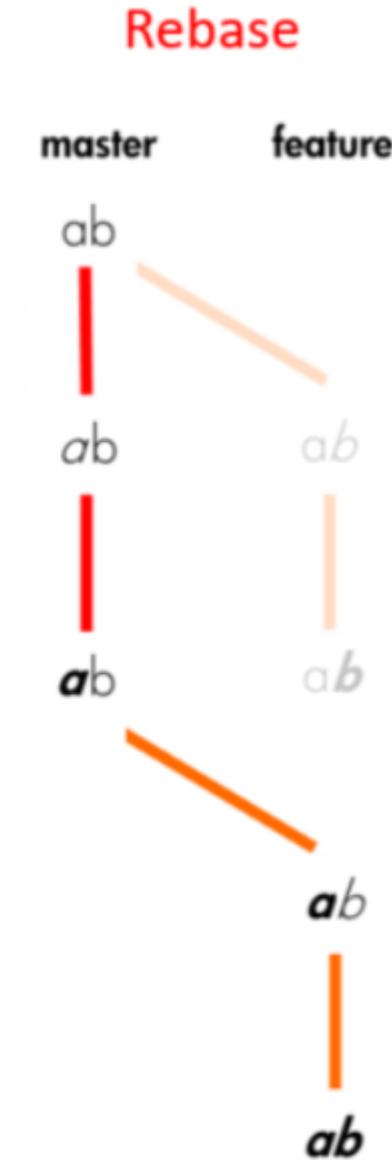
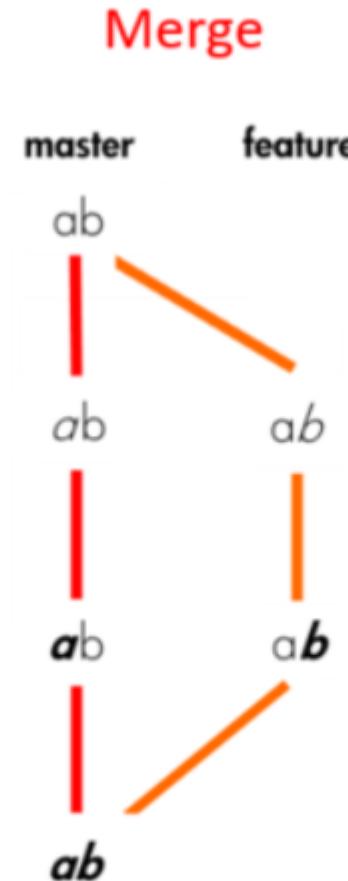
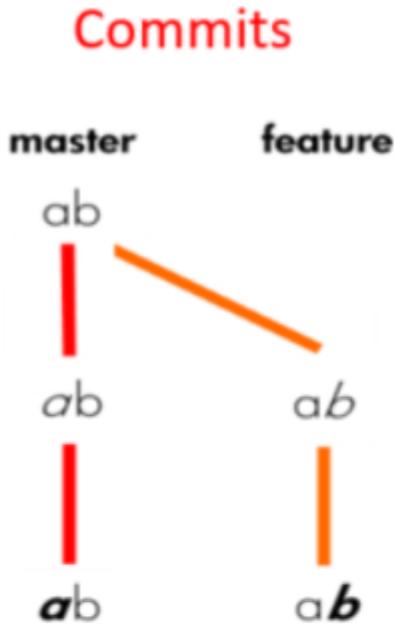
web > index.html > p

```
1 <p>coucou</p> ce texte sera la source du conflit
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrep  
(fix-bug|MERGING)  
$ git add web/index.html  
  
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrep  
(fix-bug|MERGING)  
$ git status  
On branch fix-bug  
All conflicts fixed but you are still merging.  
  (use "git commit" to conclude merge)  
  
Changes to be committed:  
  modified:   web/index.html
```

Merge et Rebase

Merge VS Rebase



Choix entre merge et rebase

- Préférences personnelles : Le choix entre merge et rebase peut dépendre des préférences de l'équipe ou du développeur.
- Collaboration : En général, pour des branches partagées avec d'autres développeurs, le merge est souvent préféré car il préserve l'historique et évite les modifications potentielles de l'historique partagé. Le rebase est plus approprié pour des branches personnelles où l'historique linéaire est privilégié.
- Résolution des conflits : La résolution des conflits peut être plus simple avec merge car les commits originaux sont préservés. Le rebase peut entraîner des conflits plus importants en raison de la réécriture de l'historique.

Exercice : Diff entre merge et rebase

- Initialisez un dépôt Git
- Créez un Commit sur main InitialCommit avec un fichier texte
- Créez une branche **feature-branch**.
- Ajoutez **2 commits** sur la branche **feature-branch**.
- Ajoutez **1 commit** sur la branche **main**
- Faites un **merge** et observez l'historique.
- **Annulez** les 4 derniers commits, et supprimer la branche feature-branch
- Refaites la même opération avec un **rebase** et comparez les résultats.

Astuce : Utilisez `git log --oneline --graph --all` pour analyser l'impact de chaque méthode sur l'historique Git.

Solution

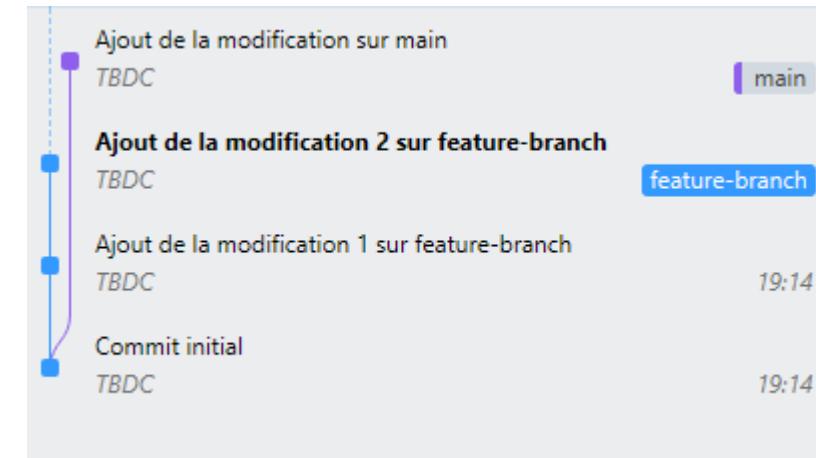
```
mkdir git-merge-vs-rebase  
cd git-merge-vs-rebase  
git init
```

```
echo "Ligne initiale" > fichier.txt  
git add fichier.txt  
git commit -m "Commit initial"
```

```
git checkout -b feature-branch  
echo "Modification 1 sur feature-branch" >> fichier.txt  
git commit -am "Ajout de la modification 1 sur feature-branch"  
echo "Modification 2 sur feature-branch" >> fichier.txt  
git commit -am "Ajout de la modification 2 sur feature-branch"
```

```
git checkout main  
echo "Modification sur main" >> fichier.txt  
git commit -am "Ajout de la modification sur main"
```

```
* ed96454 (HEAD -> main) Ajout de la modification sur main  
| * 96e2b19 (feature-branch) Ajout de la modification 2 sur feature-branch  
| * 3f4f2a8 Ajout de la modification 1 sur feature-branch  
|/  
* 3da0d41 Commit initial
```



Merge

Git switch feature-branch
Git merge main
Git switch main
Git merge feature-branch

```
* d8aa499 (HEAD -> main, feature-branch) Merge branch 'main' into feature-branch
|\ 
| * ed96454 Ajout de la modification sur main
| * 96e2b19 Ajout de la modification 2 sur feature-branch
| * 3f4f2a8 Ajout de la modification 1 sur feature-branch
|/
* 3da0d41 Commit initial
```

Annuler et supprimer la branche

Git switch main

Git reset --hard HEAD~1

Git branch -D feature-branch

Rebase

Git switch feature-branch
Git rebase main

Git switch main
Git rebase feature-branch

```
$ git log --oneline --graph --all
* 38a8a1b (HEAD -> main, feature-branch) Ajout de la modification 2 sur feature-
branch
* 59088a1 Ajout de la modification 1 sur feature-branch
* 845ed01 Ajout de la modification sur main
* 3da0d41 Commit initial
```

Différences

```
* d8aa499 (HEAD -> main, feature-branch) Merge branch 'main' into feature-branch
| \
| * ed96454 Ajout de la modification sur main
| * 96e2b19 Ajout de la modification 2 sur feature-branch
| * 3f4f2a8 Ajout de la modification 1 sur feature-branch
|/
* 3da0d41 Commit initial
```

```
c vs merge git merge vs rebase (matrix)
$ git log --oneline --graph --all
* 38a8a1b (HEAD -> main, feature-branch) Ajout de la modification 2 sur feature-branch
* 59088a1 Ajout de la modification 1 sur feature-branch
* 845ed01 Ajout de la modification sur main
* 3da0d41 Commit initial
```

Exemple avec le pull

Sur le serveur.

```
A---B---C---D---E ← master
```

Sur votre machine.

```
origin/master  
/  
A---B---C---F ← master
```

Après un `git pull`.

```
D---E ← origin/master  
/   \  
A---B---C---F---G ← master
```

Vous pourrez alors envoyer votre travail sur le serveur avec un `git push`.

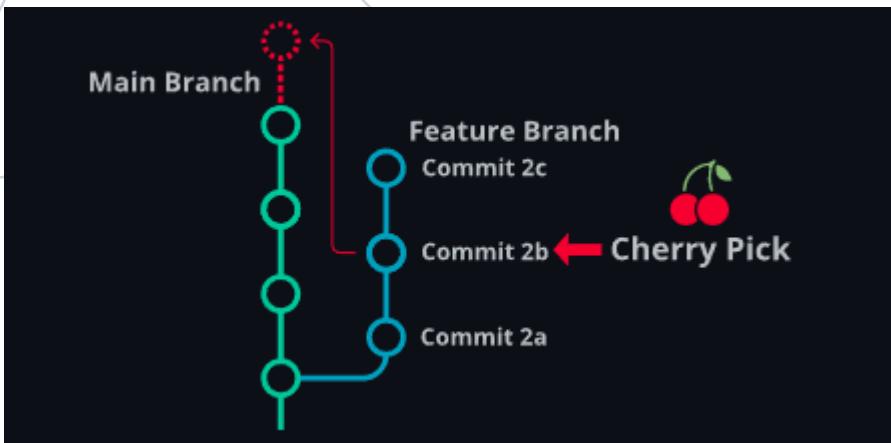
Imaginez maintenant dix personnes qui travaillent sur la même branche (c'est très mal) et qui pushent et pullent toutes les cinq minutes. Vous imaginez la gueule de l'historique ?

Pour éviter ce problème, on va utiliser l'option `git pull --rebase`, qui produira le résultat suivant.

```
origin/master  
/  
A---B---C---D---E---F ← master
```

Cherry-Pick

Cherry pick



- C'est le report de commit d'un endroit à un autre dans l'historique d'un même projet. Il s'agit donc de copier-coller un ou plusieurs commits.
- Pourquoi ?
 - Report de correctif
 - Report de fonctionnalité

Comment faire

- La procédure est simple et se fait en deux étapes :
 - Comme pour une fusion, on se place sur la branche qui va évoluer / recevoir le commit
 - On réplique le commit :

```
git cherry-pick -x <commit-1>
```

- On peut aussi en « déplacer » plusieurs :

```
git cherry-pick -x <commit-1> <commit-2>
```

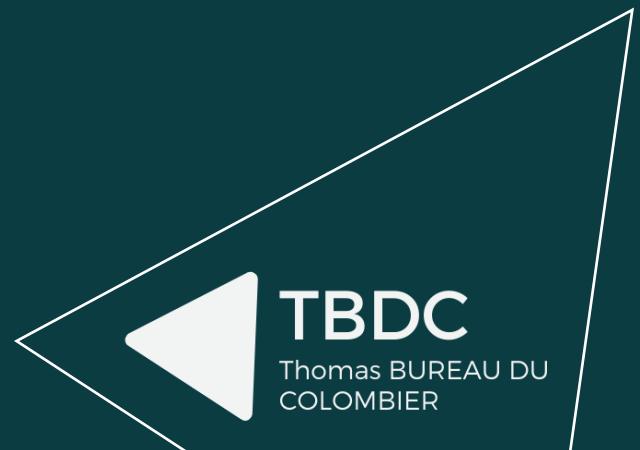
-x

Lors de l'enregistrement du commit, ajouter une ligne qui ajoute "(cherry pick from commit ...)" au message de commit original afin d'indiquer de quel commit ce changement a été picoré. Ceci est fait uniquement pour les picorages sans conflits. N'utilisez pas cette option si vous faites du picorage depuis votre branche privée car l'information est inutile pour le destinataire. En revanche, si vous faites du picorage entre deux branches visibles publiquement (par exemple, le rétroportage d'un correctif dans une branche de maintenance d'une ancienne version à partir d'une branche de développement), l'ajout de cette information peut être utile.



Git

COLLABORATION, BRANCHES ET DÉPÔTS DISTANTS



La base



Un dépôt est comme un dossier qui conserve un historique des versions et des modifications d'un projet. Il peut être local ou distant.



Dans la documentation en ligne ou en milieu professionnel, on parle souvent de repository, qui est la traduction anglaise du terme "dépôt".

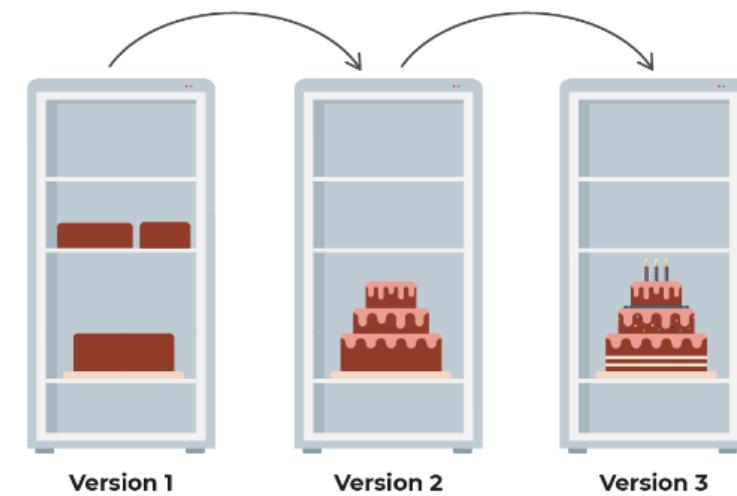
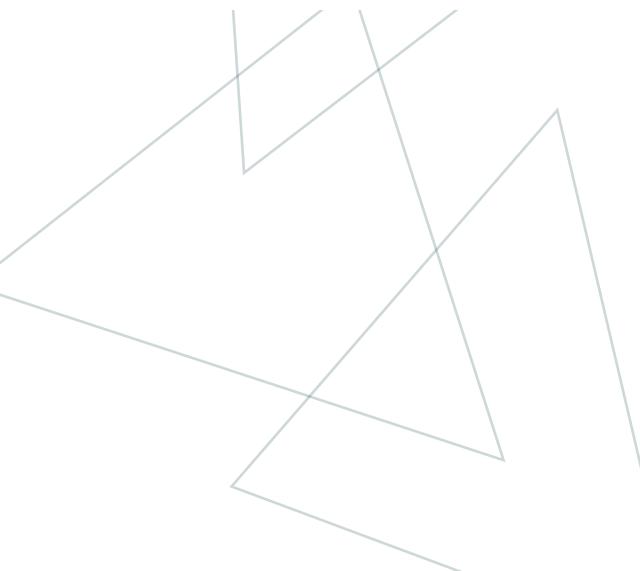
Le dépôt local

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

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

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

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



Utilisez un dépôt local comme un réfrigérateur et préparez votre gâteau !

Le dépôt distant

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

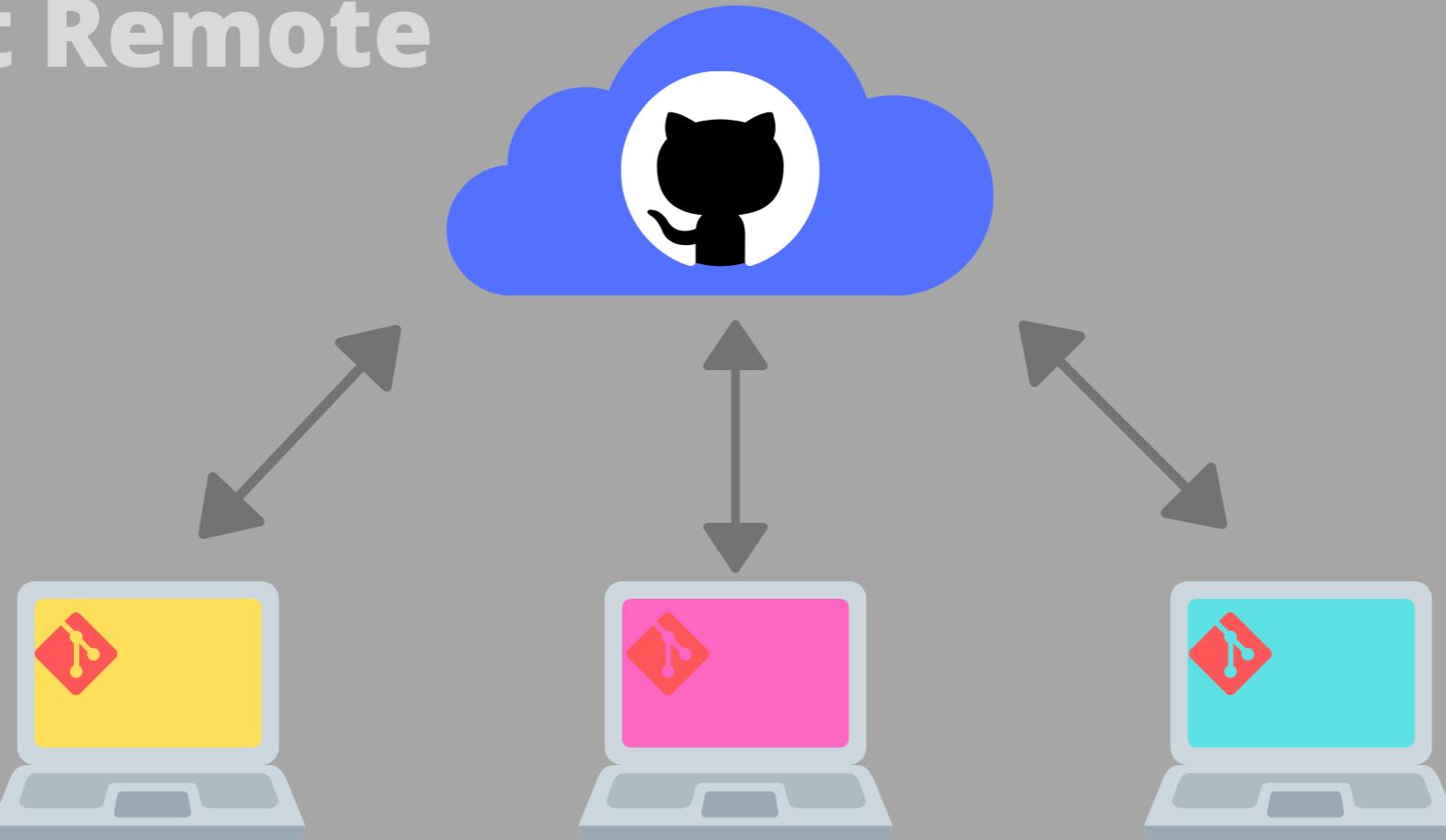
Ben oui, imaginez que votre PC rende l'âme demain, vous aurez toujours vos super programmes sur un dépôt distant ! C'est pourquoi, je vous conseille de toujours commencer par copier vos sources sur un dépôt distant lorsque vous commencez un nouveau projet, avec GitHub par exemple ! Vous pourrez aussi les rendre publics, et chacun pourra y ajouter ses évolutions.

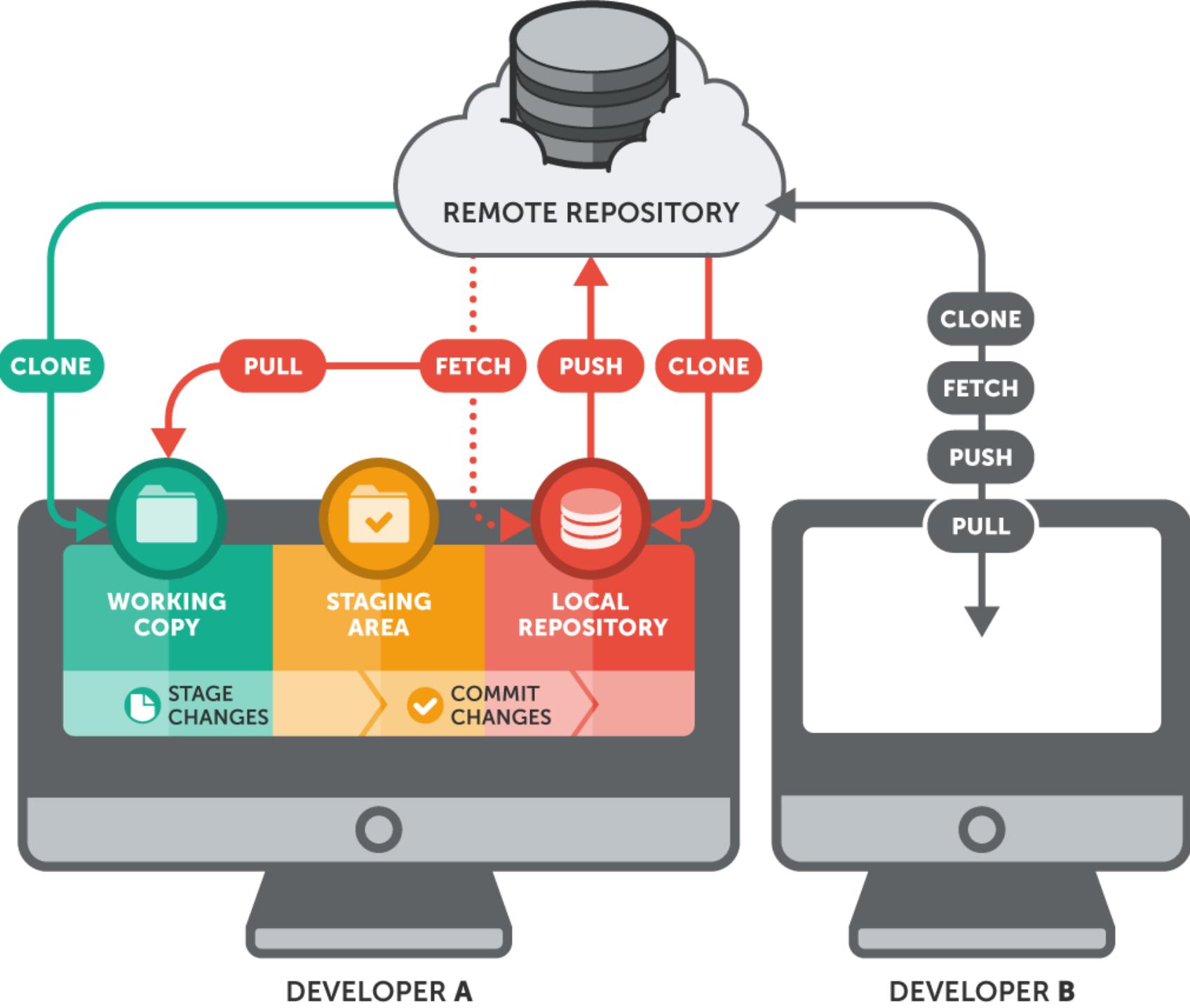


Sur un dépôt public, les personnes pourront collaborer à votre projet alors que sur un dépôt privé, vous seul aurez accès à votre travail et déciderez des personnes qui pourront ou non avoir accès à votre repository !

Organisation

Git Remote





Comment travailler avec un dépôt distant ?

- Créer un dépôt sur votre hébergeur (gitlab, github)
- Récupérer l'url du dépôt
- Ajouter la remote
- Configurer le dépôt distant :
 - Utilisez la commande git remote add pour ajouter un dépôt distant à votre projet local. Par exemple, si vous utilisez GitHub et que l'URL du dépôt distant est https://github.com/votre_utilisateur/votre_depot.git, utilisez cette commande :

```
git remote add origin https://github.com/votre_utilisateur/votre_depot.git
```
 - origin est le nom par défaut donné au dépôt distant, mais vous pouvez lui donner un autre nom si nécessaire.
- Vérifier la configuration du dépôt distant :
 - Vous pouvez utiliser la commande git remote -v pour vérifier que le dépôt distant a été correctement configuré.

Remote

- Les remotes Git sont des références vers des dépôts distants où votre code est stocké.
- Elles ont un nom et une URL
 - Le nom de la remote de base est conventionnellement « origin »
- Vous pouvez en avoir plusieurs par projet
 - Ex : origin(gitlab), prod(heroku)

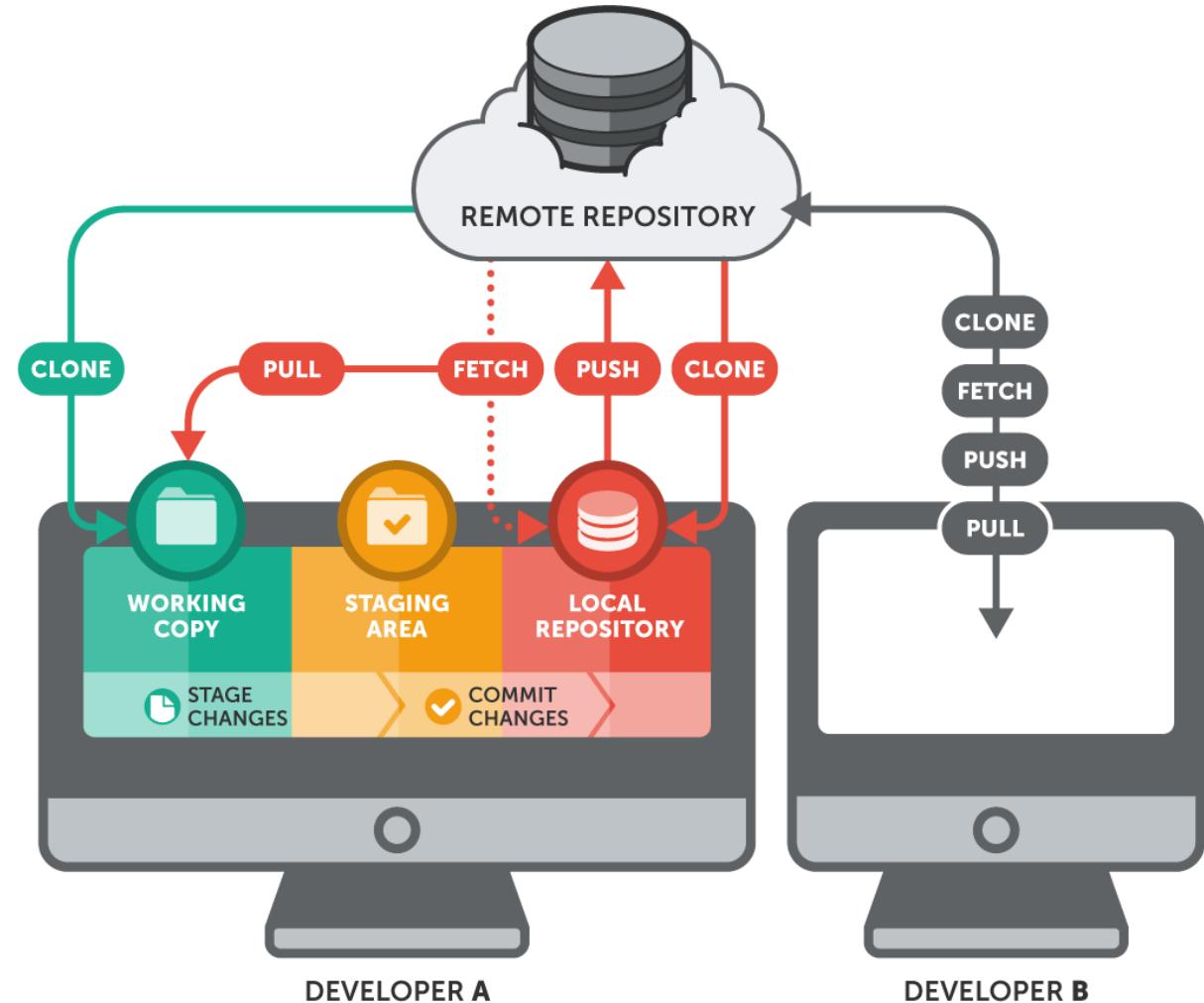
Gestion des remotes

- Ajouter une remote :
 - ‘git remote add <nom_remote> <URL_remote>’ : Cette commande ajoute une nouvelle remote avec un nom spécifié et une URL associée. Par exemple, git remote add origin <https://github.com/JohnDoeCourse/MyRepo.git> ajoute une remote appelée "origin" liée à l'URL du dépôt distant.
- Afficher la liste des remotes :
 - ‘git remote’ : Cette commande affiche la liste de toutes les remotes configurées dans votre dépôt local.
 - ‘git remote -v’ : Cette commande affiche la liste des remotes avec les URLs correspondantes.
- Renommer une remote :
 - ‘git remote rename <ancien_nom> <nouveau_nom>’ : Cette commande permet de renommer une remote existante. Par exemple, git remote rename origin new-origin renomme la remote "origin" en "new-origin".
- Supprimer une remote :
 - ‘git remote remove <nom_remote>’ : Cette commande supprime la remote spécifiée du dépôt local. Par exemple, git remote remove origin supprime la remote "origin".

Travailler avec le repo distant

Lorsque vous travaillez avec un dépôt distant, vous allez devoir **plusieurs fois par jour synchroniser votre travail** local avec le travail distant.

Pour cela, vous allez pouvoir utiliser trois commandes Git très connues Fetch, Pull et Push.



FETCH

La commande git fetch permet de récupérer **les dernières modifications du dépôt distant dans votre dépôt local, sans les fusionner automatiquement** avec votre branche courante.

Cela vous permet de voir les changements effectués par d'autres contributeurs sans modifier votre travail en cours.

Lorsque vous exécutez git fetch, Git met à jour les références de branches distantes et télécharge les commits associés, mais n'applique pas les modifications sur vos branches locales.

Vous pouvez ensuite utiliser d'autres commandes (comme git diff ou git merge) pour examiner les modifications récupérées avant de les fusionner.

FETCH

Même si on l'utilise habituellement sans paramètres, on peut lui spécifier des paramètres pour adapter la commande à nos besoins :

- 'git fetch origin' : Par défaut, git fetch récupère les modifications du dépôt distant spécifié par la remote nommée "origin". Cependant, vous pouvez remplacer "origin" par le nom de n'importe quelle remote que vous avez configurée dans votre dépôt local.
- 'git fetch --all' : Cette option permet de récupérer les modifications de toutes les remotes configurées dans votre dépôt local, et pas seulement de la remote spécifiée par défaut. Il va chercher les informations de toutes les branches

PULL

La commande git pull est en fait une **combinaison de deux commandes** : git fetch et git merge.

Lorsque vous exécutez 'git pull', Git effectue automatiquement un git fetch pour récupérer les modifications du dépôt distant, puis un git merge pour fusionner ces modifications dans votre branche courante.

Cela permet de mettre à jour votre branche locale avec les dernières modifications du dépôt distant et de fusionner les changements avec votre travail en cours.

Cependant, gardez à l'esprit que git pull fusionnera automatiquement les modifications et peut entraîner des conflits si des modifications simultanées ont été apportées aux mêmes fichiers.

PULL

Tout comme la commande fetch, on utilise généralement fetch sans paramètres, mais des paramètres sont disponibles :

- 'git pull origin <branche>' : Par défaut, git pull récupère les modifications de la branche spécifiée du dépôt distant nommé "origin". Vous pouvez remplacer "origin" par le nom de n'importe quelle remote que vous avez configurée dans votre dépôt local, et <branche> par le nom de la branche du dépôt distant que vous souhaitez récupérer.
- 'git pull --no-commit' : L'option --no-commit permet de fusionner les modifications récupérées sans créer automatiquement un commit de fusion. Cela vous donne la possibilité d'examiner les modifications et de personnaliser le commit de fusion avant de le valider.

PUSH

La commande git push est utilisée pour **envoyer vos commits locaux vers le dépôt distant** correspondant.

Lorsque vous exécutez git push, Git transfère vos modifications locales vers le dépôt distant spécifié.

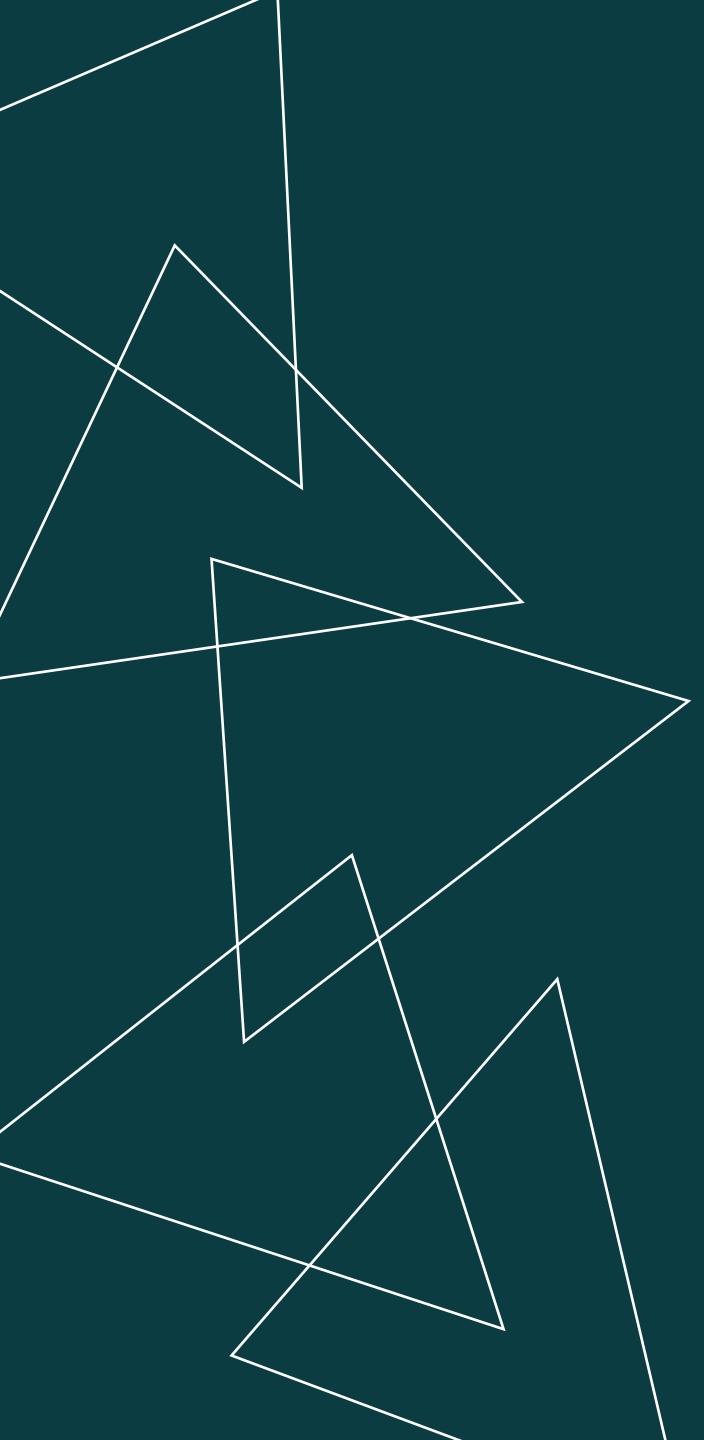
Vous devez disposer des autorisations nécessaires pour effectuer un push vers le dépôt distant.

Si d'autres contributeurs ont effectué des modifications sur le dépôt distant depuis votre dernière récupération (fetch ou pull), Git vous demandera de récupérer ces modifications avant de pouvoir pousser les vôtres.

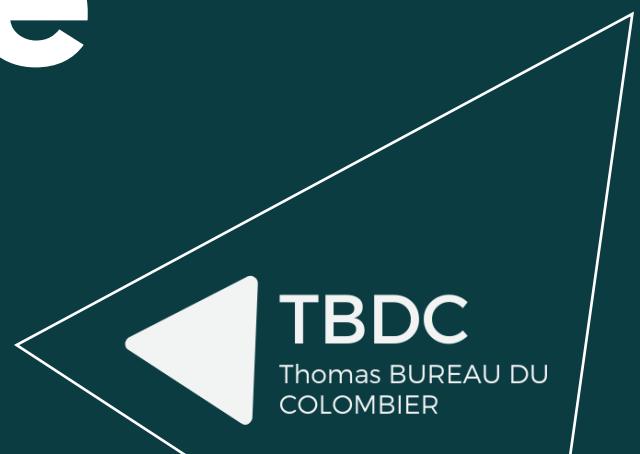
PUSH

Voici quelques paramètres possibles :

- 'git push origin <branche>' : Par défaut, git push envoie les commits de la branche courante vers la branche correspondante du dépôt distant nommé "origin". Vous pouvez remplacer "origin" par le nom de n'importe quelle remote que vous avez configurée dans votre dépôt local, et <branche> par le nom de la branche du dépôt distant vers laquelle vous souhaitez pousser les commits.
- 'git push --all' : Cette option permet de pousser toutes les branches locales vers le dépôt distant correspondant. Cela envoie les commits de toutes les branches locales vers les branches correspondantes sur le dépôt distant.
- 'git push --force' (ou -f): L'option --force permet d'écraser les commits du dépôt distant avec vos commits locaux, même si cela entraîne la perte de données. Utilisez cette option avec prudence, car elle peut modifier l'historique du dépôt distant et potentiellement causer des problèmes si d'autres contributeurs ont déjà récupéré ces commits.
- 'git push --set-upstream origin <branche>' (ou -u): Cette option configure la branche locale spécifiée comme une branche de suivi de la branche correspondante du dépôt distant. Cela signifie que vous pouvez utiliser simplement git push à l'avenir sans spécifier la remote et la branche, Git saura automatiquement où pousser les commits.



Branchons notre dépôt à la remote



Deux options

- Depuis un repo local vide, vers un dépôt distant vide
- Depuis un repo local vide, vers un dépôt distant existant :
 - git clone <url_depot>

Add your files

-
- [Create or upload files](#)
 - [Add files using the command line](#) or push an existing Git repository with the command line

```
cd existing_repo  
git remote add origin https://gitlab.com/tbdc.fr/coursgit.git  
git branch -M main  
git push -uf origin main
```

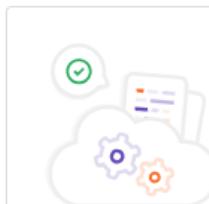
ⓘ GitLab has redesigned the left sidebar to address customer feedback. View details in [this blog post](#). Here's how to [file an issue](#) with the GitLab product team.

[Learn more](#)

⚠ You can't push or pull repositories using SSH until you add an SSH key to your profile.

[Add SSH key](#)[Don't show again](#)

⚠ Your account is authenticated with SSO or SAML. To [push and pull](#) over HTTPS with Git using this account, you must [set a password](#) or [set up a Personal Access Token](#) to use instead of a password. For more information, see [Clone with HTTPS](#).

[Remind later](#)[Don't show again](#)

Auto DevOps

It will automatically build, test, and deploy your application based on a predefined CI/CD configuration.

[Learn more in the Auto DevOps documentation](#)

[Enable in settings](#)[X](#)

ⓘ Get the most out of your trial with space for more members

During your trial, invite as many members as you like to CoursGit to collaborate with you. When your trial ends, you'll have a maximum of 5 members on the Free tier, or you can get more by upgrading to a paid tier.

[Invite more members](#)[Explore paid plans](#)[X](#)

ⓘ Project 'CoursGit' was successfully created.



CoursGit

Project ID: 52333719



Star 0



Forks 0

1 Commit 1 Branch 0 Tags 3 KiB Project Storage

1. Installer Git Credential Manager :

- Lors de l'installation de Git pour Windows, sélectionnez l'option pour installer le "Git Credential Manager".
- Sinon, vous pouvez également télécharger et installer le Git Credential Manager séparément depuis le site officiel Git : [Git Credential Manager for Windows](https://github.com/Microsoft/Git-Credential-Manager-for-Windows).

2. Configuration de GCM avec Git :

- Une fois installé, GCM devrait être automatiquement configuré pour fonctionner avec Git.
- Assurez-vous que Git est bien configuré pour utiliser le GCM en exécutant la commande suivante dans votre terminal Git :

```
lua
```

 Copy code

```
git config --global credential.helper manager
```

3. Utilisation :

- Lorsque vous clônez ou accédez à un dépôt distant pour la première fois, GCM vous demandera vos informations d'identification Git.
- Ces informations seront stockées de manière sécurisée dans le Windows Credential Store, et Git les utilisera automatiquement pour les connexions futures.

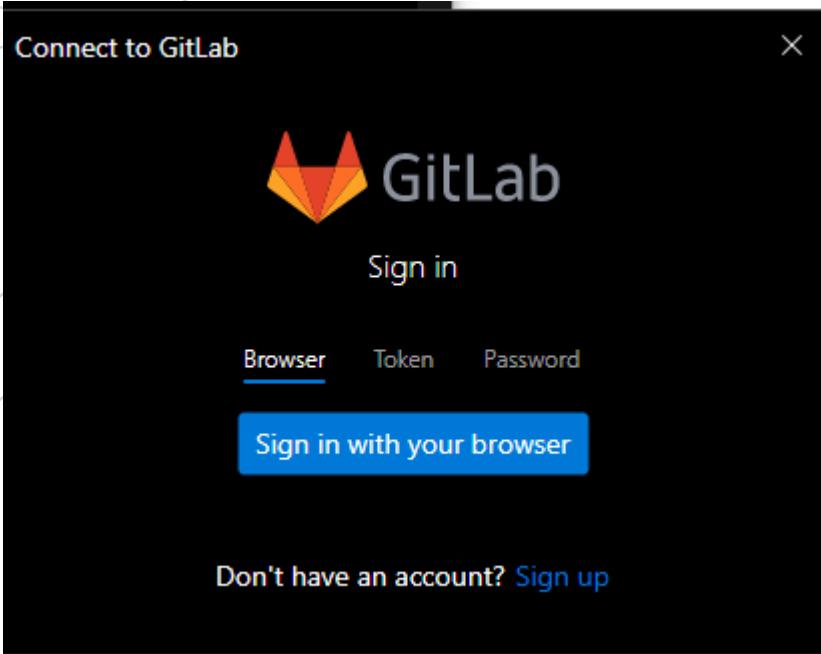
4. Mise à jour :

- Pour mettre à jour GCM, téléchargez la dernière version depuis le dépôt Git Credential Manager for Windows et suivez les instructions d'installation.

```
C:\Users\Thomas BDC>git credential-manager --version
2.3.2
```

Première connexion à Gitlab

```
$ git push -uf origin main
```



Authorize [Git Credential Manager](#) to use your account?

An application called [Git Credential Manager](#) is requesting access to your GitLab account. This application was created for group [Git Credential Manager](#). Please note that this application is not provided by GitLab and you should verify its authenticity before allowing access.

This application will be able to:

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

Deny

Authorize

Problème

- Si mon dépô local n'était pas vide, j'ai ce problème....

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble  
(main)  
$ git push -uf origin main  
Enumerating objects: 25, done.  
Counting objects: 100% (25/25), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (16/16), done.  
Writing objects: 100% (25/25), 2.15 KiB | 735.00 KiB/s, done.  
Total 25 (delta 4), reused 0 (delta 0), pack-reused 0  
remote: GitLab: You are not allowed to force push code to a protected branch on this project.  
To https://gitlab.com/tbdc.fr/coursgit.git  
! [remote rejected] main -> main (pre-receive hook declined)  
error: failed to push some refs to 'https://gitlab.com/tbdc.fr/coursgit.git'
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble  
(main)  
$ git push -u origin main  
To https://gitlab.com/tbdc.fr/coursgit.git  
! [rejected]          main -> main (fetch first)  
error: failed to push some refs to 'https://gitlab.com/tbdc.fr/coursgit.git'  
hint: Updates were rejected because the remote contains work that you do not  
hint: have locally. This is usually caused by another repository pushing to  
hint: the same ref. If you want to integrate the remote changes, use  
hint: 'git pull' before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- J'ai déjà un historique sur ma remote...

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGr  
(main)
$ git pull origin main
From https://gitlab.com/tbdc.fr/coursgit
 * branch           main        -> FETCH_HEAD
fatal: refusing to merge unrelated histories
```

main ▾ coursgit

Author ▾ Search by message



Nov 20, 2023

 Initial commit
TBDC authored 11 hours ago

38ff1f1b  

Alllow-unrelated-histories

L'idée derrière --allow-unrelated-histories est que Git vous permet de fusionner des branches sans lien entre elles. Cela fonctionne parfaitement lorsqu'il n'y a pas de conflits de fichiers. Si il y a des conflits, il vous faudra les gérer à la main.

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/GitLabRadiall/testgitradiall (main)
$ git pull origin main --allow-unrelated-histories
```

Protected branches

[Collapse](#)

Keep stable branches secure and force developers to use merge requests. [What are protected branches?](#)

 Giving merge rights to a protected branch also gives elevated permissions for certain CI/CD features. [What are the security implications?](#)

Protected branches  1					Add protected branch
Branch	Allowed to merge	Allowed to push and merge	Allowed to force push 	Code owner approval 	
main 	Maintainers 	Maintainers 	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Unprotect

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble  
(main)  
$ git push -uf origin main  
Enumerating objects: 25, done.  
Counting objects: 100% (25/25), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (16/16), done.  
Writing objects: 100% (25/25), 2.15 KiB | 735.00 KiB/s, done.  
Total 25 (delta 4), reused 0 (delta 0), pack-reused 0  
To https://gitlab.com/tbdc.fr/coursgit.git  
 + 38ff1f1...3f69bea main -> main (forced update)  
branch 'main' set up to track 'origin/main'.
```

Merge/Pull request

Merge request

Une "Merge Request" (demande de fusion) est une fonctionnalité couramment utilisée dans les systèmes de gestion de versions et de contrôle de code source, principalement dans les plateformes d'hébergement telles que GitLab, Bitbucket ou GitHub (où elle est appelée "Pull Request").

Principes de base :

- **Demande de Fusion :** Une Merge Request est une requête soumise par un collaborateur d'un projet pour fusionner les modifications qu'il a apportées dans une branche spécifique avec une autre branche, souvent la branche principale (par exemple, `main` ou `master`).
- **Travail en Branche :** Les développeurs travaillent généralement sur des branches séparées pour leurs fonctionnalités ou corrections de bogues, et une Merge Request permet de fusionner ces branches dans la branche principale après avoir terminé leurs modifications.

Éléments clés :

- **Description :** La personne soumettant la Merge Request fournit une description détaillée des changements effectués, incluant les raisons des modifications et tout détail pertinent pour les relecteurs.
- **Révision et Discussion :** Les autres membres de l'équipe peuvent examiner les modifications, la logique du code, poser des questions ou donner des suggestions directement sur la demande de fusion. Cela favorise la collaboration et la qualité du code.
- **Validation :** Les tests automatisés et les validations manuelles peuvent être effectués avant la fusion pour s'assurer que les changements ne provoquent pas de régressions ou de problèmes dans le code existant.

Processus :

1. **Création :** Le développeur crée une branche à partir de la branche principale, effectue des modifications, puis crée une demande de fusion pour intégrer ces modifications dans la branche principale.
2. **Examen :** Les autres membres de l'équipe passent en revue les modifications, posent des questions et fournissent des commentaires pour s'assurer de la qualité du code.
3. **Intégration :** Une fois les modifications approuvées, elles sont fusionnées dans la branche cible. Cela intègre les changements effectués dans la branche principale du projet.

Comment faire ?

1. **Créez une branche de fonctionnalité :** Avant de créer une merge request, créez une branche séparée pour la fonctionnalité ou la correction que vous souhaitez apporter. Utilisez des noms descriptifs pour vos branches afin de faciliter la compréhension de leur objectif.

bash

 Copy code

```
git checkout -b nom_de_la_branche
```

2. **Effectuez les modifications :** Travaillez sur votre branche de fonctionnalité, apportez les modifications nécessaires, et effectuez des commits pour sauvegarder votre travail.

bash

 Copy code

```
git add .  
git commit -m "Description des modifications effectuées"
```

3. **Poussez votre branche sur le serveur distant :** Une fois vos modifications effectuées et vos commits prêts, poussez votre branche sur le serveur distant.

bash

 Copy code

```
git push origin nom_de_la_branche
```

4. **Ouvrez une merge request :** Accédez à votre interface de gestion de versions (GitLab, GitHub, Bitbucket, etc.) et trouvez l'option pour créer une nouvelle merge request ou une pull request (selon la plateforme). Sélectionnez la branche que vous souhaitez fusionner (généralement votre branche de fonctionnalité) dans la branche cible (comme la branche principale).

Git

✓ You pushed to **dev** at [TBDC.FR / CoursGit](#) just now

[Create merge request](#)

C

CoursGit 

Project ID: 52333719 



22 Commits 2 Branches 0 Tags 109 KiB Project Storage 1 Environment



Update .gitlab-ci.yml file
TBDC authored 1 day ago

main

coursait / +

Historv

Find file

New merge request

From dev into main [Change branches](#)

Title (required)

Premier merge de dev vers main

Mark as draft

Drafts cannot be merged until marked ready.

Description

Preview

Ce merge request n'a pas d'importance, le but est juste de faire ma première merge request en direct.

[Switch to rich text editing](#)



Add [description templates](#) to help your contributors to communicate effectively!

Assignee

Unassigned

[Assign to me](#)



Reviewer

Unassigned

Approvals are optional.

[Approval rules](#)

Milestone

Select milestone



Merge options

- Delete source branch when merge request is accepted.
- Squash commits when merge request is accepted. [?](#)

[Create merge request](#)

[Cancel](#)

8 ✓

Approve

Approval is optional

Ready to merge!

Delete source branch

Squash commits

Edit commit message

The source branch is **8 commits behind** the target branch. · 1 commit and 1 merge commit will be added to main.

Merge



Le fork

Le fork

- Un **fork** en Git est une **copie d'un dépôt** sur ton propre compte, généralement utilisée pour contribuer à un projet open-source sans modifier directement le dépôt original.

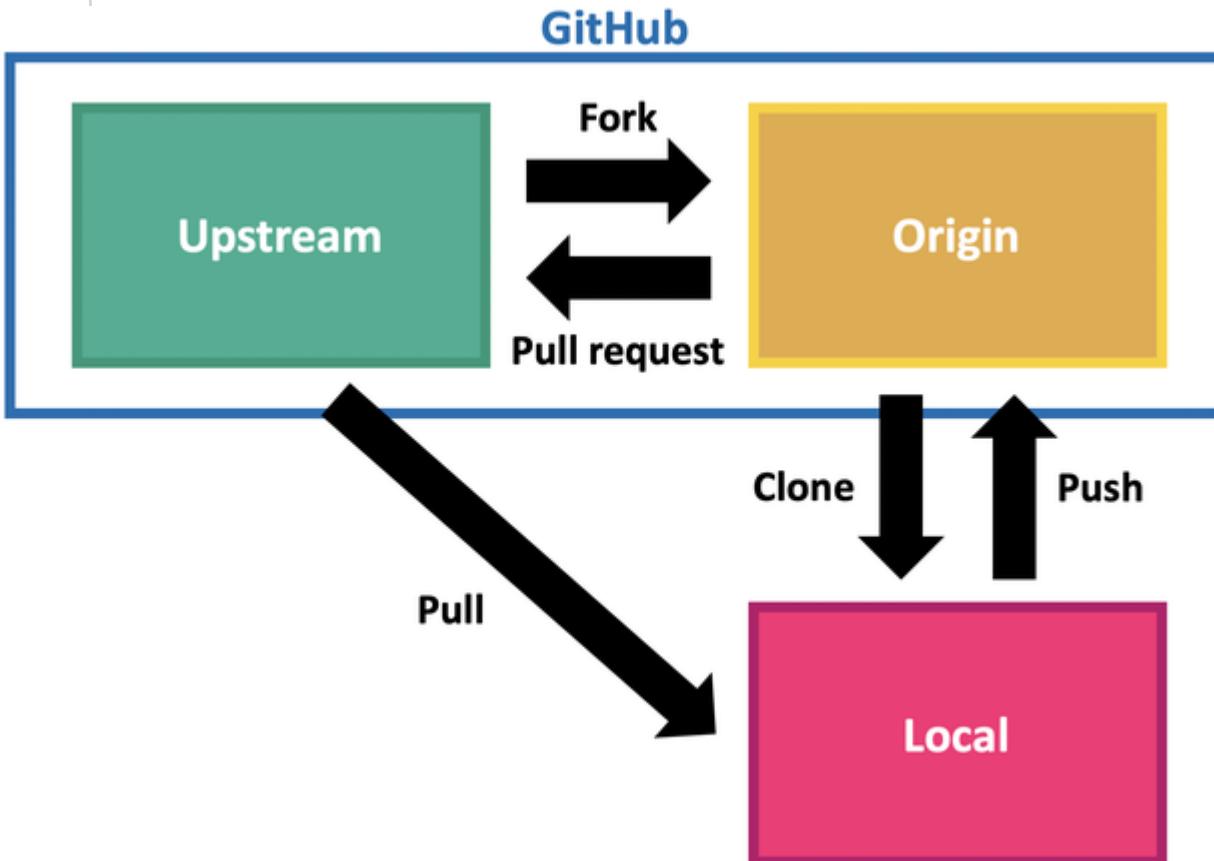


Pour contribuer à un projet open-source.



Pour travailler sur un projet sans modifier l'original.

Fonctionnement



- On aura deux remotes :
 - origin → ton fork
 - upstream → dépôt original

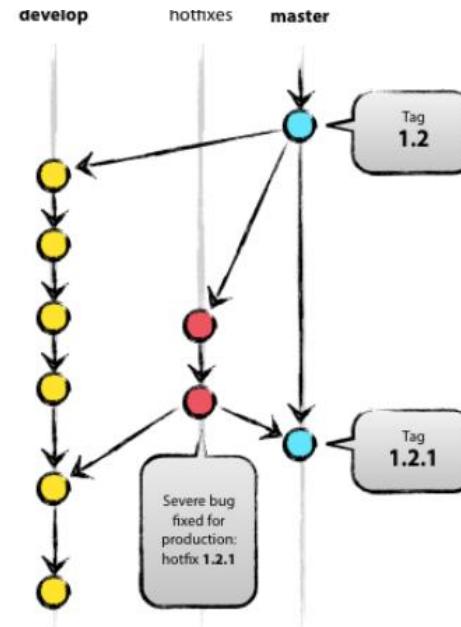
Git

**Bonnes pratiques : Bien gérer
ses commits**

Les tags

Les tags

Les tags sont des réfs qui pointent vers des points spécifiques de l'historique Git. Les tags sont généralement utilisés pour capturer un point de l'historique utilisé pour une version marquée (c.-à-d., v1.0.1). Un tag est similaire à une branche qui ne change pas. Contrairement aux branches, après leur création, les tags ne possèdent plus d'historique des commits.



Les commandes

Commande Git	Description
<code>'git tag'</code>	Affiche la liste des tags
<code>'git tag <tag_name>'</code>	Crée un tag portant le nom spécifié
<code>'git tag -a <tag_name> -m "message"'</code>	Crée un tag annoté avec un message spécifié
<code>'git tag -d <tag_name>'</code>	Supprime le tag spécifié
<code>'git push origin <tag_name>'</code>	Pousse le tag spécifié vers le dépôt distant
<code>'git push origin --tags'</code>	Pousse tous les tags vers le dépôt distant
<code>'git checkout <tag_name>'</code>	Change vers le commit associé au tag spécifié
<code>'git show <tag_name>'</code>	Affiche les informations sur le tag spécifié
<code>'git tag <tag_name> <commit_id>'</code>	Crée un tag sur le commit spécifié
<code>'git pull --tags'</code>	Récupère les modifications et les tags du dépôt distant

Attention

- Lorsque vous faites un checkout sur un tag, vous créez ce qu'on appelle un état "détaché". Cela signifie que vous n'êtes pas sur une branche mais directement sur le commit pointé par le tag. Dans cet état, si vous faites des commits, ceux-ci ne seront pas associés à une branche mais simplement au point spécifique où vous vous trouvez.
- Lorsque vous faites des commits dans cet état "détaché", cela crée des commits locaux mais sans référence de branche. Ces commits sont orphelins et ne sont pas associés à une branche spécifique. Si vous créez de nouveaux commits dans cet état détaché puis que vous revenez à une branche existante, ces nouveaux commits pourraient devenir inaccessibles si vous ne stockez pas leur identifiant (hash) quelque part.

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (dev)
$ git checkout v.1
Note: switching to 'v.1'.
```

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

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

```
git switch -c <new-branch-name>
```

Or undo this operation with:

```
git switch -
```

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

HEAD is now at 7562425 merge request

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble ((v.1))
$
```

A faire

- Créer un tag
- L'afficher via la commande gitlog
- L'afficher via la commande git show
 - Push le tag

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (dev)
$ git tag -a v.1 -m "création du premier tag"
```

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (dev)
$ git log
commit 7562425dc38414d248cdc917f3c6c8f446495e53 (HEAD -> dev, tag: v.1, origin/d
ev)
Author: Thomas BUREAU DU COLOMBIER <sio.bureauducolumbier@gmail.com>
Date:   Thu Nov 23 20:36:23 2023 +0100
```

merge request

```
Thomas BDC@Thomas-BDC MINGW64 ~/source/repos/coursGitPrepaGrenoble (dev)
$ git show v.1
tag v.1
Tagger: Thomas BUREAU DU COLOMBIER <sio.bureauducolumbier@gmail.com>
Date:   Thu Nov 23 20:45:11 2023 +0100

création du premier tag

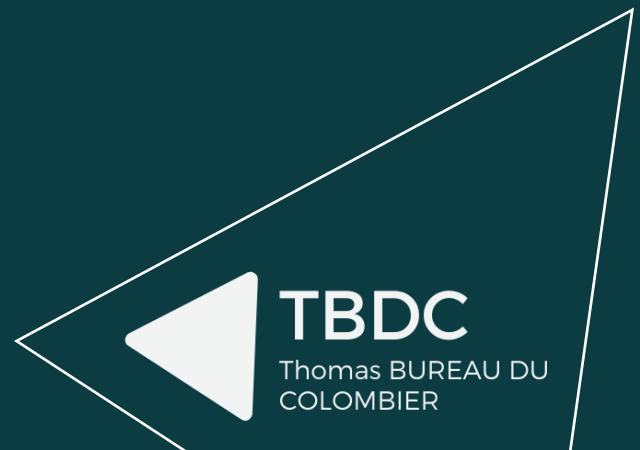
commit 7562425dc38414d248cdc917f3c6c8f446495e53 (HEAD -> dev, tag: v.1, origin/d
ev)
Author: Thomas BUREAU DU COLOMBIER <sio.bureauducolumbier@gmail.com>
Date:   Thu Nov 23 20:36:23 2023 +0100

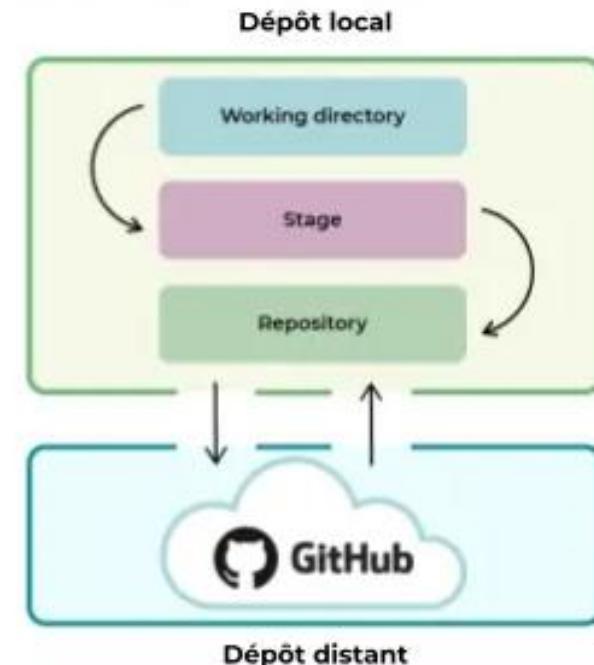
    merge request

diff --git a/mr.txt b/mr.txt
new file mode 100644
index 000000..6cb8106
--- /dev/null
```



Les bonnes pratiques



**Commandes de base****git add nom_fichier**

Pour ajouter vos fichiers dans l'index avant de les envoyer sur le dépôt distant, afin de créer une nouvelle version de vos fichiers.

git commit

Pour ajouter et lier un message rattaché au commit. Pensez à ajouter `-m` après "commit" pour préciser le message du commit dans la commande.

git push

Pour envoyer les modifications de code réalisées en local sur le dépôt distant.

git clone

Pour copier les changements d'une branche ou d'un repository sur votre dépôt local.

git branch

Pour lister les branches d'un projet.

git checkout nom_de_branche

Pour vous déplacer sur une branche.

git pull

Pour récupérer l'historique d'un dépôt et incorporer les modifications.

Bonnes pratiques

- Écrire des messages de commit précis et explicites.
- Créer une branche pour chaque nouvelle fonctionnalité de votre projet.
- S'entraîner :
 - Créer un dépôt distant et collaborer avec vos amis
 - Participer à des projets Open Source

Erreurs classiques

- Oublier d'initialiser le repository git local.
- Oublier de push votre code.
- Push des informations sensibles comme des informations personnelles ou des clés privées.
- Partager votre clé SSH privée au lieu de la clé publique.
- Oublier de pull les dernières modifications et créer des conflits.
- Abuser du flag `--hard` pour corriger des erreurs en ligne de commande.

Gestionnaire de version

Programme qui permet de conserver un historique des modifications et versions des fichiers.

Dépôt (ou repository en français)

Dossier virtuel qui conserve un historique des versions et des modifications d'un projet de code. Il peut être local ou distant.

Dépôt local

Dossier virtuel qui permet d'enregistrer en local les versions de votre code et d'y accéder au besoin.

Dépôt distant

Dossier virtuel qui permet de stocker les différentes versions de votre code afin de garder un historique délocalisé.

Définitions **Projet Open Source**

Projet accessible au public. Il peut donc être vu, lu et modifié par n'importe qui.

Branche

Copie de votre code à un instant T. Permet d'apporter des modifications au code sans impacter la branche principale.

Pull request

Fonctionnalité de GitHub qui permet de demander à fusionner son code avec une branche d'un projet.

Code review

Pratique consistant à relire et valider le code d'un collaborateur dans un projet avant de le fusionner avec la branche principale.

Quelques bonnes pratiques

1. Utilisez des messages de commit informatifs : Rédigez des messages de commit clairs et informatifs qui expliquent brièvement les modifications apportées.
2. Commitez fréquemment : Effectuez des commits fréquents, de préférence de petites unités logiques de travail, plutôt que de gros commits qui regroupent plusieurs fonctionnalités. La règle à respecter est : Chaque fois que mon code fonctionne, je peux faire un commit.
3. Fusionnez régulièrement : Si vous utilisez des branches, assurez-vous de fusionner fréquemment la branche principale (généralement master ou main) dans votre branche de fonctionnalité pour éviter les conflits massifs.
4. Ne jamais commit du code inachevé : Ne JAMAIS valider du code non testé ou non fonctionnel, car cela peut perturber le travail des autres.
5. .gitignore : Utilisez un fichier .gitignore pour spécifier les fichiers et répertoires à ignorer, tels que les fichiers de build, les fichiers de configuration locaux, etc. Vous pourrez ensuite modifier ces template pour l'adapter à vos besoins. Le template vous permettra de voir les bonnes pratiques habituelles pour votre type de projet. Voici la liste des templates de gitignore proposée par github : <https://github.com/github/gitignore>
6. Utiliser des ReadMe.Md pour documenter votre code.
7. Utilisez des tags pour les versions : Utilisez des tags Git pour marquer des versions stables de votre logiciel.
8. Synchronisez vos changements avec la remote (gitlab) régulièrement, c'est fondamental, surtout si vous travaillez à plusieurs.
9. Ayez un style d'équipe : Mettre en place un workflow, ainsi que différentes règles de nommage (nommage des commits, nommage des tags...).
10. Pour vérifier l'application de ces règles, et former l'effectif à ces règles, il peut être bien d'avoir un 'leader technique git'. Le rôle de ce développeur sera de se tenir à jour des bonnes pratiques, de maintenir le serveur Gitlab. Il pourra aussi former les nouvelles recrues à votre manière de travailler.

Créer de petits commits

La première règle est de ne pas garder un énorme tas de code non commit. Pourquoi ? La réponse est simple. Au bout d'un moment, tu ne sauras plus ce que tu as créé au début de ta fonctionnalité. De plus, si tu n'as rien poussé sur un hébergeur git distant, prie pour ne pas avoir de problème...

Pour faire simple, oublie tout ce qu'on a pu te dire, la taille ça compte.

Faire des petits commits t'aideras sur plusieurs points. Ceux-ci seront bien plus lisibles et compréhensibles. Ils contiendront peu de modifications. Ça te facilitera aussi la tâche si tu veux revenir sur un point ou changer quelques fichiers. Ça sera plus simple que dans un immense commit. Ça t'aideras notamment lors des merge ou pull request. En effet, tu auras moins de code à vérifier et donc moins de conflits potentiels à gérer. Tu augmenteras ton nombre de commits et en même temps, tu augmenteras ta satisfaction personnelle 😊

C'est un remède contre la perte de cheveux et chaque commit fait naître un bébé licorne.

Message de commits clairs

1. Soyez Descriptif :

- **Clarté et Concision :** Le titre du commit doit décrire de manière concise mais précise les changements effectués dans ce commit.
- **Expliquez le Pourquoi :** Décrivez la raison du changement. Pourquoi cette modification est-elle nécessaire ? Quel problème ou quelle fonctionnalité cela résout-il ou ajoute-t-il ?
- **Contenu du Commit :** Évitez d'inclure plusieurs modifications différentes dans un seul commit. Idéalement, chaque commit devrait se concentrer sur un seul changement logique.

2. Utilisez un Langage Compréhensible :

- **Langage clair :** Utilisez un langage simple et clair, évitez les acronymes ou les termes ambigus qui pourraient ne pas être compris par d'autres collaborateurs.
- **Verbe à l'Impératif :** Utilisez un verbe à l'impératif pour décrire l'action du commit. Par exemple : "Ajouter", "Modifier", "Supprimer", etc.

3. Longueur Appropriée :

- **Courts et Informatifs :** Essayez de garder le titre du commit court (idéalement moins de 50-72 caractères) tout en étant informatif.
- **Message de Commit Complet :** Si nécessaire, ajoutez un corps explicatif dans le message de commit pour donner plus de détails sur les changements effectués.

Exemples de Noms de Commit :

- "Ajouter la fonction de recherche par catégorie dans le catalogue"
- "Corriger le bug d'affichage sur la page de connexion"
- "Mettre à jour la documentation pour inclure les instructions d'installation"
- "Optimiser la requête SQL pour améliorer les performances"

Astuce

Une chose que tu peux également mettre en place, ce sont de petits tags au début de ton message. Par exemple, j'utilise ceux-ci :

- [func] mon commentaire pour l'ajout d'une fonctionnalité
- [edit] pour la modification d'une fonctionnalité
- [del] pour la suppression d'une fonctionnalité ou fichier
- [fix] pour la correction d'un bug
- [refa] pour du refactor de code
- [misc] quand aucun des tags précédent ne correspond à la tâche

Cela te permettras, en un coup d'œil, de voir ce que contient le commit. Si tu utilises un gestionnaire de tâche (genre taskworld) tu peux ajouter l'identifiant de la tâche pour qu'un autre développeur comprenne plus facilement ton travail.

Par exemple :

[2512][edit] modification du formulaire de création d'utilisateur pour y ajouter son adresse.

Organiser les branches

- On parlera de git flow plus tard ...

Bien organiser ses branches

La première des grandes règles de git à noter, est qu'il ne faut jamais faire des commits en cours de développement sur master. Master c'est la production donc on ne touche pas cette branche.

Sur git, tu n'es pas obligé de n'avoir qu'une seule branche. Tu peux en avoir plusieurs en même temps. Il y a différentes méthodes d'organisation, la plus connue étant Git Flow.

Ce que tu retrouveras souvent, et je te le conseille, c'est d'avoir en plus de master une branche dev. C'est ta branche avec tous les commits en cours de développement et qui recevra des merges d'autres branches.

En plus de dev, l'idéal et de créer une branche par fonctionnalité. Une fois que cette fonctionnalité est opérationnelle, tu peux la fusionner dans la branche dev.

C'est en gros du git flow avec les normes de nommage en moins. Une fois que dev atteint un certain niveau d'avancement, et que celle-ci est testée et stable, tu peux la merge sur master. Ensuite tu peux prévoir une mise en production.

Pour les bugs rencontrés en production, tu peux créer une branche à partir de master. Corriger le ou les bugs et ensuite merge le correctif sur master et dev pour qu'il soit corrigé sur les différents environnements.

Ignorer

Ne pas commiter les dépendances dans le contrôle de code source

- Le fait de pousser les dépendances dans votre origine distante augmentera la taille du référentiel. Supprimez toutes les dépendances de projets incluses dans vos référentiels et laissez votre gestionnaire de packages les télécharger dans chaque build. Si vous avez peur de la «disponibilité des dépendances», vous devriez envisager d'utiliser une solution de gestionnaire de référentiel binaire comme Jfrog ou Nexus Repository. Ou consultez Git-Sizer de GitHub.

Ne pas commiter les fichiers de configuration locaux dans le contrôle de code source

- Nous vous déconseillons fortement de commiter vos fichiers de configuration locaux dans le contrôle de version. Il s'agit généralement de fichiers de configuration privés que vous ne souhaitez pas envoyer à distance, car ils contiennent des secrets, des préférences personnelles, un historique ou des informations générales qui ne doivent rester que dans votre environnement local.

Respecter son workflow

- Qu'est-ce qu'un workflow ?
 - A quoi ça sert ?
 - Comment on fait ?



Git

Les principaux workflows

Les workflows

Utiliser Git est une excellente décision, mais après avoir choisi Git, l'idéal est de se mettre d'accord sur notre manière de travailler avec celui-ci. Il nous faut choisir notre workflow. Il en existe de toutes sortes, mais certains ont plus de succès que d'autres. A vous de choisir celui qui est le plus adapté à votre équipe, et contrôler sa bonne utilisation, pour profiter au maximum de la puissance de Git. Il est possible de changer de workflow pour un projet particulier. Par exemple, si un projet est très simple, avec un ou deux développeurs, le workflow « Centralized » peut être utilisé pour gagner du temps de développement. Vous pouvez aussi créer votre propre workflow, en adaptant un workflow existant à vos besoins, le tout étant de matérialiser cette décision quelque part pour que tout le monde puisse se référer à ce cadre de travail.

Famous workflows

Voici les workflows les plus utilisés :

- Centralized
- Feature branching
- Gitflow
- Github Flow
- Gitlab Flow

Mais comment choisir ?

Lors de l'évaluation d'un workflow pour votre équipe, il est primordial de prendre en compte la culture de votre équipe. Le workflow existe pour améliorer l'efficacité de votre équipe et donc il ne faut pas qu'il soit une charge qui limite la productivité. Voici quelques questions intéressantes à se poser :

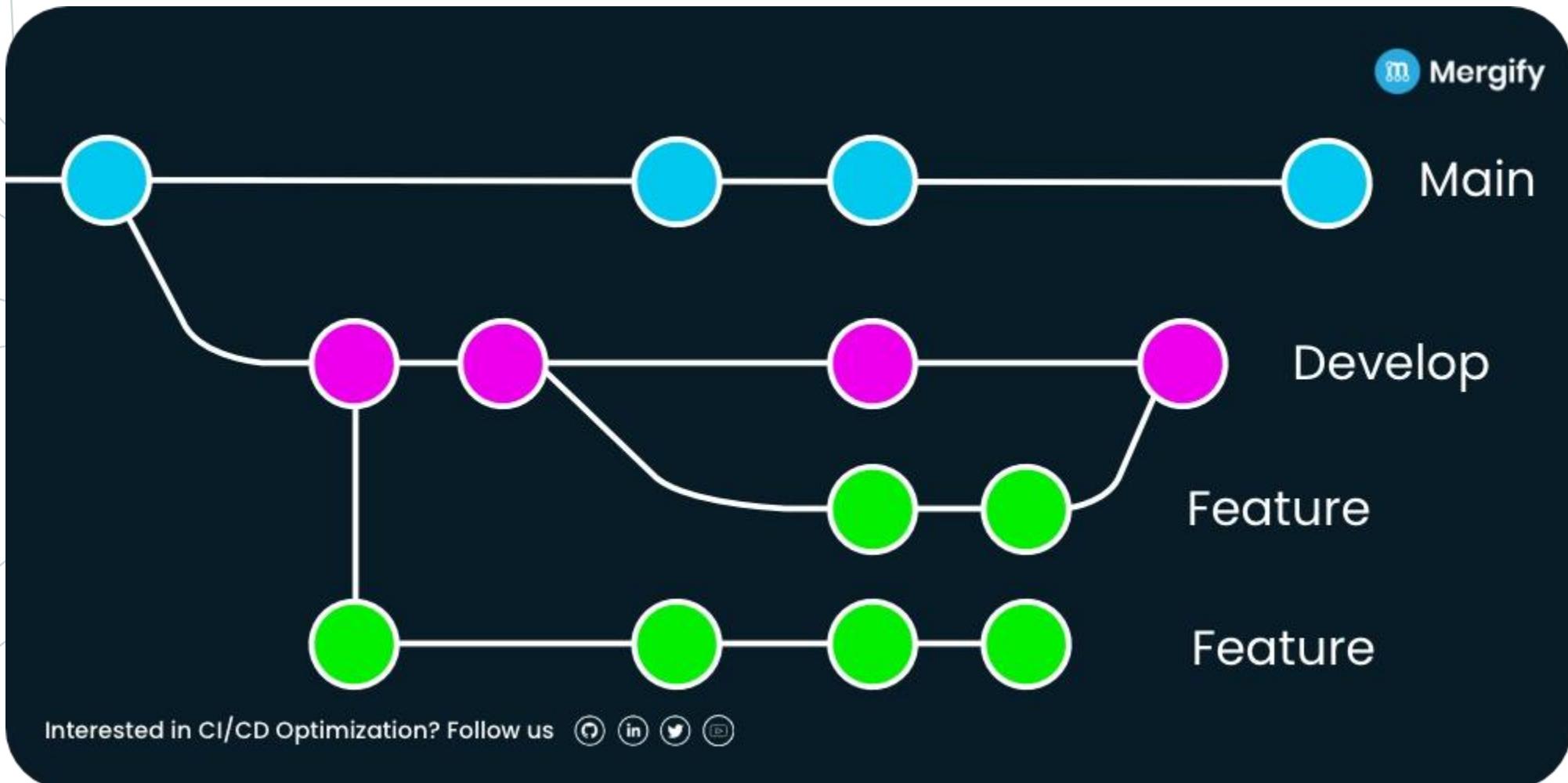
- Ce workflow est-il adaptable à la taille de l'équipe ?
- Est-il facile de corriger des erreurs et des fautes sans risque avec ce workflow ?
- Ce workflow impose-t-il une complexité inutile à l'équipe ?
- Est-il pratique au quotidien ?

Ce tableau pourra vous aider à choisir un workflow au début de chaque projet :

Workflow	Principes Clés	Utilisation Recommandée	Avantages Clés	Inconvénients Principaux
Centralized	Un référentiel central partagé Commits directs sur la branche principale	Petits projets ou équipes	Simplicité Historique linéaire	Difficulté à gérer les conflits Ralentit le développement sur des équipes nombreuses
Feature branching	Création de branches de fonctionnalités distinctes Fusion des branches de fonctionnalités dans la branche principale	Projets de taille moyenne Équipes de développement	Isolation des fonctionnalités Suivi des fonctionnalités	Possibilité de nombreuses branches Besoin d'une gestion efficace des branches
Gitflow	Utilisation de branches spécifiques (<u>feature</u> , <u>release</u> , <u>hotfix</u>) Workflow défini avec des règles strictes	Projets de taille moyenne à grande Logiciel complexe	Structure claire des branches Gestion des versions	Complexité Potentiel de fusion de branches
GitHub Flow	Basé sur des branches de fonctionnalités et des Pull Requests* (PR) pour la révision avant la fusion.	Projets collaboratifs avec PR et intégration continue.	Facilité de révision et d'intégration. Déploiement continu.	Manque de gestion des versions Difficulté pour les mises à jour en cascade
GitLab Flow	Combine l'utilisation de branches de fonctionnalités et des Merge Requests* (MR) avec des environnements de pré-production	Projets avec des besoins de tests avancés avant la fusion.	Test avancé avant la fusion. Environnements de pré-production.	Nécessite des efforts de configuration et de gestion Plus de complexité pour les projets plus petits

*Merge request et pull Request sont des termes similaires : le but est de demander à d'autres membres de l'équipe de réviser et de fusionner des modifications

Gitflow



<https://blog.mergify.com/trunk-based-development-vs-git-flow-when-to-use-which-development-style/>

Projet

Créer une documentation sur git

- Choisir un workflow à plusieurs
- Créer une documentation factice de l'utilisation de Git dans votre entreprise
- (rappel des commandes, tutos, workflows, bonnes pratiques...)
- **Equipes de 4**
- Idées : Faites des documents au format .md

Quelques sources pratiques

<https://grafikart.fr/formations/git>

<https://www.atlassian.com/git/tutorials>

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

Supprimer son compte de GCM

(Win + R → tape control keymgr.dll)

connectées et les réseaux.



Informations d'identification Web



Informations d'identification Windows

[Sauvegarder les informations d'identification](#) [Restaurer les informations d'identification](#)

Informations d'identification Windows

[Ajouter des informations d'identification Windows](#)



Git Cheat Sheet

Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new_branch

```
$ git branch new_branch
```

Delete the branch called my_branch

```
$ git branch -d my_branch
```

Merge branch_a into branch_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

```
$ git push
```

Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit <https://training.github.com/> for official GitHub training.

