

Programmation et projet encadré - L7TI005

Git : introduction et préparer son environnement de travail

Yoann Dupont yoann.dupont@sorbonne-nouvelle.fr

Pierre Magistry pierre.magistry@inalco.fr

2025-2026

Université Sorbonne-Nouvelle
INALCO
Université Paris-Nanterre

Les bases



Git...

- ... est un **système de gestion de versions** ou SGV (en anglais *Version Control Software* ou *VCS*).

Git...

- ... est un **système de gestion de versions** ou SGV (en anglais *Version Control Software* ou *VCS*).
- ... permet de gérer les **modifications** effectuées sur un dossier données de manière **décentralisée**.

Git...

- ... est un **système de gestion de versions** ou SGV (en anglais *Version Control Software* ou *VCS*).
- ... permet de gérer les **modifications** effectuées sur un dossier données de manière **décentralisée**.
- ... ne versionne pas des fichiers, mais des ensembles ordonnés de modifications.

Git...

- ... est un **système de gestion de versions** ou SGV (en anglais *Version Control Software* ou *VCS*).
- ... permet de gérer les **modifications** effectuées sur un dossier données de manière **décentralisée**.
- ... ne versionne pas des fichiers, mais des ensembles ordonnés de modifications.

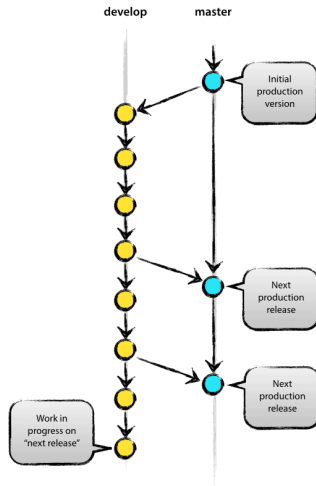


Figure 1: Une vision simplifiée d'un dépôt.
À gauche, vos changements locaux, à droite, l'état de votre dépôt distant.

Git et Github sont deux choses différentes :

- Git est un outil créé par Linus Torvald (créateur du noyau Linux)
 - programme qui sera à installer sur votre ordinateur
- GitHub est un service web construit autour de Git créé par l'entreprise Github, Inc. (rachetée par Microsoft en 2018)
 - Offre un espace de stockage en ligne pour les dépôts Git
 - Mais aussi d'autres fonctionnalités absentes de Git seul

Préparation de l'environnement

Installer git

Git est installé par défaut sur Kubuntu 23.04, rien à faire !

Si ce n'est pas le cas, deux choix :

1. aller sur Discover et cherchez *git*.
2. ouvrir un terminal¹, puis `sudo apt install git`.

¹ctrl+alt+t sous *ubuntu, launchpad → terminal sous macOS

Installer git

Git est installé par défaut sur Kubuntu 23.04, rien à faire !

Si ce n'est pas le cas, deux choix :

1. aller sur Discover et cherchez *git*.
2. ouvrir un terminal¹, puis `sudo apt install git`.

Pour vérifier que git est bien installé, on teste cette commande sur un terminal :

`git`

¹ctrl+alt+t sous *ubuntu, launchpad → terminal sous macOS

Créer un compte Github

Allez sur <https://github.com> et cliquez sur *sign up* en haut à droite/gauche (dépend de la taille de l'écran).

Créer un compte Github

Allez sur <https://github.com> et cliquez sur *sign up* en haut à droite/gauche (dépend de la taille de l'écran).

Il est possible que GitHub vous demande de configurer l'authentification à deux facteurs (2FA). Cela ne sera pas forcément nécessaire, au moins pour aujourd'hui.

Créer un compte Github

Allez sur <https://github.com> et cliquez sur *sign up* en haut à droite/gauche (dépend de la taille de l'écran).

Il est possible que GitHub vous demande de configurer l'authentification à deux facteurs (2FA). Cela ne sera pas forcément nécessaire, au moins pour aujourd'hui.

Sinon, suivez la doc : <https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/configuring-two-factor-authentication>

Prévoyez un smartphone qui scanne les QR code pour y installez une application qui gèrera les mots de passe de type TOTP (*time-based one time password*).

Parmi les apps citées par GitHub, privilégiez 1password ou Authy (évitez l'app Microsoft, GAFAM tout ça).

GitHub est une plateforme collaborative sur laquelle circule notamment du code.

- Besoin de sécuriser le contenu des dépôts (usurpation, vandalisme, etc.)
 - Sécuriser des informations privées (chiffrer)
 - Authentification de l'auteur·ice d'un message (signer)
- Une solution prévue dans github consiste à ajouter une clé de sécurité

Aussi appelée Cryptographie à Clé Publique (*Public Key Cryptography*).

Sans rentrer dans les détails techniques, il faut comprendre la finalité :

- Vous avez deux clés : une **publique** (que tout le monde peut voir) et une **privée** (qui doit n'être accessible qu'à vous).
- cela permet de chiffrer et signer ce que vous envoyez vers / recevez de Github.

GitHub devient de plus en plus strict avec la sécurité :

- Vous aurez sans doute besoin d'une clé pour pousser vos changements.
- Type de chiffage recommandé : Ed25519

GitHub devient de plus en plus strict avec la sécurité :

- Vous aurez sans doute besoin d'une clé pour pousser vos changements.
- Type de chiffage recommandé : Ed25519

Comment utiliser ces clés sur GitHub ?

1. Créer la paire clé publique/privée sur votre ordinateur
2. Donner la clé **publique** à GitHub

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

source : <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

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

La commande vous demandera alors :

1. Où sauvegarder la clé (par défaut la clé privée dans `~/.ssh/id_ed25519` et la publique dans `~/.ssh/id_ed25519.pub`)
2. Un mot de passe pour la clé (qu'il faudra entrer à chaque fois)
 - 2.1 On peut ne rien donner, mais pas recommandé

source : <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

Ajouter la clé à votre agent SSH

Vérifier que votre agent SSH tourne :

```
$ eval "$(ssh-agent -s)"
```

source : <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

Ajouter la clé à votre agent SSH

Vérifier que votre agent SSH tourne :

```
$ eval "$(ssh-agent -s)"
```

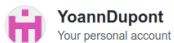
Affichera un message type "Agent pid XXXX". Si tel est le cas, on ajoute la clé à l'agent (en supposant que ~/.ssh/id_ed25519 est votre clé privée) :

```
ssh-add ~/.ssh/id_ed25519
```

source : <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

Ajouter la clé à GitHub

Finalement, on ajoute la clé **publique** (termine par .pub) à GitHub via les *settings*.



Public profile

Account

Appearance

Accessibility

Notifications

Access

Billing and plans

Emails

Password and authentication

SSH and GPG keys

Organizations


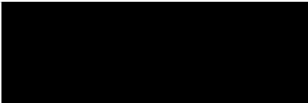

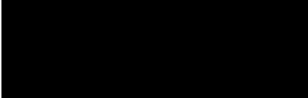
Moderation

SSH keys

New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

Authentication Keys

 SSH		Delete
 SSH		Delete

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH problems](#).

source : <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>

Les commandes git



Git fournit un ensemble de commandes qui permettent de gérer les changements.

La syntaxe générale prend la forme suivante :

```
git <sous-commande> [-options...] [arguments...]
```


Git fournit un ensemble de commandes qui permettent de gérer les changements.

La syntaxe générale prend la forme suivante :

```
git <sous-commande> [-options...] [arguments...]
```

Les commandes git sont paramétrables, on ne traitera ici que des cas de base.

En accord avec la philosophie Unix, les commandes git sont également très précises et font le moins de choses possibles.

1ère commande avec exemple : git clone

Permet de créer une copie d'un dépôt existant sur sa machine.

Cette copie peut de suivre et de faire suivre les changements apportés au dépôt. On a donc un **clone** du dépôt distant.

```
git clone [-options...] <URL>
```

Où <URL> est le lien vers un dossier git (généralement distant).

1ère commande avec exemple : git clone

Permet de créer une copie d'un dépôt existant sur sa machine.

Cette copie peut de suivre et de faire suivre les changements apportés au dépôt. On a donc un **clone** du dépôt distant.

```
git clone [-options...] <URL>
```

Où <URL> est le lien vers un dossier git (généralement distant).

Lier un dépôt de travail à son compte GitHub

En supposant que USER soit votre identifiant GitHub et EMAIL votre email :

```
git config user.name "USER"  
git config user.email "EMAIL"
```

On peut mettre l'option `-global` pour ne pas faire ces deux lignes à chaque fois.

On dit bien un clone, pas un fork !
Si vous forcez, vous ne verrez aucune mise à jour.

Petit test - récupérer les slides du cours

Créez-vous un dossier plural quelque part **PAS SUR LE BUREAU**.

Allez dedans et tapez la commande suivante :

```
git clone https://github.com/YoannDupont/PPE1-2025.git
```

Inutile de lier vos identifiants à ce dépôt, vous ne pourrez pas pousser.

Permet de mettre-à-jour votre branche vers la bonne version (par défaut : la dernière version en ligne).

En termes git, on **tire** les changements du dépôt distant vers notre répertoire local.

```
git pull
```

Permet de mettre-à-jour les métadonnées du dépôt sur votre répertoire local.

Le terme **fetch** fait sans doute référence à **play fetch** (jouer à la ramener balle).

```
git fetch
```

Permet de mettre-à-jour les métadonnées du dépôt sur votre répertoire local.

Le terme **fetch** fait sans doute référence à **play fetch** (jouer à la ramener balle).

```
git fetch
```

`git fetch` a quelques avantages par rapport à `pull` :

- Fonctionne toujours (ajoute simplement plus de métadonnées).
- Ne modifie pas les fichiers du dépôt.
- Permet d'anticiper les conflits (changements incompatibles).

Permet d'indiquer des fichiers dont on veut suivre les modifications avant validation.

En terminologie git, on **ajoute au suivi** des modifications faites sur des fichiers. On appelle cette étape la mise-en-place (*staging*).

```
git add <FILE...>
```

Où <FILE...> est un ou plusieurs fichiers dont on souhaite suivre les modifications

Permet d'indiquer des fichiers dont on veut suivre les modifications avant validation.

En terminologie git, on **ajoute au suivi** des modifications faites sur des fichiers. On appelle cette étape la mise-en-place (*staging*).

```
git add <FILE...>
```

Où <FILE...> est un ou plusieurs fichiers dont on souhaite suivre les modifications

Attention :

git add suit les modifications **passées, mais pas futures**. Si vous modifiez un fichiers après un add, ces nouveaux changements ne seront pas suivis.

Permet de **retirer du suivi** (*remove*) un fichier. Cela équivaut à supprimer le fichier des futures versions.

```
git rm <FILE...>
```

Où <FILE...> est un ou plusieurs fichiers dont on souhaite supprimer du dépôt. Retirer un fichier via `git rm` met en place le changement automatiquement.

Permet de **retirer du suivi** (*remove*) un fichier. Cela équivaut à supprimer le fichier des futures versions.

```
git rm <FILE...>
```

Où <FILE...> est un ou plusieurs fichiers dont on souhaite supprimer du dépôt. Retirer un fichier via `git rm` met en place le changement automatiquement.

Attention :

- Un fichier retiré du dépôt peut être à nouveau ajouté par la suite.
- Un fichier retiré du dépôt ne disparaît pas totalement : il demeure accessible sur les versions précédentes.

Permet de valider les modifications des fichiers suivis.

Pourquoi *commit* ? De nouvelles modifications peuvent être faites après un commit, mais git **se tiendra** aux modifications validées.

```
git commit [-m message]
```

Où message est le message qui décrit les changements que vous faites.

Permet de valider les modifications des fichiers suivis.

Pourquoi *commit* ? De nouvelles modifications peuvent être faites après un commit, mais git **se tiendra** aux modifications validées.

```
git commit [-m message]
```

Où message est le message qui décrit les changements que vous faites.

Attention :

Si aucun message n'est donné, git ouvrira un éditeur de texte dans le terminal pour que vous puissiez écrire. Ça peut faire de mauvaises surprises.

Envoie les modifications mises en place (ou "commitées") vers le dépôt distant.

On dit, en git, qu'on **pousse** les modifications.

```
git push
```

Envoie les modifications mises en place (ou "commitées") vers le dépôt distant.

On dit, en git, qu'on **pousse** les modifications.

```
git push
```

On push le moins possible mais autant que nécessaire ! Un commit pushé sera toujours plus dur à corriger qu'un commit uniquement sur votre ordi.

Permet de voir les changements de votre dossier par rapport à la version du dépôt.

```
git status
```

Cela affichera les fichiers mis en place (*staged*) et les modifications non suivies.

Permet de voir l'ensemble des *commits* ayant été effectués sur le dépôt du plus récent au plus ancien.

```
git log
```

Log vous indiquera notamment les auteur·ices, dates et messages des commits. Il faut appuyer sur la touche "q" pour sortir du log.

Mettre à jour un dépôt depuis votre ordinateur

Le travail sur un dépôt git suit une certaine routine. Le scénario le plus simple :

- on répète plusieurs fois :
 - `git pull`
 - On fait des modifications...
 - `git add / git rm`
 - `git commit`
 - `git push`

Mettre à jour un dépôt depuis votre ordinateur

Le travail sur un dépôt git suit une certaine routine. Le scénario le plus simple :

- on répète plusieurs fois :
 - `git pull`
 - On fait des modifications...
 - `git add / git rm`
 - `git commit`
 - `git push`
- quand on a fini :
 - `git tag`

Mettre à jour un dépôt depuis votre ordinateur

Le travail sur un dépôt git suit une certaine routine. Le scénario le plus simple :

- on répète plusieurs fois :
 - `git pull`
 - On fait des modifications...
 - `git add / git rm`
 - `git commit`
 - `git push`
- quand on a fini :
 - `git tag`

`git tag ?`

GitHub : Les étiquettes

Quoi et pourquoi

Dans la longue chaîne des modifications, il peut être difficile de savoir si des commits sont plus intéressants que d'autres.

- Les étiquettes (*tag*) permettent de marquer un commit particulier.
- La valeur et la signification du tag dépend uniquement des mainteneurs.
- GitHub transforme cependant les tags en "*release*", ce qui est un parti pris.

source : <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

Quoi et pourquoi

Dans la longue chaîne des modifications, il peut être difficile de savoir si des commits sont plus intéressants que d'autres.

- Les étiquettes (*tag*) permettent de marquer un commit particulier.
- La valeur et la signification du tag dépend uniquement des mainteneurs.
- GitHub transforme cependant les tags en "*release*", ce qui est un parti pris.

Utilisation des tags dans ce cours :

Vous utiliserez les tags pour nous indiquer **uniquement** les versions finies de vos travaux pour que nous allions les vérifier.

source : <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

On peut ajouter un tag à un commit :

```
git tag [-a] [-m message] <tagname> [commit]
```

- tagname est le nom du tag (seul élément non-optionnel).
- [commit] indique le commit qu'on veut tagger (sinon, le commit courant).
- -a permet d'annoter un tag avec un message donné par -m.

source : <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

On peut ajouter un tag à un commit :

```
git tag [-a] [-m message] <tagname> [commit]
```

- tagname est le nom du tag (seul élément non-optionnel).
- [commit] indique le commit qu'on veut tagger (sinon, le commit courant).
- -a permet d'annoter un tag avec un message donné par -m.

Pousser un tag vers GitHub :

```
git push origin <tagname>
```

source : <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

On peut ajouter un tag à un commit :

```
git tag [-a] [-m message] <tagname> [commit]
```

- tagname est le nom du tag (seul élément non-optionnel).
- [commit] indique le commit qu'on veut tagger (sinon, le commit courant).
- -a permet d'annoter un tag avec un message donné par -m.

Pousser un tag vers GitHub :

```
git push origin <tagname>
```

à faire maintenant avec le <tagname> *seance1*.

source : <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

Dans les cas simples, on a tendance à suivre cette façon de faire :

- plusieurs modifications locales \implies un *add*

Fonctionnement de "base"

Dans les cas simples, on a tendance à suivre cette façon de faire :

- plusieurs modifications locales \implies un *add*
- 1 à plusieurs *add* \implies 1 *commit*

Dans les cas simples, on a tendance à suivre cette façon de faire :

- plusieurs modifications locales \implies un *add*
- 1 à plusieurs *add* \implies 1 *commit*
- 1 à plusieurs *commits* \implies 1 *push*

Dans les cas simples, on a tendance à suivre cette façon de faire :

- plusieurs modifications locales \implies un *add*
- 1 à plusieurs *add* \implies 1 *commit*
- 1 à plusieurs *commits* \implies 1 *push*
- 1 à plusieurs *push* \implies 1 *tag*

Dans les cas simples, on a tendance à suivre cette façon de faire :

- plusieurs modifications locales \implies un *add*
- 1 à plusieurs *add* \implies 1 *commit*
- 1 à plusieurs *commits* \implies 1 *push*
- 1 à plusieurs *push* \implies 1 *tag*

Soyez certain(e) de ce que vous faites quand vous créez un *tag*. On s'en servira pour avoir vos exercices finis.

Et maintenant la pratique (git along)

Deuxième partie



Buts de cette partie :

- Apprendre à corriger des erreurs en git
- Apprendre à gérer certains conflits

Annonce



Si vous avez effectué votre inscription complémentaire à Sorbonne Nouvelle, vous avez un mail @sorbonne-nouvelle.fr et un accès à icampus.

Lien du cours :

<https://icampus.univ-paris3.fr/course/view.php?id=74259>

clé d'inscription (autoinscription) : PPE1@2526

GitHub : Corriger des erreurs

Quels moyens pour quelles erreurs.

Il est normal de faire des erreurs en git. En fonction de différents cas, on va vous apprendre à les corriger.

Quels moyens pour quelles erreurs.

Il est normal de faire des erreurs en git. En fonction de différents cas, on va vous apprendre à les corriger.

Nous allons utiliser les commandes pour cela :

- `git reset`
- `git revert`
- `git stash`

Quels moyens pour quelles erreurs.

Il est normal de faire des erreurs en git. En fonction de différents cas, on va vous apprendre à les corriger.

Nous allons utiliser les commandes pour cela :

- `git reset`
- `git revert`
- `git stash`

De manière générale, il est recommandé d'utiliser régulièrement `git fetch` pour récupérer des métadonnées du dépôt. Cette commande a l'avantage de ne pas pouvoir échouer (au niveau de git).

Un peu de syntaxe avant

Quelques éléments à savoir avant de continuer :

- HEAD : représente le commit sur lequel vous êtes en train de travailler
- origin/main : représente le dernier commit poussé en ligne² (faites un "fetch origin" avant)
- <tag> : représente le commit sur lequel on a placé l'étiquette
- ~[N] : représente l'ascendance directe de votre commit (linéaire, par défaut N=1 représente le commit parent)
- ^[N] : représente le n-ième parent du commit (non linéaire, par défaut N=1 représente le commit parent)

source : <https://git-scm.com/docs/git-rev-parse>

²techniquement, c'est le nom de la branche principale, mais on verra la différence au S2.

Un peu de syntaxe avant

Quelques éléments à savoir avant de continuer :

- HEAD : représente le commit sur lequel vous êtes en train de travailler
- origin/main : représente le dernier commit poussé en ligne² (faites un "fetch origin" avant)
- <tag> : représente le commit sur lequel on a placé l'étiquette
- ~[N] : représente l'ascendance directe de votre commit (linéaire, par défaut N=1 représente le commit parent)
- ^[N] : représente le n-ième parent du commit (non linéaire, par défaut N=1 représente le commit parent)

On peut faire des choses très précises, on se contentera de travailler ici dans l'ascendance directe.

source : <https://git-scm.com/docs/git-rev-parse>

²techniquement, c'est le nom de la branche principale, mais on verra la différence au S2.

`git reset` permet de faire machine arrière dans les commits. La commande permet de revenir dans le passé commit par commit.

`git reset` permet de faire machine arrière dans les commits. La commande permet de revenir dans le passé commit par commit.

Ce semestre, on utilisera surtout cette commande pour annuler un/plusieurs commits non poussés, mais la commande a d'autres cas d'usage, qu'on pourra voir au S2.

Défaire jusqu'à des commits non poussés (gentil)

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Défaire jusqu'à des commits non poussés (gentil)

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

```
git reset --soft HEAD~
```

Revient à dernière la version du dépôt mais n'annule la mise-en-place (*staging*).

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Défaire jusqu'à des commits non poussés (gentil)

```
git reset HEAD~
```

Revient à dernière la version du dépôt et annule la mise-en-place (*staging*).

```
git reset --soft HEAD~
```

Revient à dernière la version du dépôt mais n'annule la mise-en-place (*staging*).

Attention :

`git reset` fonctionne sur des commits entiers, pas sur des fichiers spécifiques.

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Défaire jusqu'à un commit non poussé (méchant)

```
git reset --hard
```

Revient à la version HEAD. Vous perdrez tous les changements que vous avez fait.

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Revenir à un commits spécifique

```
git reset <commit>
```

Où <commit> peut être :

- l'identifiant SHA du commit (longue chaîne de lettre et nombres).
- un tag

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Revenir à un commits spécifique

```
git reset <commit>
```

Où <commit> peut être :

- l'identifiant SHA du commit (longue chaîne de lettre et nombres).
- un tag

Les options `soft` et `hard` s'appliquent comme précédemment.

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Revenir à un commits spécifique

```
git reset <commit>
```

Où <commit> peut être :

- l'identifiant SHA du commit (longue chaîne de lettre et nombres).
- un tag

Les options `soft` et `hard` s'appliquent comme précédemment.

Exemple :

```
git reset origin/main
```

Pour revenir au dernier commit synchronisé en ligne (plus de précisions au S2).

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Défaire un commit spécifique

Là où `git reset` fonctionne de manière "linéaire", `git revert` permet de sélectionner un commit particulier et de le *défaire*.

Défaire un commit spécifique

Là où `git reset` fonctionne de manière "linéaire", `git revert` permet de sélectionner un commit particulier et de le *défaire*.

Quand on parle de *défaire* un commit, c'est au sens strict : `git revert` va créer un nouveau commit où tous les changements seront joués à l'envers. Les ajouts sont supprimés et les suppressions sont ajoutées.

Défaire un commit spécifique

```
git revert <commit>
```

Où <commit> peut être :

- l'identifiant SHA du commit (longue chaîne de lettres et nombres).
- un tag

Crée un nouveau commit où les changements sont annulés.

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Pour que des modifications soient acceptées sur un dépôt en ligne, il faut que nos modifications soient faites sur la dernière version (la version à jour). Si nous ne sommes pas à jour, git va simplement refuser nos modifications.

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Pour que des modifications soient acceptées sur un dépôt en ligne, il faut que nos modifications soient faites sur la dernière version (la version à jour). Si nous ne sommes pas à jour, git va simplement refuser nos modifications.

Il est possible de récupérer les changements avec un `git pull`, mais cette commande peut échouer si nous avons des modifications en conflit (au moins un fichier modifié en local et en ligne). La seule solution est alors de nettoyer votre dépôt local pour retirer tous les conflits.

Conserver des modifications pour les appliquer plus tard

Git permet de travailler de manière désynchronisée et collaborative. Cela veut dire que des changements peuvent provenir de plusieurs sources.

Pour que des modifications soient acceptées sur un dépôt en ligne, il faut que nos modifications soient faites sur la dernière version (la version à jour). Si nous ne sommes pas à jour, git va simplement refuser nos modifications.

Il est possible de récupérer les changements avec un `git pull`, mais cette commande peut échouer si nous avons des modifications en conflit (au moins un fichier modifié en local et en ligne). La seule solution est alors de nettoyer votre dépôt local pour retirer tous les conflits.

Pour éviter de perdre ses modifications, il est possible de les mettre de côté (*stash*) pour les ressortir plus tard.

```
git stash push [-m <message>]
```

Où :

- -m <message> permet de mettre un message spécifique

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

```
git stash push [-m <message>]
```

Où :

- `-m <message>` permet de mettre un message spécifique

`git stash` va mettre de côté vos modifications dans un index. Chaque appel à *git stash push* crée une nouvelle entrée.

source : <https://til.bhupesh.me/git/how-to-undo-anything-in-git>

```
git stash list
```

permet de voir la liste des paquets de modifications mis de côté.

```
git stash list
```

permet de voir la liste des paquets de modifications mis de côté.

```
git stash show [-p] <stash>
```

Où :

- `-p` permet d'afficher un diff avec le stash
- `<stash>` est un identifiant de stash (typiquement son indice)

permet de voir le contenu d'un *stash*.

```
git stash apply <stash>  
git stash pop <stash>
```

Permettent d'appliquer les changements contenus dans un *stash*.

```
git stash apply <stash>  
git stash pop <stash>
```

Permettent d'appliquer les changements contenus dans un *stash*.

Le différence en *apply* et *pop* est que *pop* supprime le *stash* une fois appliqué.


```
git stash apply <stash>  
git stash pop <stash>
```

Permettent d'appliquer les changements contenus dans un *stash*.

Le différence en *apply* et *pop* est que *pop* supprime le *stash* une fois appliqué.

```
git stash drop <stash>
```

Supprime manuellement un *stash*.

Exercices :

- Faire la feuille d'exercices
02-git-more-exercices.pdf
- Pensez à pousser les tags créés