



Systeme de controle de version

GitHub

Introduction

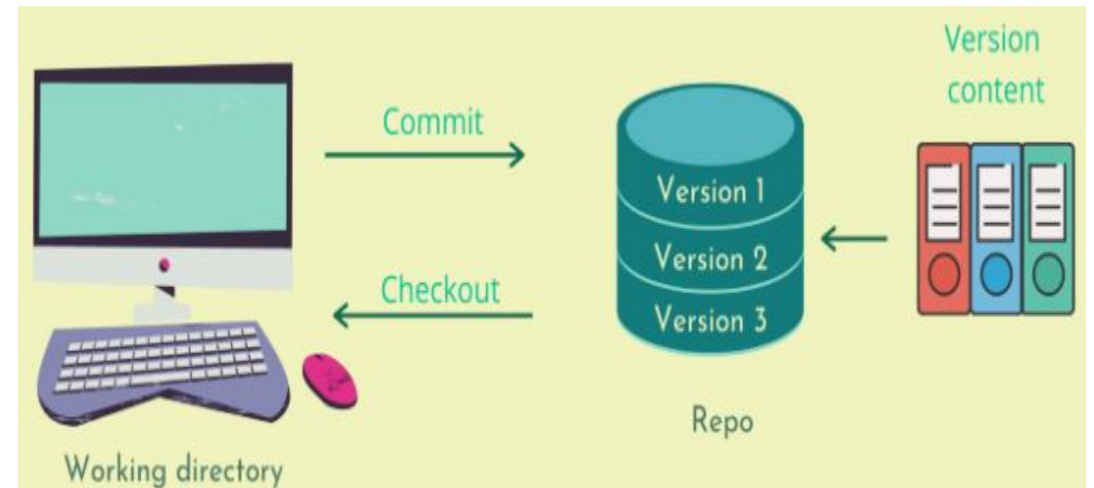
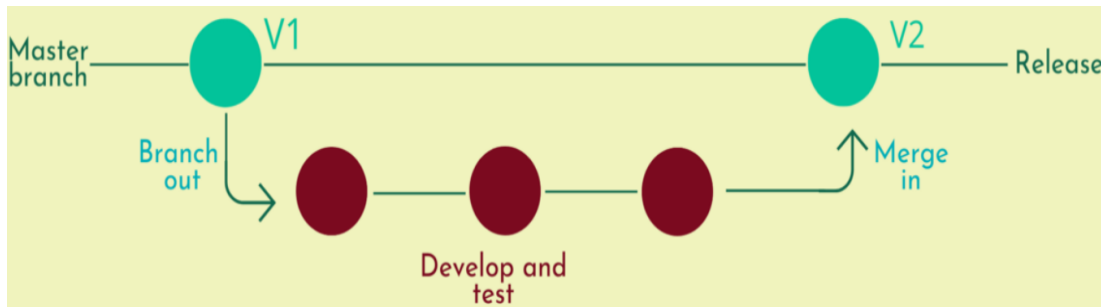
```
|-- core/
|   |-- tests/
|       |-- test_pull_data.py
|       |-- test_prepare_data.py
|       |-- test_model.py
|       |-- test_deploy.py
|       |-- test_utils.py
|   |-- pull_data.py
|   |-- prepare_data.py
|   |-- model.py
|   |-- deploy.py
|   |-- utils.py
|-- experiment_1/
|   |-- data/
|       |-- training.csv
|       |-- validation.csv
|       |-- test.csv
|   |-- output/
|       |-- results.json
|       |-- models/
|           |-- model1
|           |-- model2
|   |-- job_config.py
|   |-- build_data.py
|   |-- train.py
|   |-- evaluate.py
|   |-- prod.py
|-- experiment_2/
|   |-- data/
|       |-- training.csv
|       |-- validation.csv
|       |-- test.csv
|   |-- output/
|       |-- results.json
|       |-- models/
|           |-- model1
|           |-- model2
|   |-- job_config.py
|   |-- build_data.py
|   |-- train.py
|   |-- evaluate.py
|   |-- prod.py
```

Qu'est-ce que le contrôle de version

- Le contrôle de version vous permettent de garder une **trace** des changements que vous avez apportées à votre travail au **fil du temps**.
- Les systèmes de contrôle de version prennent également en charge l'idée de **branchement**, permettant à différentes personnes d'apporter différents ensembles de modifications aux mêmes fichiers sous-jacents, puis de fusionner leur travail plus tard.

Suite- Qu'est-ce que le système de contrôle de version?

- Avec un système de contrôle de version, vous pouvez:
- Voir tous les changements
- Récupérer les anciennes versions
- Créer des *branches*



types de systèmes de contrôle de version

Il existe deux types de systèmes de contrôle de version:

- **Système de contrôle de version distribué**
- **Système de contrôle de version centralisé**

Qu'est-ce qu'un système de contrôle de version distribué?

- Dans le **système de contrôle de version distribué**, chaque utilisateur clone le dépôt central pour créer son propre dépôt personnel qui comprend un historique complet du projet.
- Dans DVCS, chaque développeur ou client a son propre serveur et ils auront une copie de l'historique complet ou de la version du code et de toutes ses branches sur leur serveur ou machine local. Fondamentalement, chaque client ou utilisateur peut travailler **localement** et **déconnecté**, ce qui est plus pratique **que le contrôle de source centralisé** où il y a un seul serveur où tous les développeurs rapportent et apportent des modifications et c'est pourquoi il est appelé distribué.

Qu'est-ce que Git?

- **Git** est un système de contrôle de version distribué pour suivre les modifications du code source lors du développement logiciel. Il a été créé par **Linus Torvalds** (créateur et développeur de **Linux**) en 2005.
- Il simplifie le processus de **collaboration** sur les projets et suit les modifications dans n'importe quel fichier pendant la phase de développement.

Installer Git

- Si vous n'avez pas installé Git, allez [ici](#) et suivez la procédure d'installation de votre système d'exploitation.

Différence entre Git et GitHub

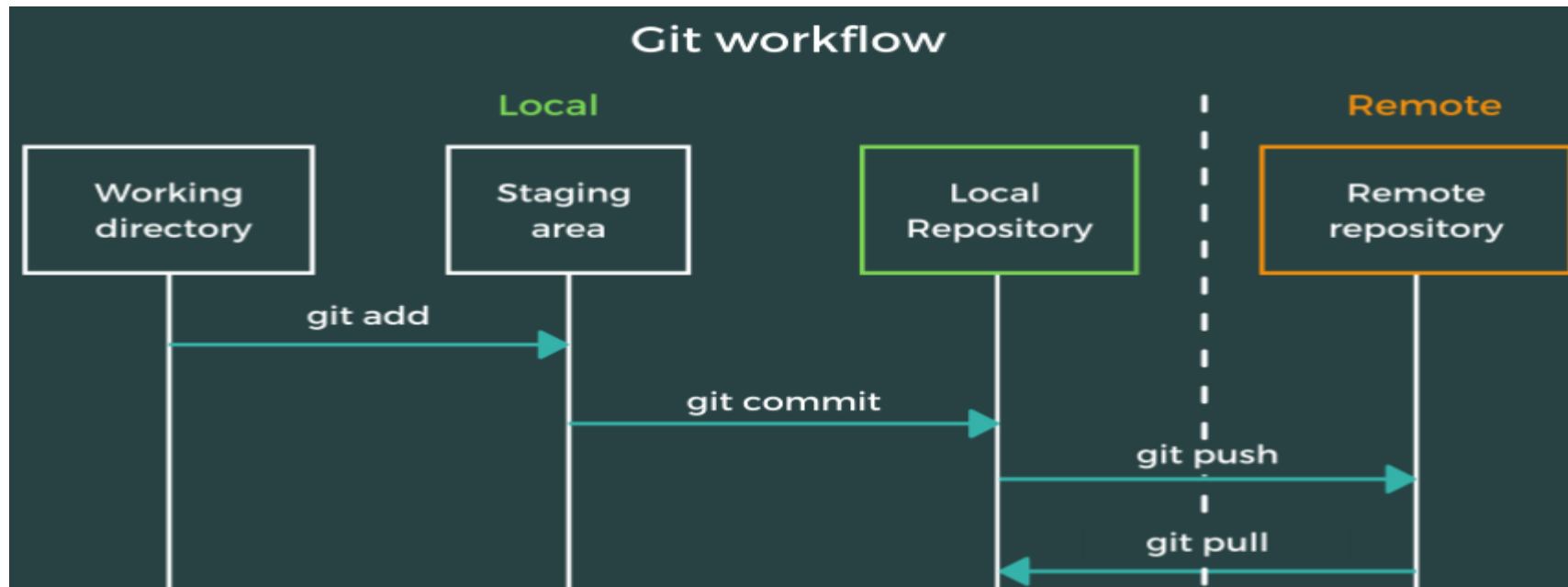
- **Git** est un système de contrôle de version tandis que **GitHub** est une plateforme de collaboration de code!
- **Git** est un logiciel tandis que GitHub est un service
- **Git** est installé localement sur l'ordinateur, GitHub est hébergé sur le Web
- **Git** fournit des outils de ligne de commande, GitHub fournit une interface graphique
- **GitHub** utilise **Git** pour le contrôle de version et vous fournit toutes sortes d'outils de collaboration

Git GitHub

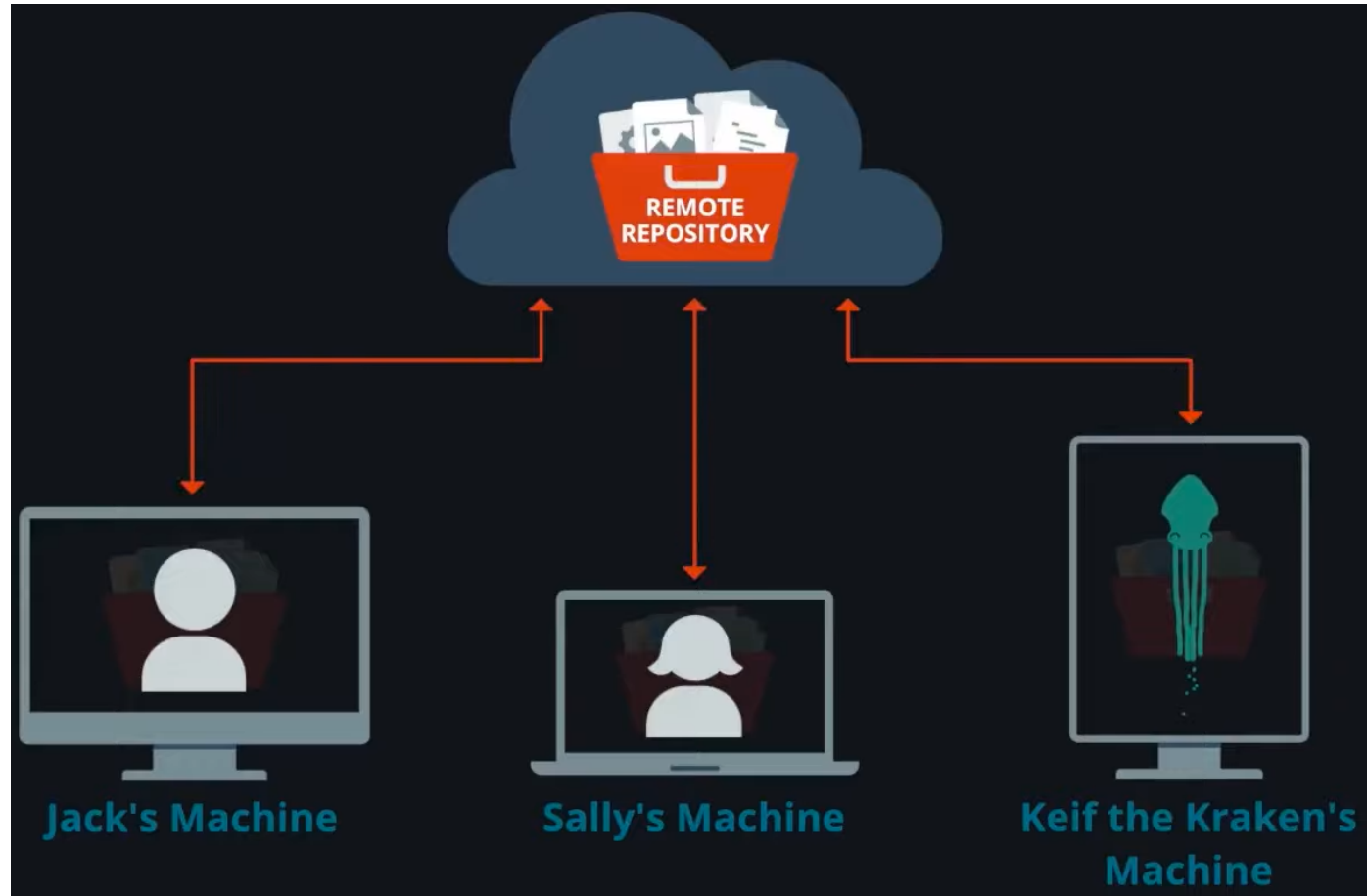


Git/Github

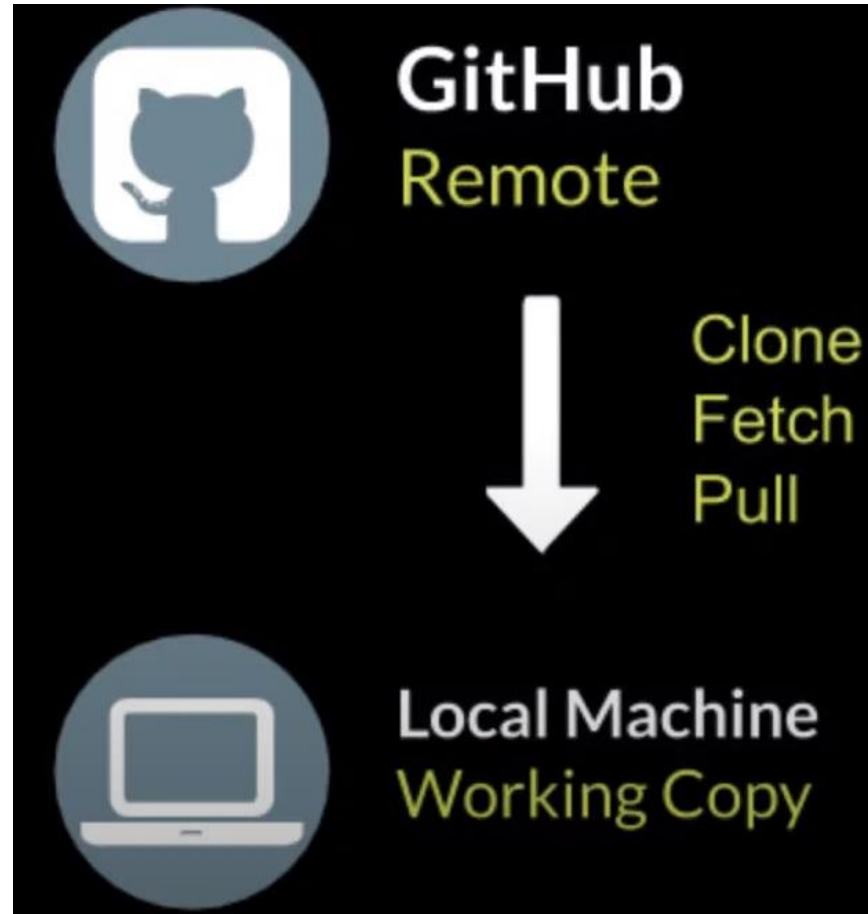
- Voici un flux de travail de base sur la façon dont certaines des commandes les plus fréquemment utilisées (add, commit, push et pull) sont utilisées dans un projet git.



Git GitHub



Git GitHub




BitBucket vs GitLab vs GitHub?

github

Create a new repository

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

Owner *

 cegep2020 ▾

Repository name *

GitExemple_1 ✓

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

Description (optional)

Pour tester git et github



Public

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



Private

You choose who can see and commit to this repository.

Initialize this repository with:

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



Add a README file

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




Add .gitignore

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



Choose a license

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

This will set  `main` as the default branch. Change the default name in your [settings](#).


Create repository

Fichier .gitignore

- Votre répertoire de projet peut contenir certains fichiers ou dossiers que vous ne souhaitez pas nécessairement suivre dans le dépôt.
.gitignore permet d'ignorer / ne pas suivre certains fichiers ou dossiers dans le répertoire.
- gitignore.io pour créer un fichier .gitignore utile
- [Comment utiliser un fichier .gitignore](#)

Git Large File Storage (LFS)

- <https://git-lfs.github.com/>




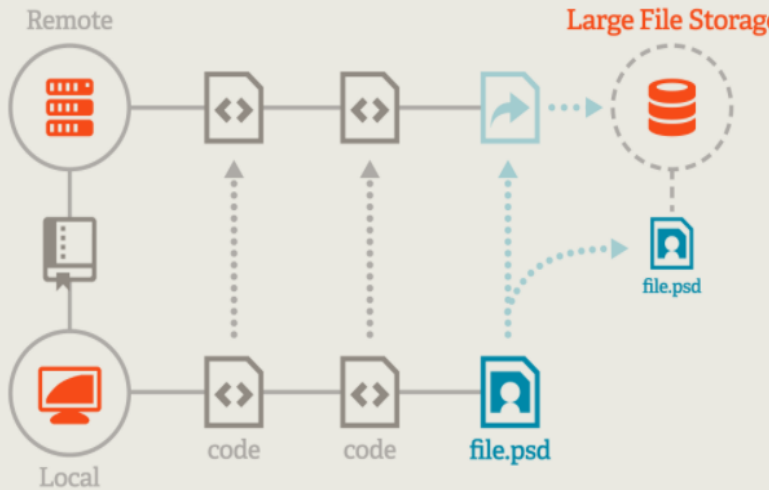
Git Large File Storage

[Docs](#) [Downloads](#) [Source](#)

An open source Git extension for versioning large files

Git Large File Storage (LFS) replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server like GitHub.com or GitHub Enterprise.

 **Download** v2.13.1 (Windows)



The diagram illustrates the Git LFS workflow. It shows a 'Local' environment at the bottom and a 'Remote' environment at the top. In the Local environment, there are 'code' files (represented by code icons) and a 'file.psd' (represented by a PSD icon). In the Remote environment, there are 'code' files and a 'Large File Storage' (represented by a database icon). Arrows indicate the flow of data: 'code' files are pushed from Local to Remote. The 'file.psd' is pushed from Local to the 'Large File Storage' and then to the Remote 'code' file. A dashed arrow also points from the Remote 'code' file back to the Local 'file.psd'.

Concepts de base-Git

Commençons par définir quelques concepts clés qui vous aideront lorsque nous parlerons de Git:

- **Repository:** C'est le nom de Git pour un projet. Il comprend tous les fichiers du projet ainsi que toutes les informations sur leur évolution au fil du temps. Si vous disposez d'une copie complète d'un dépôt(repository) (souvent appelé «repo»), vous pouvez afficher l'état actuel du projet, mais vous pouvez également afficher tout état antérieur dans lequel se trouvait le projet.

Concepts de base-Git

- **Commit:** Dans Git, l'historique est composé d'une série de commits qui sont stockés dans le changelog. Chaque fois que vous apportez un ensemble significatif de modifications à votre projet, vous devez les valider afin de pouvoir toujours ramener le projet dans cet état à l'avenir.
- **Staging Area:** C'est comme un panier pour le contrôle de version. C'est là que vous chargez les ensembles de modifications que vous souhaitez mettre dans votre prochain commit, donc si vous avez édité trois fichiers, mais que vous voulez faire un commit avec deux d'entre eux et un autre commit avec le troisième, vous n'avez qu'à "mettre en scène" Les deux premiers à l'aide de la commande, puis validez-les avec un message approprié, puis ajoutez et validez le dernier fichier séparément.

Qu'est-ce que la zone de staging area?

- Vous vous posez peut-être la question tout à fait raisonnable «pourquoi devons-nous exécuter deux commandes distinctes **git add**, puis **git commit** juste pour sauvegarder notre travail?»
- Premièrement, ce n'est pas quelque chose que vous devrez faire tout le temps. En tant que **data scientist**, vous passez la plupart de votre temps à modifier des fichiers - généralement votre bloc-notes Jupyter et peut-être quelques fichiers Python pris en charge. Lorsque vous modifiez des fichiers, Git donne un raccourci qui ajoutera à la fois les fichiers modifiés et les validera sur une seule ligne, donc la plupart du temps, vous n'avez qu'à taper une seule commande:

git commit -am "votre message ici"

suite

- le véritable pouvoir de la zone de préparation réside dans chaque fois que vous effectuez plusieurs modifications et que vous souhaitez ensuite revenir en arrière et les trier dans des commits distincts.

Les domaines de git

- Votre flux de travail git typique fonctionne de **gauche à droite**, en commençant par la **zone de travail**.
- Lorsque vous apportez des modifications aux fichiers dans un dépôt git, ces modifications s'affichent dans la **zone de travail**.
- Pour voir quels fichiers ont été modifiés avec les nouveaux fichiers que vous avez créés, effectuez simplement un **git status**.
- Pour afficher des détails plus détaillés sur ce qui a exactement été modifié dans un fichier, effectuez simplement un **git diff [file name]**.

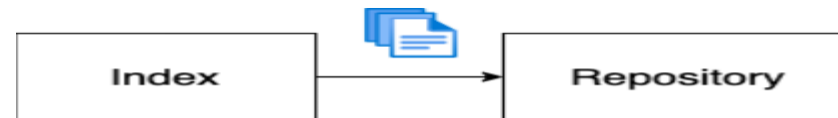


Suite- zones dans Git

- Une fois que vous êtes satisfait de vos modifications, vous ajoutez ensuite les fichiers modifiés de la zone de travail à l'index avec une commande **git add**



- Une fois que vous avez ajouté tous les fichiers pertinents à l'index, déplacez-les enfin vers le dépôt en effectuant un **git commit -m 'votre message'**.



- Vous devriez voir qu'il n'y a plus de différence entre l'index et le dépôt maintenant - ce que vous pouvez vérifier avec **git diff --cached**.

Les commandes Git

Configurations Git

- Nom d'utilisateur: **git config --global user.name <username>**
- Courriel: **git config --global user.email <email>**
- Accéder au fichier de configuration globale: **git config --global --edit**

Configuration initiale

- Vérifions simplement que vous avez la configuration de base pour Git afin que, lorsque vous enregistrez des fichiers, il connaisse votre nom et votre adresse e-mail.

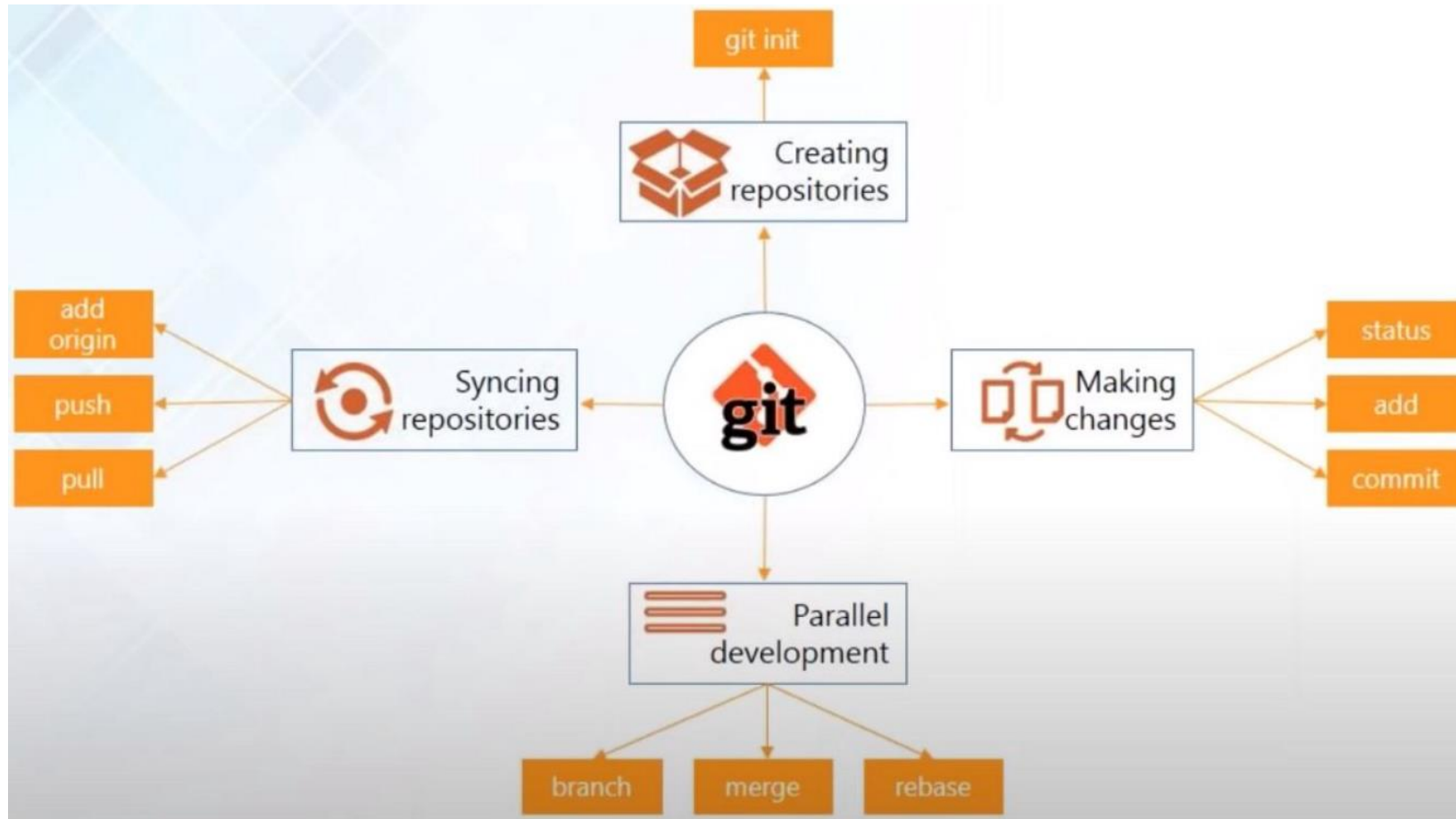
```
Utilisateur@ShahinServer MTINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git config --global user.name
cegep2020
```

Avec la commande ci-dessus, nous accédons aux paramètres de configuration de Git sur votre ordinateur.

-- **global** signifie que nous examinons les paramètres de configuration qui s'appliqueront à tous les projets sur lesquels vous travaillez connecté en tant qu'utilisateur sur cette machine.

--**system** indicateur inhabituel accède aux paramètres partagés entre tous les utilisateurs de votre ordinateur.

Les commandes de base utilisées dans un workflow



Commandes utiles

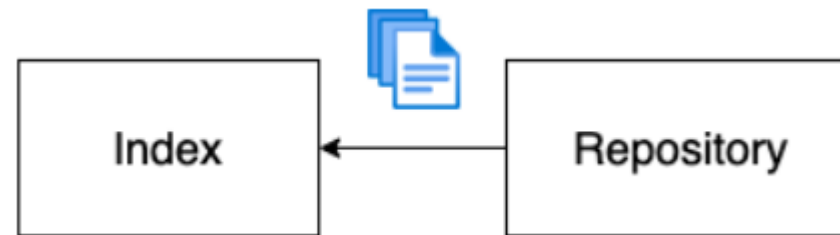
➤ `git reset --hard HEAD [file name]`.

➤ `git log`

➤ `git log -p`

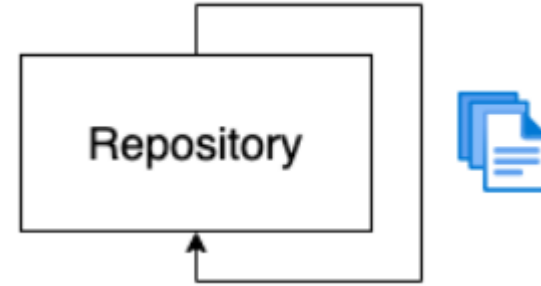


➤ `git reset --mixed HEAD [file name]`



Suite- commandes utiles

➤ `git reset --soft HEAD [file name]`



➤ Revenir à un commit précédent: **git revert [commit hash]**

➤ vous pouvez trouver votre hachage de validation en effectuant:

git log

Suite- Modifier votre historique de commit

➤ **Git rebase**: Par exemple, vous avez validé un fichier dans le dépôt local, mais vous avez fait un certain nombre de très petits commits que vous préférez simplement regrouper en un seul commit, voici la commande:

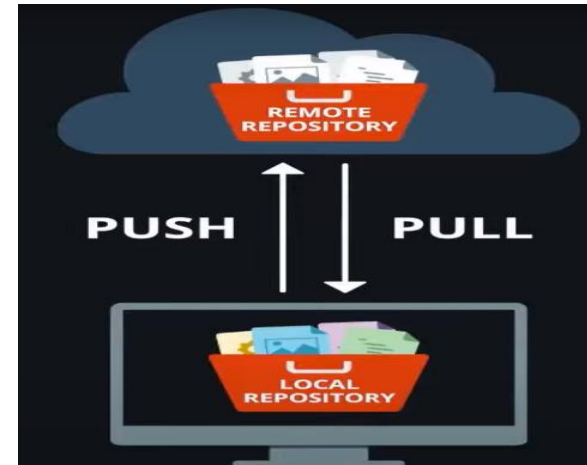
```
git rebase --interactive origin/main
```

Push/ Pull- clone

➤ **git push:** Cette commande est utilisée pour pousser les modifications validées vers le dépôt distant.

➤ **git pull:** Cette commande est utilisée pour extraire (obtenir) les nouvelles modifications du dépôt distant.

clone git: cette commande est utilisée pour créer une copie de travail locale d'un dépôt distant existant.



- **git checkout:** Cette commande est utilisée pour basculer entre différentes branches tout en travaillant. Il permet également d'ajouter une nouvelle branche et d'y basculer.

```
# Switching to an existing branch
$ git checkout <branch_name>

# Switching to a new branch
$ git checkout -b <new_branch_name>
```

git merge: Cette commande combine les modifications de différentes branches ensemble.

```
# Fusionner les modifications d'une branche dans la branche courante
$ git merge <branch_name>
```


Afficher les informations

- Déterminez le statut d'un Repo: *git status*
- Afficher les validations d'un Repo: *git log*
- Affichez les validations d'un Repo de manière compacte: *git log --oneline*
- Affichage des fichiers modifiés: *git log --stat*
- Affichage des modifications de fichier: *git log -p*
- Affichage des modifications de fichier en ignorant les modifications d'espaces: *git log -p -w*
- Affichage du dernier commit: *git show*
- Affichage d'un commit spécifique: *git show <SHA of commit>*

Ajouter- add

- «Staging» signifie déplacer un fichier du répertoire de travail vers l'index de transfert.
- Fichiers intermédiaires: `git add <file1> <file2> ... <fileN>`
- Annulation des fichiers: `git rm --cached <file>...`
- Mettre en scène tous les fichiers: `git add .`

Commit

- Validez les fichiers sans ouvrir l'éditeur de code: `git commit -m "Commit message"`

Suite- Commit

➤ Corriger des choses

Modifiez le dernier message de validation: `git commit –amend`

➤ Annuler un commit

- Cela créera un nouveau commit qui rétablira ou annulera un commit précédent.
- Annulez les modifications apportées dans un commit:

`git revert <SHA of commit>`

Réinitialiser un commit

❑ Cela effacera les commits.

`git reset <reference to commit>`

Les branches

- Lorsqu'un commit est effectué dans un dépôt, il est ajouté à la branche sur laquelle vous vous trouvez actuellement. Par défaut, un dépôt a une branche appelée *master*. En particulier lorsque vous expérimentez de nouvelles fonctionnalités sur votre code, il est souvent utile de créer une branche distincte qui agit comme un environnement isolé sûr de votre dernier commit. Vous pouvez basculer entre les branches et les validations ne seront ajoutées qu'à celle sur laquelle vous êtes actuellement.

Suite- Les branches

- Lister toutes les branches: `git branch`
- Créer une nouvelle branche: `git branch <branch-name>`
- Supprimer la branche: `git branch -d <branch-name>`

Notes:

- *Vous ne pouvez pas supprimer une branche sur laquelle vous vous trouvez actuellement.*
- *Vous ne pouvez pas supprimer une branche si elle contient des validations qui ne figurent sur aucune autre branche.*

Suite- Les branches

- Pour forcer la suppression: `git branch -D <branch-name>`
- Basculer entre les branches: `git checkout <branch-name>`
- Ajouter une branche sur un commit spécifique: `git branch <branch-name> <SHA of commit>`
- Démarrez la branche au même emplacement que la branche principale: `git branch <branch-name> master`
- Voir tous les banches à la fois: `git log --oneline --decorate --graph --all`

Renommer une branche distante

1. `git checkout [your branch]`
2. `git branch -m [new branch name]`
3. `git push origin :[your branch] [new branch name]`
4. `git push origin -u [new branch name]`

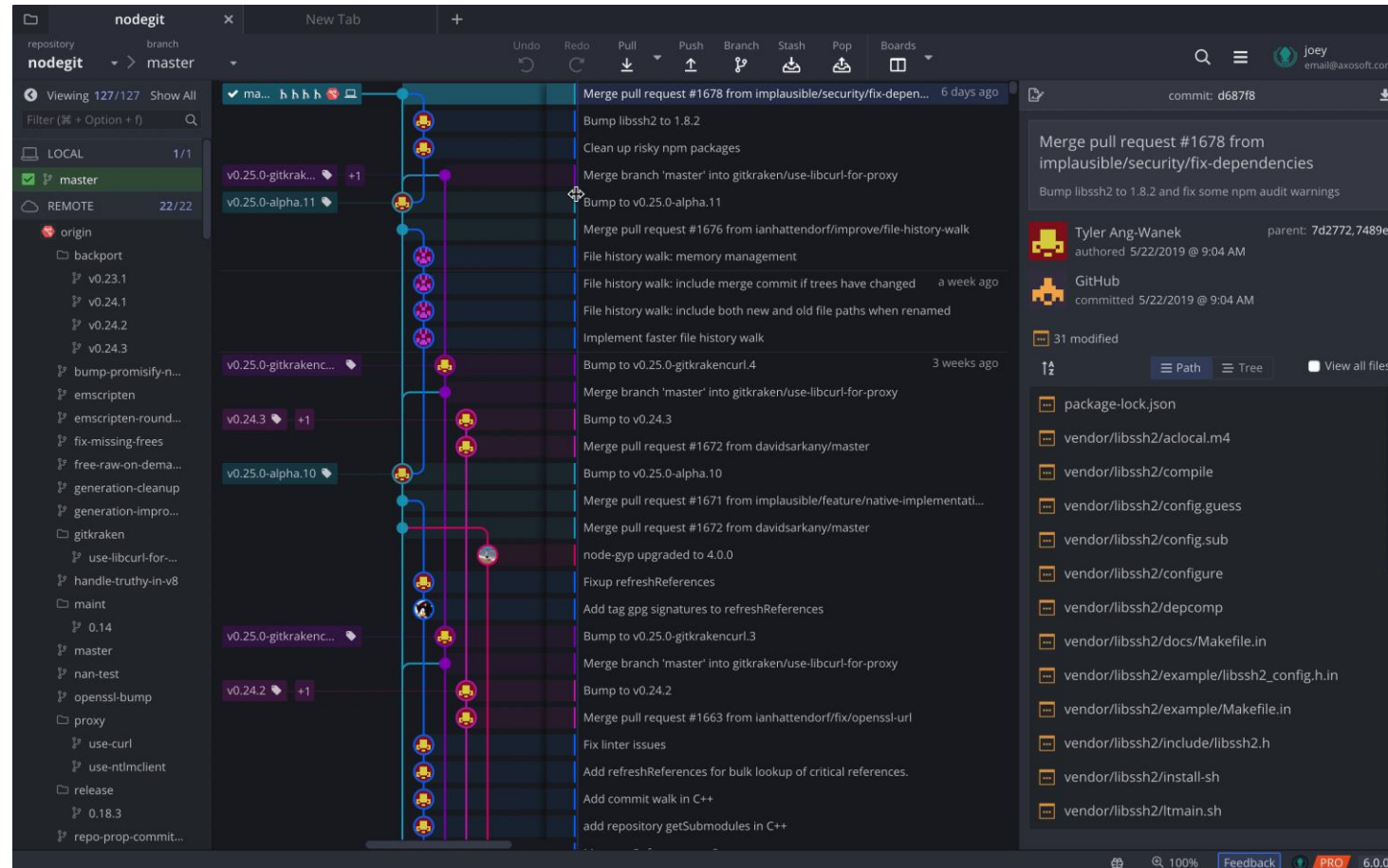
Fusion- Merging

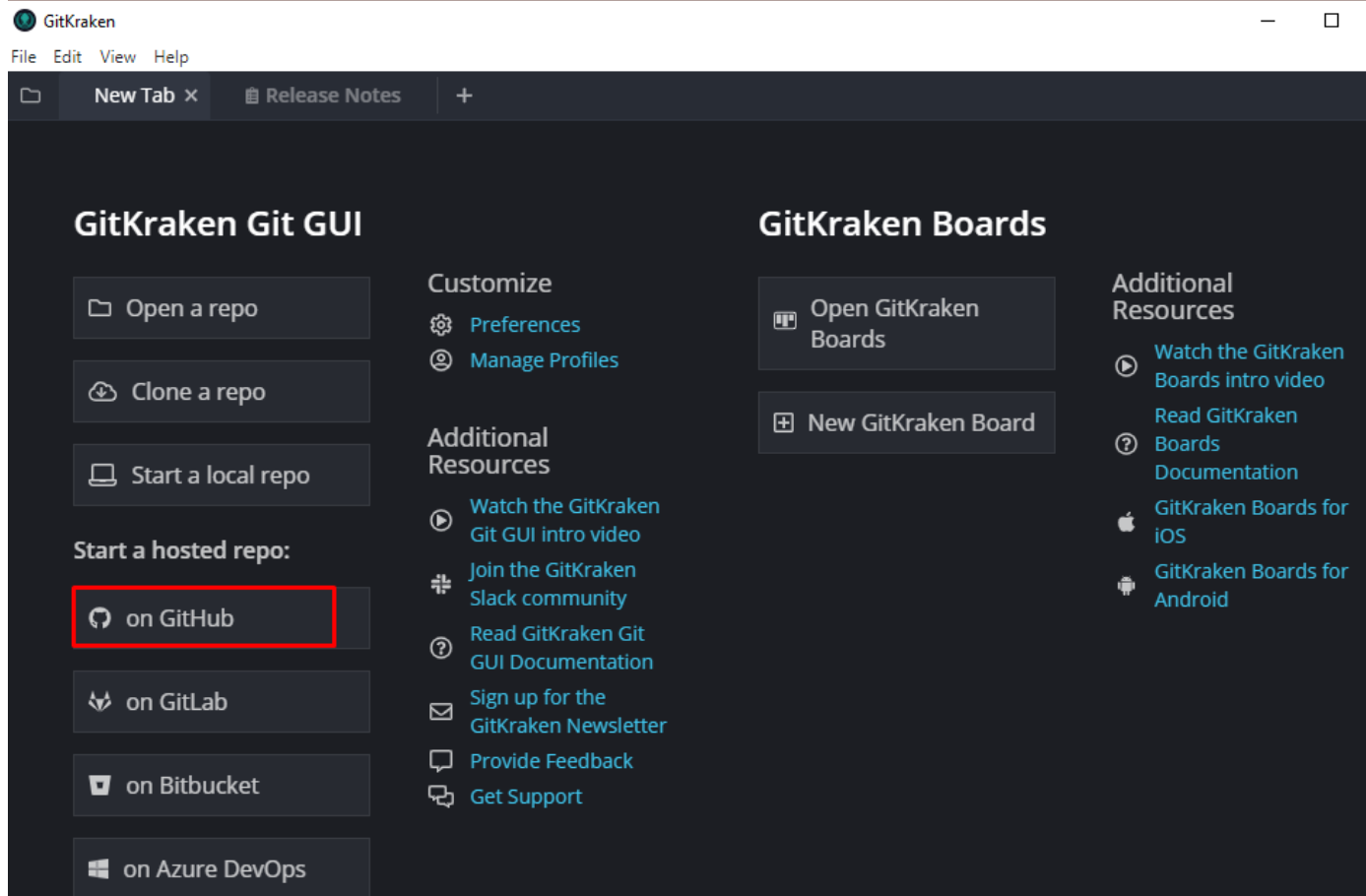
- Lorsqu'une fusion est effectuée, les modifications de l'autre branche sont introduites dans la branche actuellement extraite.
- Effectuer la fusion: `git merge <branch to merge in>`

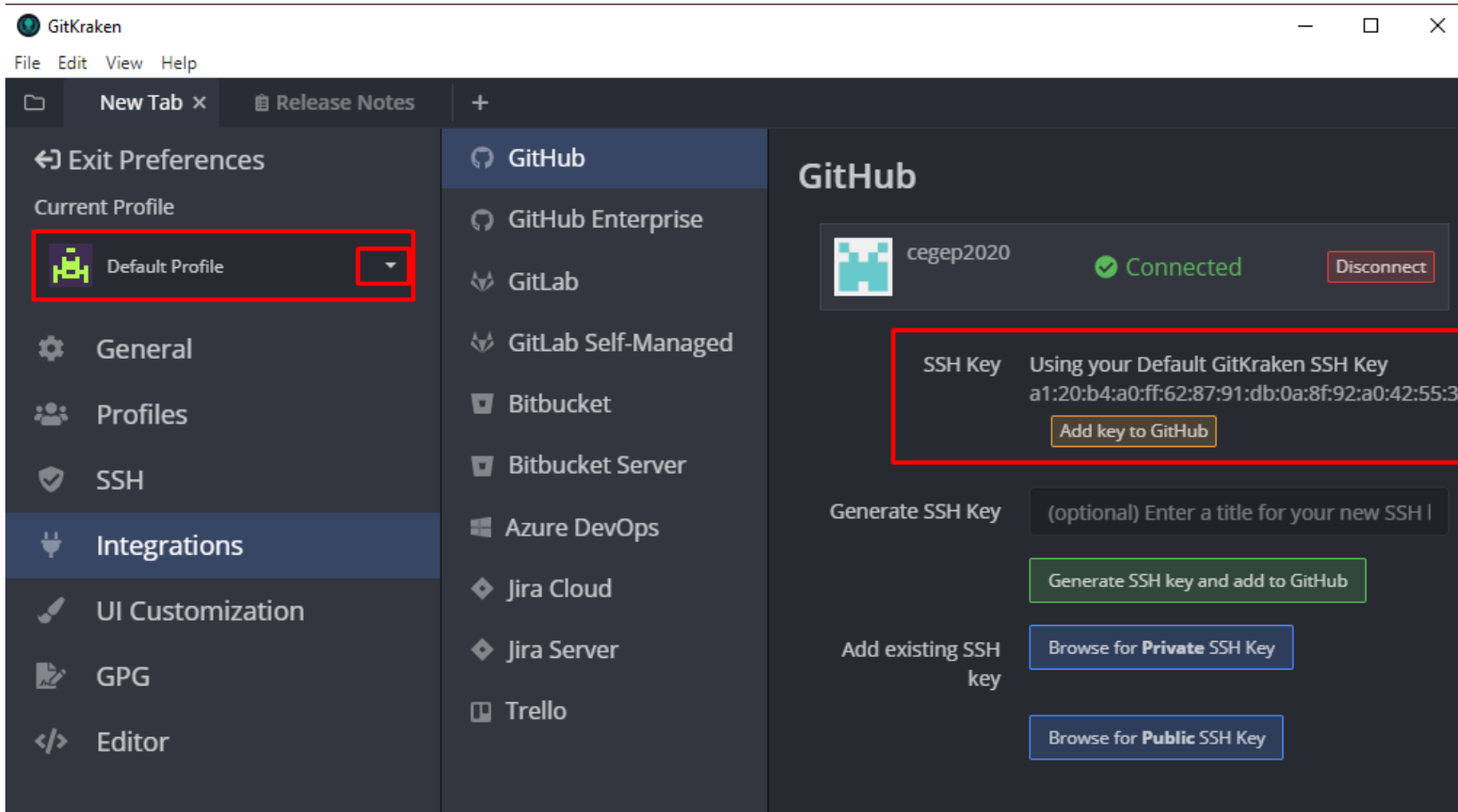
interface graphique ou d'une interface de ligne de commande

- <https://git-scm.com/downloads/guis>
- Aide Git
- <https://docs.github.com/en>

interface graphique Git GitKraken:







Examples

```
Utilisateur@ShahinServer MINGW64 ~
```

```
$ git status
```

```
fatal: not a git repository (or any of the parent directories): .git
```


Créons maintenant un nouveau dépôt(repo) et un nouveau répertoire Git en un:

```
Utilisateur@ShahinServer MINGW64 ~/Desktop
$ mkdir test_repo

Utilisateur@ShahinServer MINGW64 ~/Desktop
$ cd test_repo/

Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo
$ git init my_test_repo
Initialized empty Git repository in c:/Users/Utilisateur/Desktop/test_repo/my_test_repo/.git/

Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo
$
```

- Il a donc créé un dépôt sous le répertoire dans lequel j'étais. Utilisons la commande Unix «change directory (cd)» pour y aller:

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo  
$ cd my_test_repo/  
  
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)  
$ pwd  
/c/Users/Utilisateur/Desktop/test_repo/my_test_repo
```

- Mon terminal me dit quand je suis dans un dépôt Git en affichant le `git:(master)`message. Voyons le statut de ce projet:

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

- Ne vous inquiétez pas si vous voyez des messages légèrement différents - ils varient selon le système d'exploitation et la version de Git, mais l'essentiel est que Git nous dit que nous n'avons pas encore de commits, nous sommes sur la branche «master» (la branche principale) et il n'y a pas de fichiers ici à enregistrer dans le contrôle de version.

Scénario

Exemple-Demo

Ajout de fichiers

- ❑ Le moyen le plus simple de créer un fichier de test consiste à utiliser la commande Unix «**touch**».
 - Si le fichier existe, il mettra simplement à jour l'horodatage.
 - Si ce n'est pas le cas, cela créera un fichier vierge que nous pourrons ensuite ajouter au contrôle de version.
- ❑ Alors créons trois fichiers. Ils n'auront aucun contenu, mais nous leur donnerons des noms que nous pourrions utiliser lorsque nous travaillons sur un vrai projet de science des données.

Suite- exemple

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ touch preprocessing.py

Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ touch Data_analysis.ipynb

Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ touch import.py

Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Data_analysis.ipynb
    import.py
    preprocessing.py

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

Suite- exemple

- Nous sommes toujours sur la branche principale. Nous n'avons pas encore engagé (enregistré dans l'historique permanent dans Git), et les trois fichiers sont «non suivis» - Git n'y prête pas vraiment attention tant que nous ne les ajoutons pas.
- Imaginons maintenant que nous voulions faire un **commit** initial pour le fichier Jupyter Notebook (**Data_analysis.ipynb**), puis un autre commit pour les scripts d'**importation** et de **préparation**.

Suite- exemple

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git add Data_analysis.ipynb

Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git status
On branch master

No commits yet

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

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

Donc, cela nous dit que lorsque nous faisons un commit maintenant, le fichier **Data_analysis.ipynb** est celui qui sera enregistré. Faisons cela:

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git commit -m 'Add Jupyter Notebook file'
[master (root-commit) d1aef3b] Add Jupyter Notebook file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Data_analysis.ipynb
```


Remarques-Suite- exemple

- Il est important de savoir que chaque validation nécessite deux choses: un **message de validation** et au moins un fichier ajouté, modifié, renommé ou supprimé.
- En fonction de votre système d'exploitation et de la version de Git, si vous ne transmettez pas de message de validation, cela créera un message par défaut pour vous.

Note-Suite- exemple

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        import.py
        preprocessing.py

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

- Il voit que nous avons encore deux fichiers non suivis. Ajoutons-les à la zone de préparation(staging area):

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git add .
```

Note-Suite- exemple

- ❑ Il existe de nombreuses façons d'ajouter des fichiers à la «zone de préparation» dans Git.
 - Vous pouvez les nommer un par un (**git add preprocessing.py import.py**).
 - Vous pouvez faire correspondre un ensemble de fichiers en utilisant un modèle fileglob (**git add *.py**) ou vous pouvez simplement ajouter tous les fichiers dans le repo (**git add .**).
- Quelle que soit l'approche que vous adoptez, cela ajoute les deux autres nouveaux fichiers à la zone de préparation.

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   import.py
        new file:   preprocessing.py
```

Suite- exemple

- Il ne nous reste plus qu'à les commits:

```
Utilisateur@ShahinServer MINGW64 ~/Desktop/test_repo/my_test_repo (master)
$ git commit -m 'Add import and preprocessing scripts'
[master 9a526b7] Add import and preprocessing scripts
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 import.py
create mode 100644 preprocessing.py
```

- Il a fait un nouveau commit sur master (**9a526b7** dans mon cas - pour vous, ce sera différent car il est basé en partie sur le **nom d'utilisateur** et **l'email** utilisés dans ce commit et les précédents) et a ajouté deux nouveaux fichiers (mais pas de lignes de texte car dans cet exemple simple, ils étaient tous les deux des fichiers vierges).

Scénario - Le flux de travail git typique

Scénario - Le flux de travail git typique

Supposons que vous et votre équipe contribuez à un dépôt dans GitHub. Vous avez une branche **main** comme branche principale par défaut. Mais vous voulez savoir comment collaborer les uns avec les autres afin de ne pas écraser le travail des autres. Voici un de ces flux de travail comme exemple:

1. Assurez-vous d'abord que la branche **main** est à jour avec **git checkout main; git pull**.
2. Créez une nouvelle branche hors de la branche **main** avec **git checkout -b [your branch name]**.

Suite- Scénario - Le flux de travail git typique

3. Ajoutez cette nouvelle fonctionnalité ou corrigez ce bogue et faites un **git push** pour le pousser dans votre branche.
4. Lorsque vous êtes prêt à fusionner vos modifications dans la branche main, créez une **pull request** dans GitHub afin que vos coéquipiers puissent consulter votre code et proposer des suggestions d'amélioration. Si cela semble bon, GitHub vous permettra de fusionner vos modifications dans **main**.

Suite-Scénario - Le flux de travail git typique

5. Maintenant , il suffit de nettoyer votre dépôt local par:

- a) Mise à jour de votre branche locale main avec les changements qui viennent d'être fusionnés dans la branche main avec git
checkout main; git merge your-branch
- b) la suppression de la branche locale que vous travaillez à partir,
git branch -d [your branch name]

Remarques

<https://stackoverflow.com/questions/47465644/github-remote-permission-denied>

Gestionnaire d'informations d'identification

Panneau de configuration > Comptes d'utilisateurs > Gestionnaire d'informations d'identification

connectées et les réseaux.

Page d'accueil du panneau de configuration

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

autodiscover.lacapitale.com Modifié : 2018-01-15

Informations d'identification à base de certificat Ajouter des informations d'identification à base de certificat

Aucun certificat.

Informations d'identification génériques Ajouter des informations d'identification génériques

database-24cfcae1-cc45-41a1-b771-6616ec9f4867/neo4j	Modifié : 2020-03-27
MicrosoftOffice16_Data:SSPI:amami.amor.ca@gmail.c...	Modifié : 2018-01-13
outlook.office365.com	Modifié : 2018-01-13
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19
Adobe User Info(Part1)	Modifié : 2020-02-04
Adobe User Info(Part2)	Modifié : 2020-02-04
Adobe User OS Info(Part1)	Modifié : 2020-01-27

Voir aussi

Comptes d'utilisateurs

Suite

 Gestionnaire d'informations d'identification



 > Panneau de configuration > Comptes d'utilisateurs > Gestionnaire d'informations d'identification

Page d'accueil du panneau de configuration

Aucun certificat.

Informations d'identification génériques

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

database-24cfcae1-cc45-41a1-b771-6616ec9f4867/neo4j	Modifié : 2020-03-27	⌵
MicrosoftOffice16_Data:SSPI:amami.amor.ca@gmail.c...	Modifié : 2018-01-13	⌵
outlook.office365.com	Modifié : 2018-01-13	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe App Prefetched Info (QWNYb2JhdERDMXt9Mj...	Modifié : 2020-03-19	⌵
Adobe User Info(Part1)	Modifié : 2020-02-04	⌵
Adobe User Info(Part2)	Modifié : 2020-02-04	⌵
Adobe User OS Info(Part1)	Modifié : 2020-01-27	⌵
git:https://github.com	Modifié : Aujourd'hui	⌴

Adresse Internet ou réseau : git:https://github.com

Nom d'utilisateur : cegep2020

Mot de passe :

Persistance : Ordinateur local

[Modifier](#) [Supprimer](#)

Comment les **data scientists** utilisent-ils le contrôle de version?

- En tant que data scientist, même lorsque vous travaillez avec un seul fichier (par exemple un **Jupyter Notebook**), vous pouvez suivre vos modifications à l'aide d'un système de contrôle de version
- Il vous permet d'enregistrer votre travail périodiquement
- À mesure que votre projet devient plus complexe, le contrôle de version devient encore plus précieux.
- Au fil du temps, ce bloc-notes devient si rempli de petites fonctions pour nettoyer vos données importées qu'il devient difficile de se concentrer sur les parties importantes du bloc-notes.

Suite- Comment les **data scientists** utilisent-ils le contrôle de version?

- Diviser ces fonctions dans des fichiers Python distincts que vous pouvez appeler en utilisant une seule ligne de code. De cette façon, toute personne cherchant à comprendre votre projet peut obtenir une vue de haut niveau du bloc-notes Jupyter, puis elle peut toujours fouiller dans vos fichiers Python si elle souhaite comprendre les nuances de vos scripts de nettoyage de données.
- Cette stratégie facilite également l'écriture [de tests unitaires automatisés](#) pour confirmer que vos scripts de nettoyage de données appliquent les transformations que vous attendez à différents types d'entrées.

Suite- Comment les **data scientists** utilisent-ils le contrôle de version?

- Une fois que votre projet contient plusieurs fichiers que vous devez synchroniser, un système de contrôle de version comme Git est particulièrement utile car il vous permettra d'effectuer un ensemble de modifications sur plusieurs fichiers, puis de les «valider» ensemble afin que vous puissiez tous les obtenir facilement des fichiers à cet état dans le futur simplement en «récupérant» ce commit.