

Git, *essentials*

Ce cours *n'est pas une introduction à git*. Cette partie est simplement une révision des concepts et commandes principales de git côté utilisateur·ice

Installation, outils avec interface graphique

Installation de git

Ubuntu/Debian :

```
sudo apt install git
```

Windows

Installer [Git SCM to Windows](#)

Alternative sur Windows: avec (<https://learn.microsoft.com/en-us/windows/wsl/install>) (Windows Subsystem for Linux)

Ouvrir une invite de commande Powershell (configuration d'une machine virtuelle requise)

```
wsl --install
```

Voir [la documentation](#) si intéressé.e.

Installation de git

macOS

Avec le gestionnaire de paquet [brew](#) (recommandé)

```
brew install git
```

ou avec [MacPorts](#)

```
sudo port install git
```

Outils supplémentaires: qgit

[qgit](#), un client git graphique pour visualiser les log et l'historique

Ubuntu/Debian:

```
sudo apt install qgit
```

Windows :

- Suivre les instructions du [dépôt officiel](#)
- Via [sourceforge](#) (non testé)

Voir [la liste de nombreux clients git avec interface graphique disponibles](#) sur le site officiel. Ici une [liste exhaustive](#).

Outils supplémentaires: tig

[tig](#) est une interface texte (CLI). C'est un navigateur de dépôt git en ligne de commande.

Sur Windows, il est nativement intégré dans [Git SCM to Windows](#) !

Debian/Ubuntu

```
sudo apt install tig
```

macOS (avec brew)

```
brew install tig
```

Configuration

Les différentes configurations de git

3 niveaux de configuration:

- **Local** au dépôt. Emplacement `.git/config`.
- Propre à l'**utilisateur** courant (de la machine)
`~/.gitconfig`
- Pour le **système** `/etc/gitconfig`

Lister les configurations d'un dépôt

```
#toutes les configs appliqués au dépôt
git config -l
#config locale au dépôt
git config --local -l
```

```
[15:30] genesis : ~/tmp/git $
cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
```


Configuration des **user**

La configuration d'un user **global** (machine)

```
git config --global user.name "nom user"  
git config --global user.email "mail user"
```

La configuration d'un user local au dépôt (si plusieurs users sur une même machine)

```
git config --local user.name "nom user"  
git config --local user.email "mail user"
```

L'user sert juste à vous identifier sur l'historique (arbre des commits), pas à l'authentification sur GitHub ou autre !

Configuration de l'éditeur

Pour configurer l'éditeur de texte utilisé par git pour apporter des modifications (commit de merge, amend, rebase interactif, etc.)

```
git config --global core.editor <votre éditeur de texte favori>
```

Choisissez un éditeur de texte simple dans lequel vous êtes à l'aise

(Savoir) Lire et utiliser la documentation avec `man git <command>`

À tout moment, lire la documentation sur chaque commande avec `man git <command>`, par exemple `man git reset`

- Lister les commandes de navigation dans `man` :
`man man`, puis `h` pour help
- Rechercher par pattern: `/search`
- Occurrence suivante: `n`
- Occurrence précédente: `N`

`git <command> -h` vous fournira aussi une documentation plus succincte et pragmatique.

```
GIT(1)                               Git Manual                               GIT(1)

NAME
    git - the stupid content tracker

SYNOPSIS
    git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        [--super-prefix=<path>] [--config-env=<name>=<envvar>]
        <command> [<args>]

DESCRIPTION
    Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals.

    See gittutorial(7) to get started, then see giteveryday(7) for a useful minimum set of commands. The Git User's Manual[1] has a more in-depth introduction.
```

Les commandes de base à connaître

Sélection de commandes *utilisateur·ice* à savoir utiliser. Suffisantes dans 95% des cas

Local (sur votre machine)

Commandes liées aux commits

```
git add
git status
git mv
git rm
git diff
git commit
git cherry-pick
git blame
git log
git stash
git reset
```

Commandes pour la gestion des branches

```
git branch
git checkout
git merge
git rebase
git tag
```

Local (sur votre machine)

Commandes de configuration

```
git remote  
git config  
git init
```

Les commandes de base à connaître

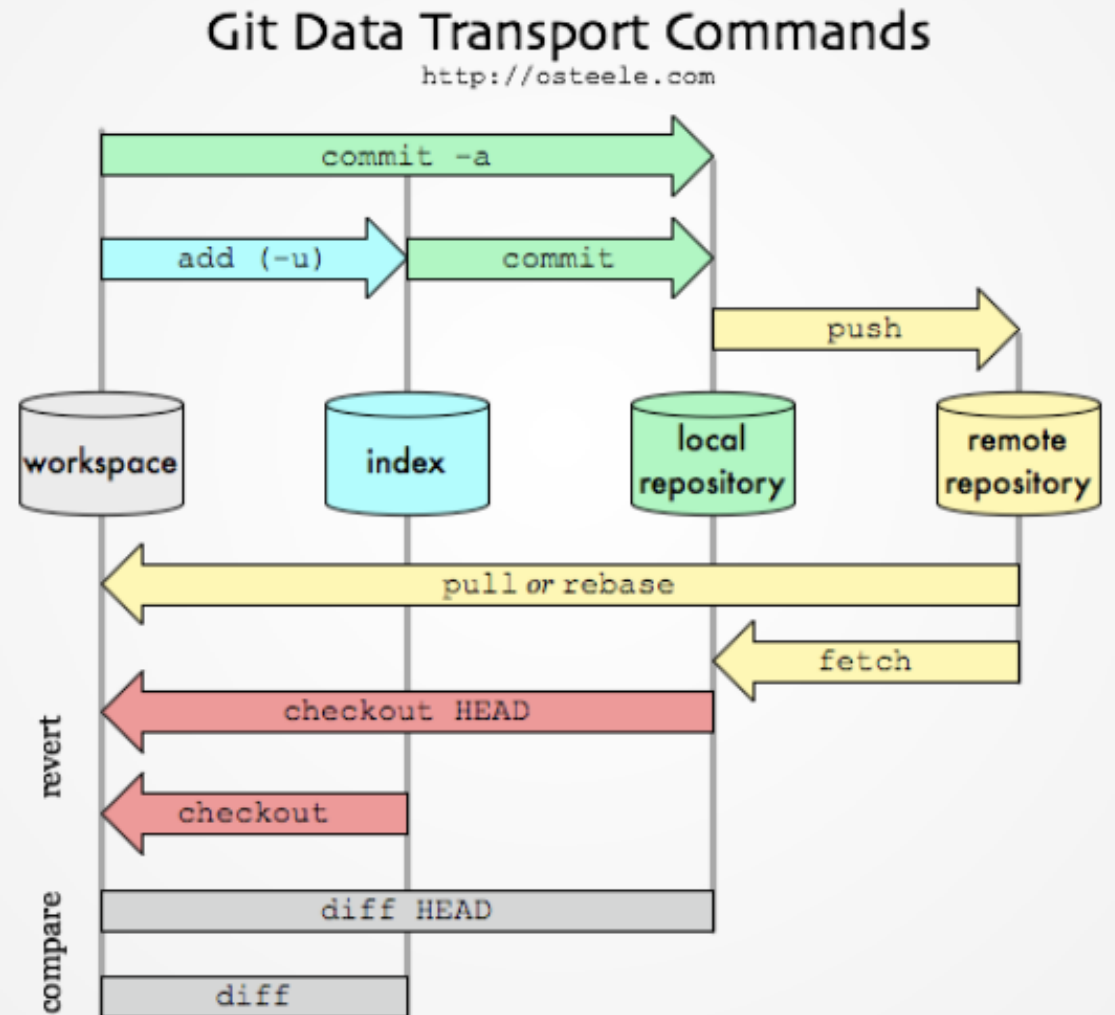
A distance. Commande pour gérer le dépôt distant (*remote*)

```
git clone  
git fetch  
git pull  
git push
```

La [liste de toutes les commandes](#) sur le site officiel

Les différents *espaces*

- Le workspace : le système de fichiers sur votre machine hébergeant le dépôt
- Le working tree : les fichiers trackés par git
- La staging area ou index : là où sont enregistrés les changements à tracker
- Le dépôt local (repositoy) : là où sont commit les changements (historique)
- Le dépôt remote (optionnel) : une copie du dépôt sur une autre machine



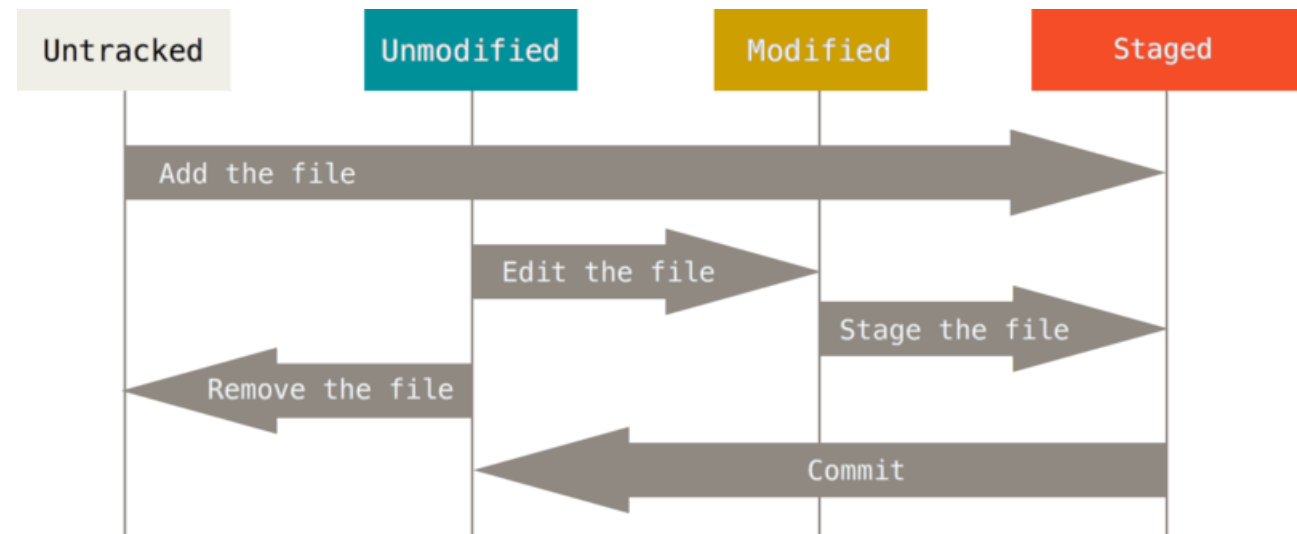
Workflow en local

Status des fichiers

`git status` permet de voir l'état du *working tree*.

L'état *staged* correspond à la salle d'attente (*staged area*) du commit. C'est dans l'état *staged* qu'on accumule les changements à apporter au dépôt pour un *commit*.

staged area = l'index



Exercice et `diff`

- Créer un nouveau dépôt dans un dossier vide
- Créer un fichier `foo` avec le contenu `'Hello'`
- Tracker le fichier `foo`
- Modifier le fichier `foo` en remplaçant `'Hello'` par `'Hello, how are you ?'`
- Afficher la différence avec `diff`
 - entre le staging area et le working tree
 - entre le staging area et le repository
- Tracker les dernières modification apportées au fichier `foo`
- Que retourne `git diff` ? `git diff --cached` ?
- Renommer et stager le fichier `foo` en `bar` **en une seule commande**
- Créer un nouveau fichier `bat`
- Renommer le en `baz`. Peut on utiliser `git mv bat baz` ? Pourquoi ?

Correction

```
mkdir tmp
cd tmp
git init
echo 'Hello' > foo
#tracker le fichier foo (l'ajouter au working tree)
git add foo
echo 'Hello, how are you ?' > foo
#diff entre le staging area et le working tree
git diff
#diff entre le staging area et le repository
git diff --cached
touch bat
# On ne peut pas faire git mv bat baz car le fichier bat n'est pas tracké par git(pas dans le working tree)
```

| `--staged` est un synonyme de `--cached`. Donc `git diff --cached` et `git diff --staged` font la même chose

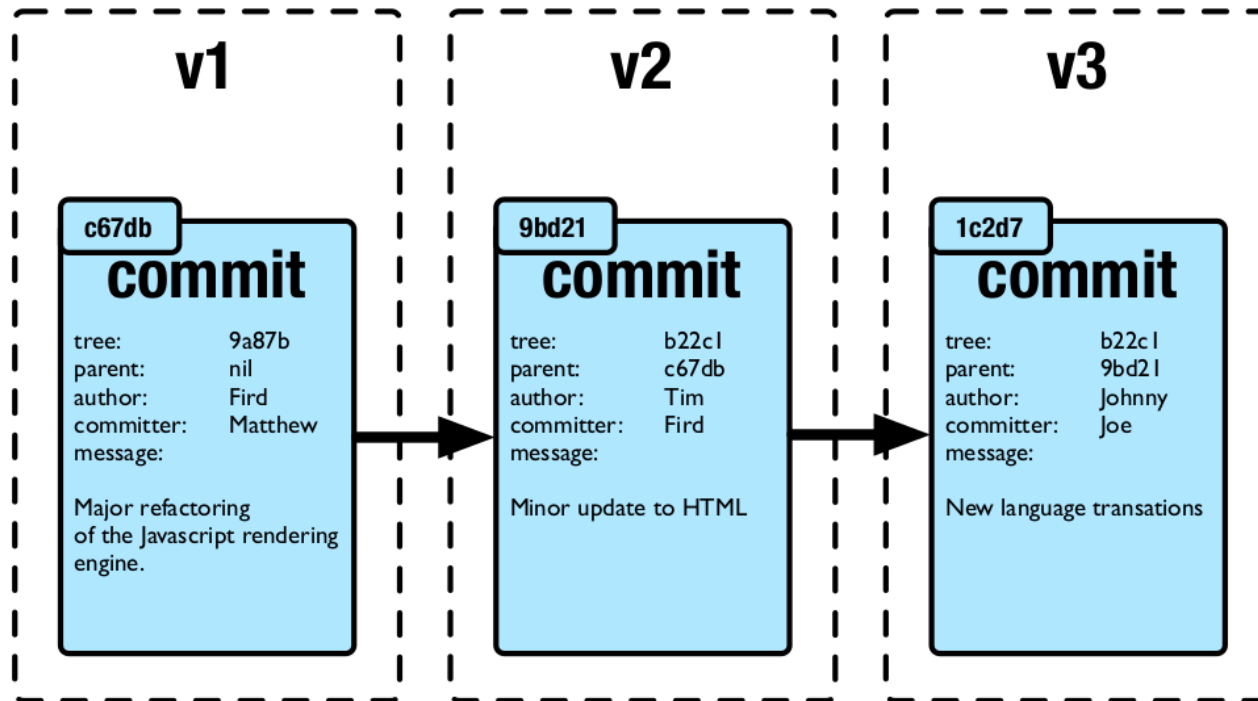
(Rappels sur) Les commit

Le **commit** est la brique élémentaire de git.

Un *commit*:

- est une **photographie de l'état du projet, inséré dans l'historique**
- représente l'état courant de l'index (changements apportés)
- peut avoir 0, 1 ou 2 commits parents
- est créée sur un ordinateur, *par quelqu'un*, **localement**
- est identifié par un identifiant unique: un hash **SHA1**. Par exemple `9AB223D28B1AA46EF1780B22F304982E39872C34`

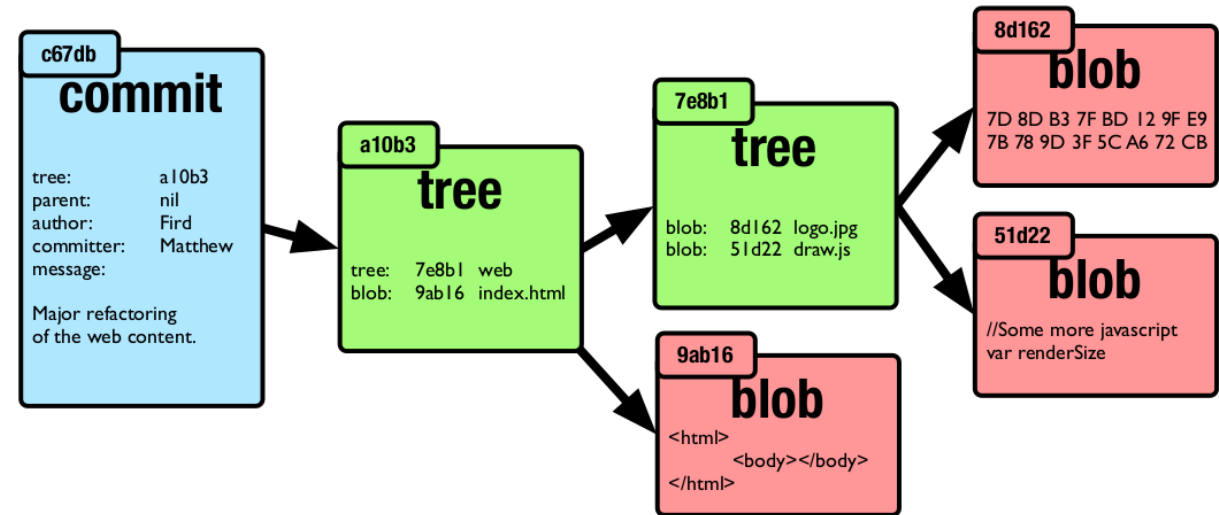
Historique: l'arbre des commits



Plomberie de git : les objets manipulés par git

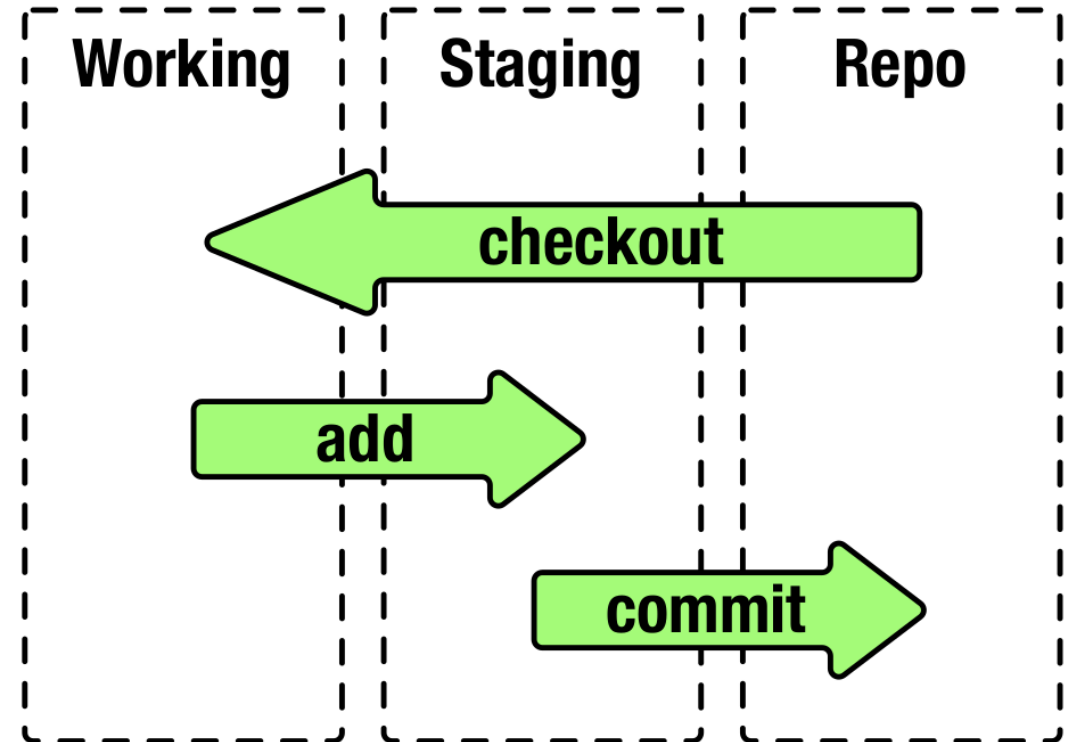
- **Blob**: équivalent d'un fichier
- **Tree**: équivalent d'un dossier (structure récursive)
- **Commit**: snapshot
- **Tag**: label d'un commit

Pour en savoir plus sur les rouages internes de git, consulter [cet excellent ouvrage en ligne *Git from the bottom up*, de John Wiegley](#). Nous nous arrêterons à cette couche d'abstraction pour travailler.



Un travail en 3 temps

- On récupère un état du dépôt (working tree) (`git checkout`)
- On modifie les fichiers (édition)
- On enregistre les modifications apportées dans la staging area (`git add`)
- On commit les changements pour valider et ajouter les changements dans l'historique (`git commit`)



Exercice 1: quelques étirements

1. **Créer** un dépôt local qui contiendra vos notes sur ce cours
2. **Créer** un fichier `README.md` et renseigner le titre
3. **Ajouter** ce fichier au *stage*
4. **Faire** un commit
5. **Inspecter** les commits avec `git log`
6. **Créer** un fichier `foo.md`
7. **Ajouter** ce fichier au *stage*
8. Finalement nous avons décidé de le supprimer. **Supprimer** le avec `git rm`. Que remarquez-vous ? Quelles sont vos deux options ? **Finaliser** le commit puis supprimer le fichier dans un nouveau commit. Inspecter le contenu *du dossier*
9. À l'aide de `checkout`, **se déplacer** dans l'état précédent (où le fichier `foo.md` existe)
10. **Revenir** au dernier commit réalisé avec `checkout`

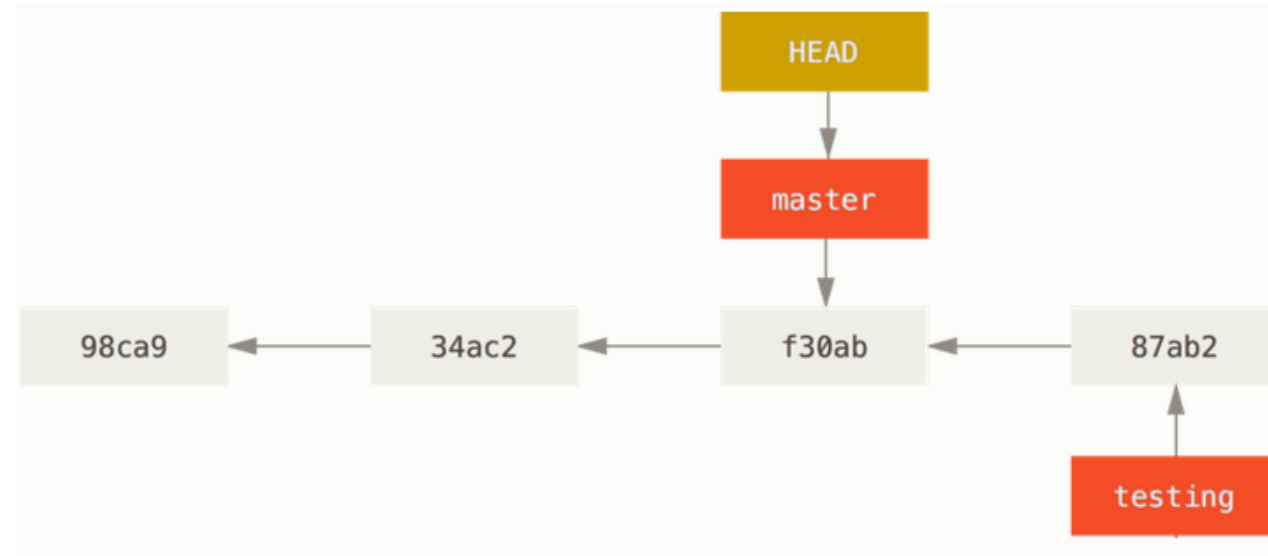
Les branches, le cœur de Git

Les **branches** permettent de travailler à plusieurs sur un même dépôt de *façon isolée*.

Une **branche** est une **étiquette (un tag particulier) sur un commit**, qui se déplace toujours *à la pointe*.

Git permet de:

- Créer et supprimer des branches
- De les **fusionner** (2 méthodes : via `merge` ou `rebase`)



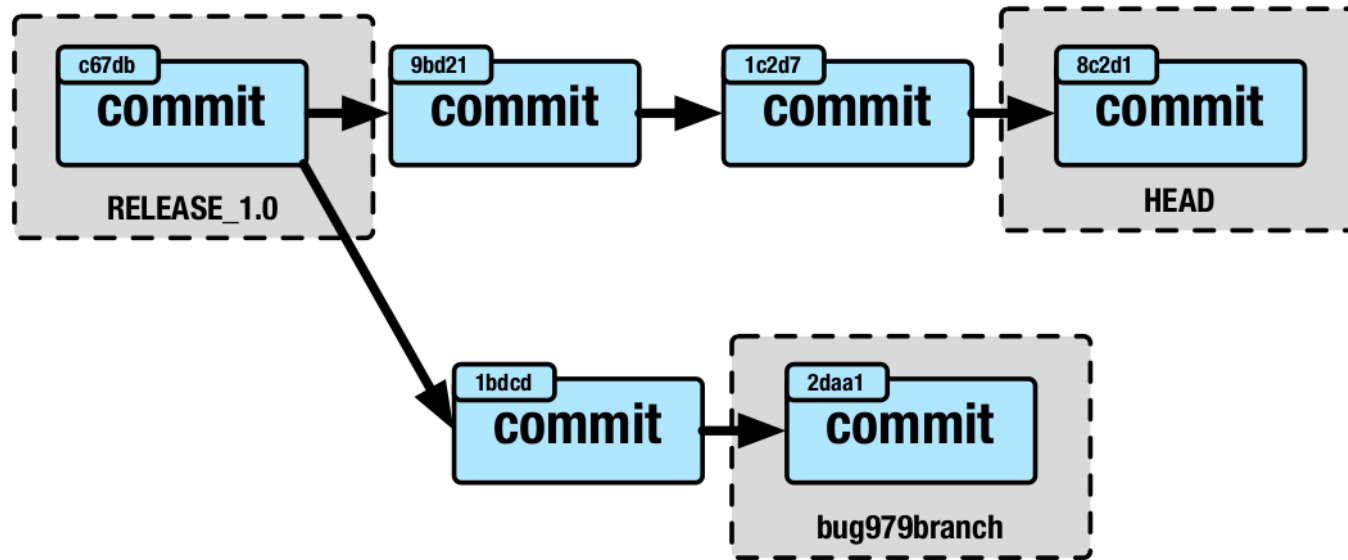
Exercice (20min)

Lire [cette page de la documentation officielle](#) et reproduire l'exemple proposé et discuté, pour bien maîtriser la notion de branche.

Gérer les branches locales

```
# Lister les branches locales
git branch # ou git branch -l
# Créer une branche
git branch foo
# Se déplacer sur une branche
git checkout foo
# Renommer une branche *foo* en *bar*
git branch -m foo bar
# Supprimer une branche (impossible de scier la branche sur laquelle vous êtes assis.e !)
git checkout main
git branch -d bar
```

Les tags



Lister les tags, ajouter un tag `v0.1` sur `HEAD` avec le message "app v 0.1"

```
git tag -l  
git tag -a v0.1 -m "app v 0.1"
```

Les branches sont un *cas particulier* de tags. Un tag est une référence (une étiquette) fixe, attachée à un commit, contrairement à une branche qui se déplace automatiquement sur le commit le plus récent

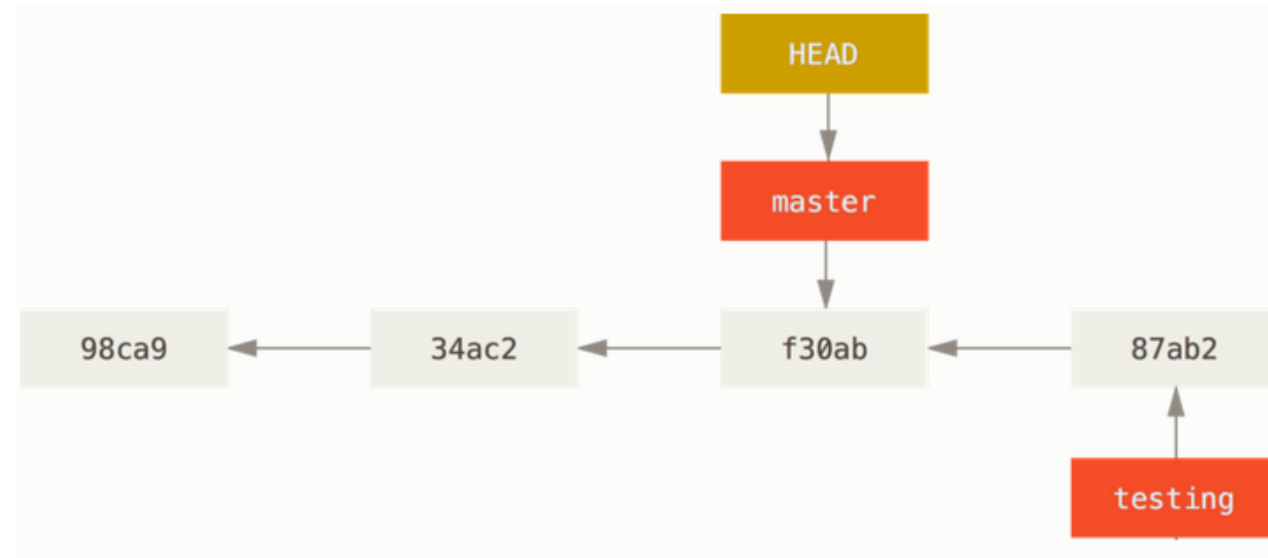
Le pointeur HEAD *

HEAD est un nom symbolique qui pointe toujours sur le commit le plus récent par défaut.

HEAD est déplacé lors d'un checkout . Par exemple, `git checkout master` déplace HEAD pour le faire pointer sur la branch master .

Normalement HEAD pointe sur le nom d'une branche (mais pas toujours, il peut pointer sur n'importe quel commit)

Détacher HEAD : attacher HEAD à un commit au lieu d'une branche



Opérations

Gérer le stash

Remiser les modifications d'un répertoire de travail *sale*.

Utilisez `git stash` lorsque vous voulez conserver l'état actuel du répertoire de travail et de l'index, mais que vous voulez revenir à un répertoire de travail propre (dernier checkout).

```
# modification du working tree (edition des fichiers)
git stash
#faire autre chose, puis revenir
git stash pop
```


Annuler les commits avec **reset**

```
# rembobiner la branche courante d'un commit. Les changements sont gardés en *stage*  
git reset --soft HEAD~1  
# rembobiner la branche courante d'un commit. Les changements sont supprimés  
git reset --hard HEAD~1  
# rembobiner la branche courante jusqu'au commit <hash>  
git reset --hard <hash>
```

Annuler les commits avec **reset** : Démo

```
git init
echo 'Foo' > foo
git add foo
git commit -m "first commit"
echo 'Bar' > foo
git commit -am "second commit"
git log
#observer la différence
git reset --soft HEAD~1
git log
git status
#les changements sont conservés
cat foo
git commit -m "second commit"
git reset --hard HEAD~1
git log
git status
#les changements ne sont pas conservés
cat foo
```

Rappatrier du travail : Merge, Rebase et cherry-pick

Deux mécanismes pour rappatrier du travail d'une branche vers une autre.

Exercice 2 : Rappatriement avec `merge`

1. Créer un fichier `index.html` et commit le changement sur la branche `main`.
2. **Créer** une branche de développement appelée `dev`.
3. **Créer** les fichiers `index.html`, `index.js` et `style.css`.
4. On souhaiterait merger la branche `dev` sur la branche `main`, mais on souhaiterait créer un commit de merge. Or dans ce cas, le `merge` peut s'en passer (voir *fast-forward*). À l'aide de `git merge -h`, **trouver** l'option qui permet de le faire.
5. **Merger** localement cette branche dans `main` en forçant la création d'un commit de merge avec l'option trouvée précédemment.
6. **Lister** les commits et identifier le commit de `merge`.
7. **Lister** les branches locales et la branche courante.

Exercice 3 : Merge vs Rebase

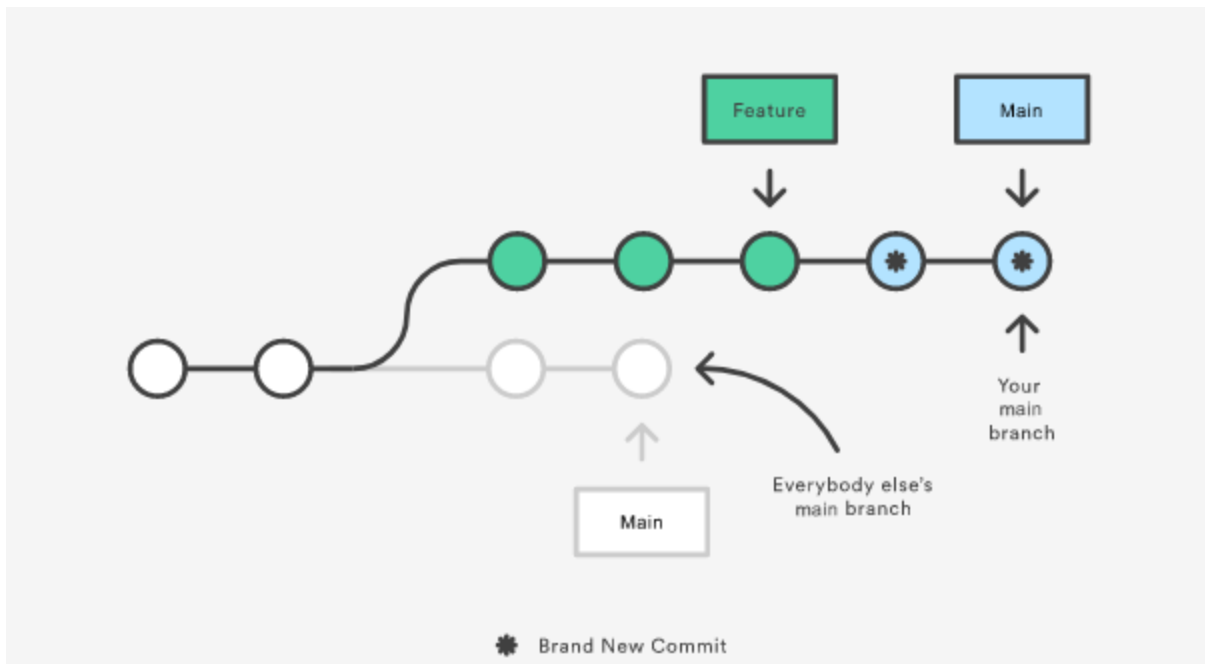
1. **Lire** la documentation de merge: `man git merge`
 2. **Lire** la documentation de rebase: `man git rebase`
 3. **Quelle** est la différence fondamentale entre les deux ?
 4. **Refaire** le rapatriement du travail mergé précédemment avec `rebase` . Pour cela, **supprimer** le dernier commit de merge avec `reset` pour réinitialiser le dépôt avant merge.
 5. **Observer** la différence entre merge et rebase sur l'historique des commits (arbre)
- | Observer la différence entre merge et rebase sur l'historique des commits (arbre)

Rebase

- Permet de rappatrier des changements sur une branche en réappliquant les commits (*nouveaux commits*)
- Peut être utilisé de manière "manuel" ou "interactive" (voir la suite)
- Maintient un historique plus linéaire (réécrit l'histoire)
- Utile pour bien nettoyer son travail sur une branche locale avant de la publier (local cleanup)

Rebase : règles d'utilisation

- Ne pas rebase de branche *publique* !
- Rebase que vos branches locales



Une branche publique est une branche sur laquelle travaille plusieurs personnes (par exemple la branche main)

Cherry pick : réappliquer des commits sur une autre branche

`cherry-pick` est une commande (avec `merge` et `rebase`) pour rapatrier du travail d'une branche à une autre.

Attention, `cherry-pick` applique les commit choisis à la branche courante (les commits sont donc **dupliqués**).

```
a - b - c - d  Main
      \
      e - f - g Feature
```

```
a - b - c - d - f  Main
      \
      e - f - g Feature
```

Par exemple, `git cherry-pick f`

`cherry-pick` est une commande puissante, elle permet de choisir individuellement chaque commit à réappliquer d'une branche à l'autre.

Utile dans certains cas, à ne pas confondre avec `rebase` . En apprendre plus [ici](#).

Rebase interactif

`rebase` peut également être exécutée *de manière interactive*.

```
# une utilisation particulière de git rebase, voir la doc pour plus d'options
git rebase -i <commit de départ exclus/branche> <commit de fin/branche> --onto <branche où appliquer la selection>
# réorganiser les 9 derniers commits sur une branche
git rebase -i HEAD~9
```

Pour en apprendre plus sur la réécriture de l'histoire, et sur git rebase interactif, [lisez cette page de la documentation officielle \(fr\)](#)

Rebase interactif

Une fois la sélection de commits faite, et que vous avez décidé où les appliquer, git vous ouvre un éditeur. Vous pouvez:

- *pick*: garder le commit ou supprimer la ligne pour l'écarter
- *reword*: reformuler le commentaire
- *edit*: modifier le commit
- *squash*: fusionner le commit avec le commit précédent

Enregistrer le travail à faire, puis fermer l'éditeur pour appliquer le rebase.

Par défaut, `git rebase -i` ouvre l'éditeur `nano`. Vous pouvez changer l'éditeur en modifiant la variable d'environnement `GIT_EDITOR`.

Exercice 4 : cherry-pick et rebase interactif

1. Sur la branche `dev`, **modifier** les fichiers suivants:
 - i. **Ajouter** `<!DOCTYPE HTML><html><head><link rel="stylesheet" href="style.css"></head><body><h1>Git workflows</h1></body></html>` dans `index.html` (commit *ajoute le contenu au document*)
 - ii. **Ajouter** `console.log('hello git')` dans `index.js` (commit *ajoute un code test*)
 - iii. **Ajouter** `html{color:red}` dans `style.css` (commit *met le document en rouge*)
2. **Créer** un commit **pour chaque changement** (3 commits)
3. Pour notre site web, on souhaiterait récupérer le nouveau code html et css, **mais pas le js** pour le moment, sur la branche `main`. Utiliser `cherry-pick` pour le faire **en une commande**.

Exercice 4 (suite)

4. Sur la branche `dev`, **corriger** le document html pour intégrer le script `index.js`. (commit *integre le js au document*)
5. Récupérer ce travail sur la branche `main` en utilisant `git rebase -i` (git rebase interactif).
6. Toujours en utilisant le git rebase interactif, **réorganiser** les commits dans l'ordre suivant:
 - i. *ajoute le contenu au document*
 - ii. *integre le js au document*
 - iii. *met le document en rouge*
 - iv. *ajoute un code test*
7. Toujours à l'aide de rebase interactif, **fusionner** les deux commits *ajoute le contenu au document* et *integre le js au document* avec comme nouveau message *intialialise le document html*
8. Toujours à l'aide de rebase interactif, **réécrire** le commentaire du commit *met le document en rouge* en *met le document html en rouge*.

Workflow avec un dépôt distant

Utiliser git en mode *centralisé* avec un dépôt distant

L'équipe joue le rôle du réseau de confiance (*network of trust*).

Fonctions du dépôt distant:

- Joue le rôle de lead du projet (*official maintainer*) dans la dynamique open source
- Backup du dépôt autre que sur sa machine
- Partager le travail avec d'autres développeur·euses
- Revue de code avec les Merge/Pull request
- Gestionnaire de ticket à la plateforme (gestion des issues), discussion maintenue au plus près du code
- Forker le projet
- Wiki intégré
- Documentation (README.md)
- etc.

Mise en place d'un dépôt local et dépôt distant (sur [Github](#) par ex)

- Se créer un compte sur [Github](#)
- Créer un dépôt `demo` *vide* (sans `README.md`, ni licence)
- Faire pointer votre dépôt local sur le dépôt distant (`git remote add origin`)
- Pousser votre travail local sur le dépôt distant (`git push`)

```
echo "# demo" >> README.md
git init
git add README.md
git commit -m "mon premier commit"
git branch -M main
git remote add origin https://github.com/paul-schuhm/demo.git
git push -u origin main
```

Inspecter le dépôt distant (url, protocole, utilisateur) avec `git remote -v`

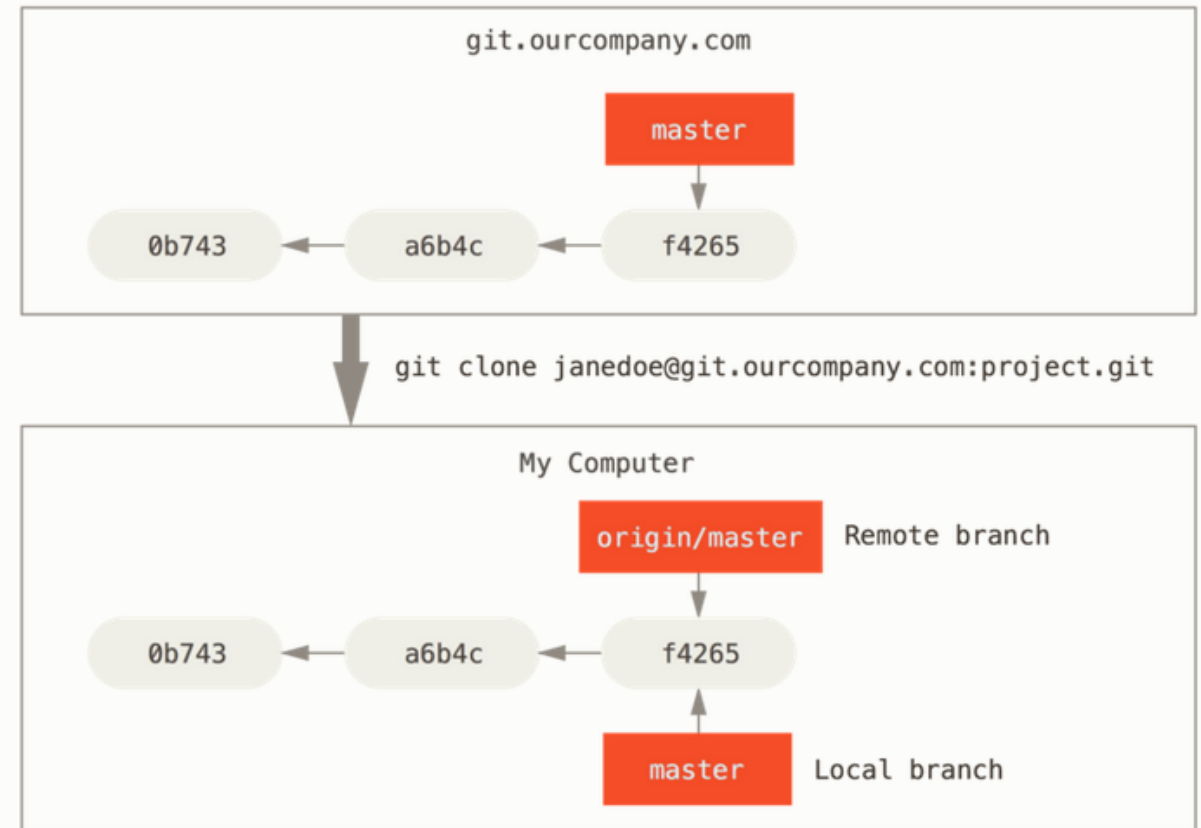
Il existe d'autres solutions que Github : héberger un *scm* sur son serveur ([gitlab](#), gitea, sourcehut, etc.), ou aller sur une autre plateforme (bitbucket, codeberg, sourcehut, [framagit](#), etc.)

Différents types de branches

Il existe 3 types de branches:

- les branches créées localement
- les branches distantes (`distant/branch`)
- les branches qui sont des copies locales des branches distantes

Commande pour savoir quelle branche distante (remote) est traquée par une branche locale (copie locale) `git branch -vv`



Exercice 5: dépôt distant

1. **Créer** un compte sur [Github](#) (si vous n'en avez pas déjà un)
2. **Créer** un *dépôt* distant pour héberger le contenu de ce cours
3. **Utiliser** votre précédent dépôt (Exercice 1) pour le faire pointer sur le dépôt distant avec `git remote add origin <url du dépôt distant>`
4. **Pousser** vos précédents *commit* sur le *dépôt* distant, avec `git push --set-upstream origin main`. *Vous allez devoir vous identifier, soit par mot de passe/token (HTTPS), soit par clé SSH (SSH)*
5. **Inspecter** les branches *locales*
6. **Inspecter** les branches *sur le dépôt distant*

Pull Request (ou Merge request)

Pull Request: *appel* à récupérer votre travail (*pull from you*)

But: indiquer aux autres utilisateur·ices que les modifications envoyées à une branche sur un dépôt GitHub.

Une fois la Pull Request (PR) émise, vous pouvez discuter, examiner les modifications avec les autres collaborateur·ices et ajouter des validations de suivi avant que vos modifications soient fusionnées dans la branche de base.

Pour faire une PR:

1. **Forker** le projet
2. **Créer** une branche, travailler dessus
3. **Publier** la branche sur son fork
4. **Créer** la pull-request

Ce processus est à la base de la contribution aux projets open source sur GitHub

Gestion des conflits

Lorsque l'on travaille à plusieurs, **si la communication n'est pas bonne** ou pour d'autres raisons, il se peut que plusieurs personnes modifient les mêmes fichiers en même temps sur la même branche, ou modifient l'historique public (sur le dépôt distant).

Dans ce cas, des conflits émergent, Git ne peut pas décider quoi faire *sans votre intervention*.

Résoudre les conflits de merge

Deux cas:

- git ne parvient pas à **lancer** le merge (problème local à votre dépôt)
- git rencontre un problème **pendant** le merge (problème avec les branches distantes)

Démo : créer un conflit local à résoudre

1. **Créer** un répertoire `git-merge-test` , **initialiser** un dépôt.
2. **Créer** un fichier `merge.txt` avec du contenu.
3. **Ajouter** `merge.txt` au dépôt et **commiter-le**.
4. **Créer** une branche `new_branch`
5. **Remplacer** le contenu du fichier `merge.txt`
6. **Commit**.
7. **Repasser** sur main et **modifier** le fichier `merge.txt` sur la même ligne.
8. Merger la branche `new_branch` avec `main` . Vous devriez obtenir un conflit.
9. **Inspecter** le fichier à l'origine du conflit. **Résoudre** le conflit en éditant le fichier.
10. **Commit** le changement pour **finaliser** le merge.
11. **Inspecter** avec `git log --graph` pour voir le commit de merge.

| Lire [cet article](#) pour en apprendre plus sur la résolution de conflits en merge.

Exercice 4 (1h)

Consolider les concepts et les commandes de base associées.

Rendez-vous à [cette adresse](#) et réaliser *tous* les exercices (local et remote).

Prenez votre temps et passez plus de temps sur ce que vous avez du mal à maîtriser ou à comprendre.

Annexe - quelques définitions

Pour que tout le monde parle la même langue

- **workspace** (espace de travail): le dossier sur votre machine dans lequel un répertoire a été initialisé.
- **repository** (dépôt): "abstrait", géré par git et le dossier caché `.git` dans votre workspace.
- **working tree**: l'état des fichiers *vu depuis un commit donné* (checkout).
- **staging area ou index**: un gros fichier binaire dans le dépôt `.git` (`.git/index`). Liste tous les fichiers *de la branche courante*, leurs SHA1, timestamps et leurs noms (**ce n'est pas une copie**). Grâce à l'index, git détecte les changements dans le *working tree* (modification, ajout, suppression).
- **HEAD**: une référence particulière qui pointe sur le commit courant (checkout). Indique *là où on est*. Souvent représenté par un astérisque sur les schémas (et dans ce cours).
- **tag**: une référence fixe, attachée à un commit.
- **branch**: une référence spéciale. Elle bouge pour refléter la position courante du projet en développement, c'est à dire sur le dernier commit (*tip*).

Références

- [gittutorial](#), le tutoriel officiel de git, pour revoir les bases.
- [Un cours d'introduction à git](#), de [Fabien Rozar](#) (*trust network*). Présentation sur laquelle je me suis beaucoup inspiré pour ce module d'introduction/rappel à git. Présente l'essentiel à connaître.
- [Git going with DVCS](#), une présentation de [Matthew McCullough](#) sur les fondamentaux de git. Il a créé beaucoup de présentations sur le sujet (à fouiller). Utilisé plusieurs schémas de celle-ci.
- [Git: Cheat Sheet \(advanced\)](#), une antisèche des commandes git (avancé), publiée par Maxence Poutord
- [Git from the bottom up](#), un livre open source sur git et ses rouages internes, très complet, par John Wiegley
- [learngitbranching](#), apprendre les concepts et fonctions essentielles de git via une appli web interactive