Git, *essentials*

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
```

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.mail "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.mail "mail user"
```

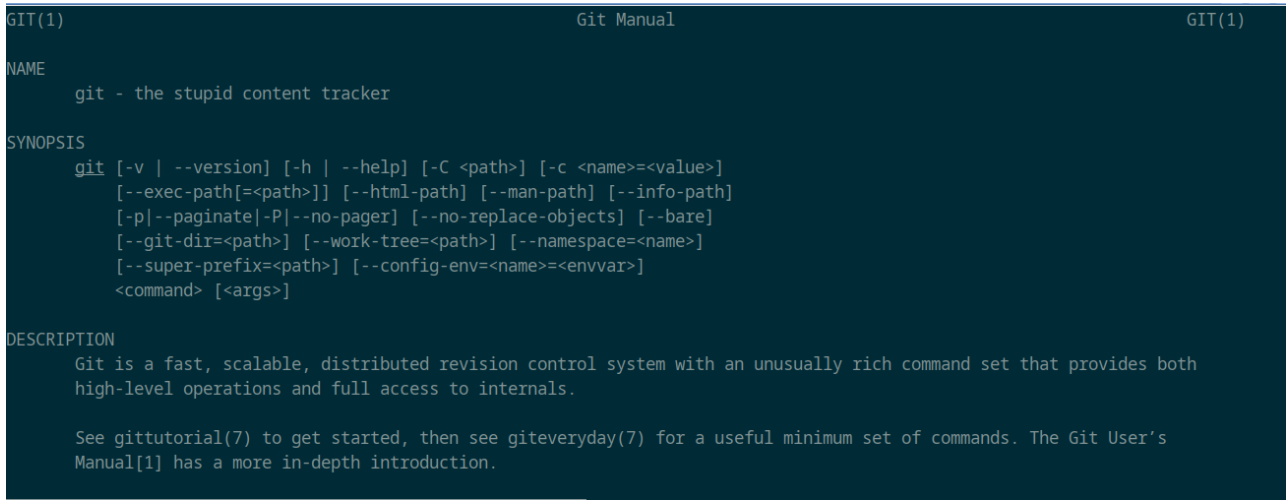
> L'user sert juste à vous identifier sur l'historique (arbre des commits), pas à l'authentification sur GitHub ou autre !

(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: `/search`
- occurrence suivante: `n`
- occurrence précédente: `N`

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



Les commandes de base (à connaître)

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
```

Les commandes de base (à connaître)

Local (sur votre machine)

Commandes pour la gestion des branches

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

Commandes de configuration

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

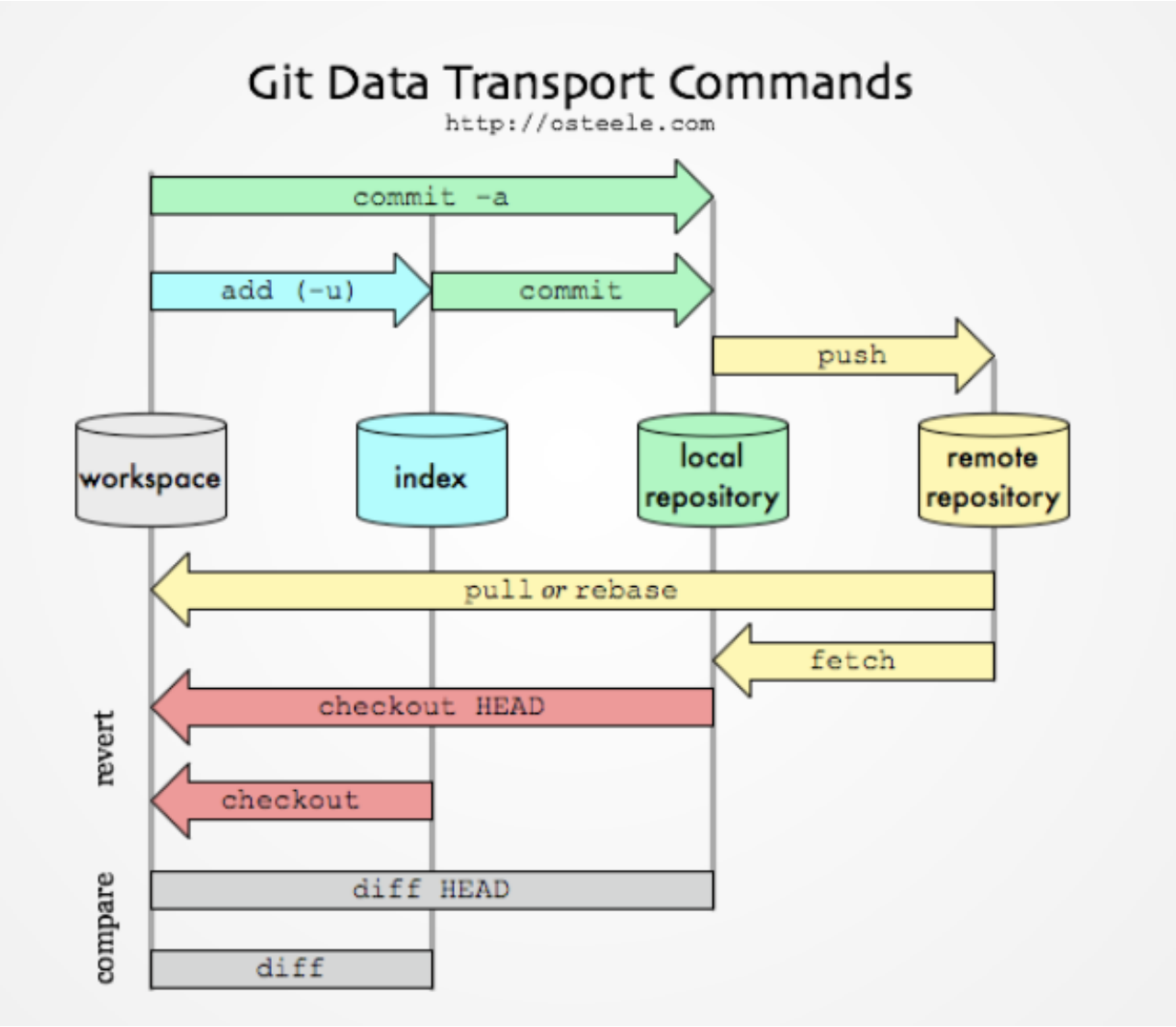
Les commandes de base (à connaître)

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

Quelques commandes en image



Workflow en local

(Rappels sur) Les commit

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

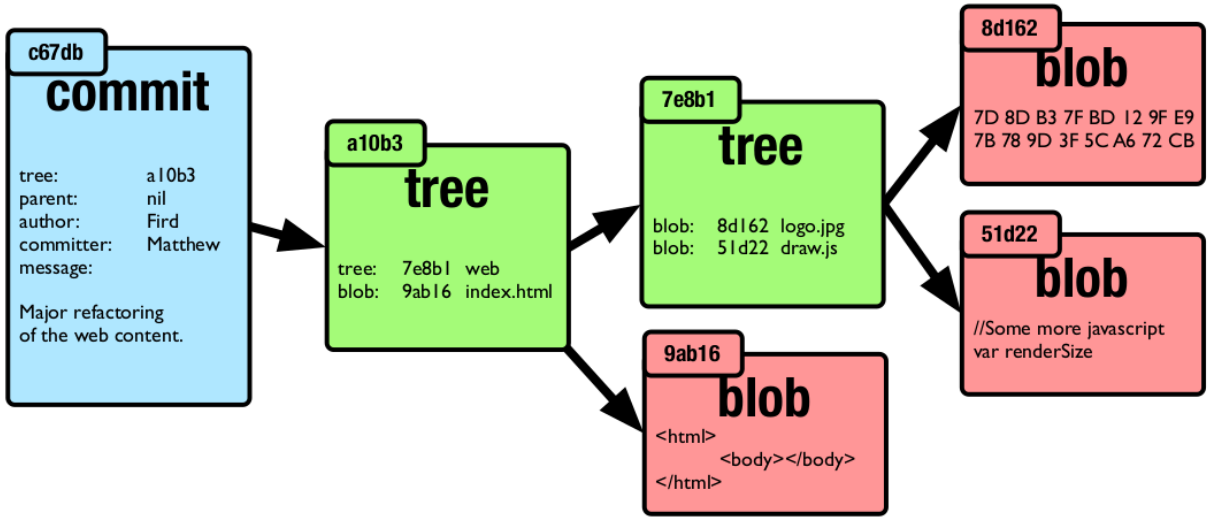
Un *commit*:

- est une **photographie du dépôt dans un état donné**
- stocke la **différence** entre deux commits
- peut avoir 0, 1 ou 2 commits parents
- est crée sur un ordinateur, *par quelqu'un*, **localement**
- est identifié par un identifiant unique: un hash SHA1. Par exemple `9AB223D28B1AA46EF1780B22F304982E39872C34`

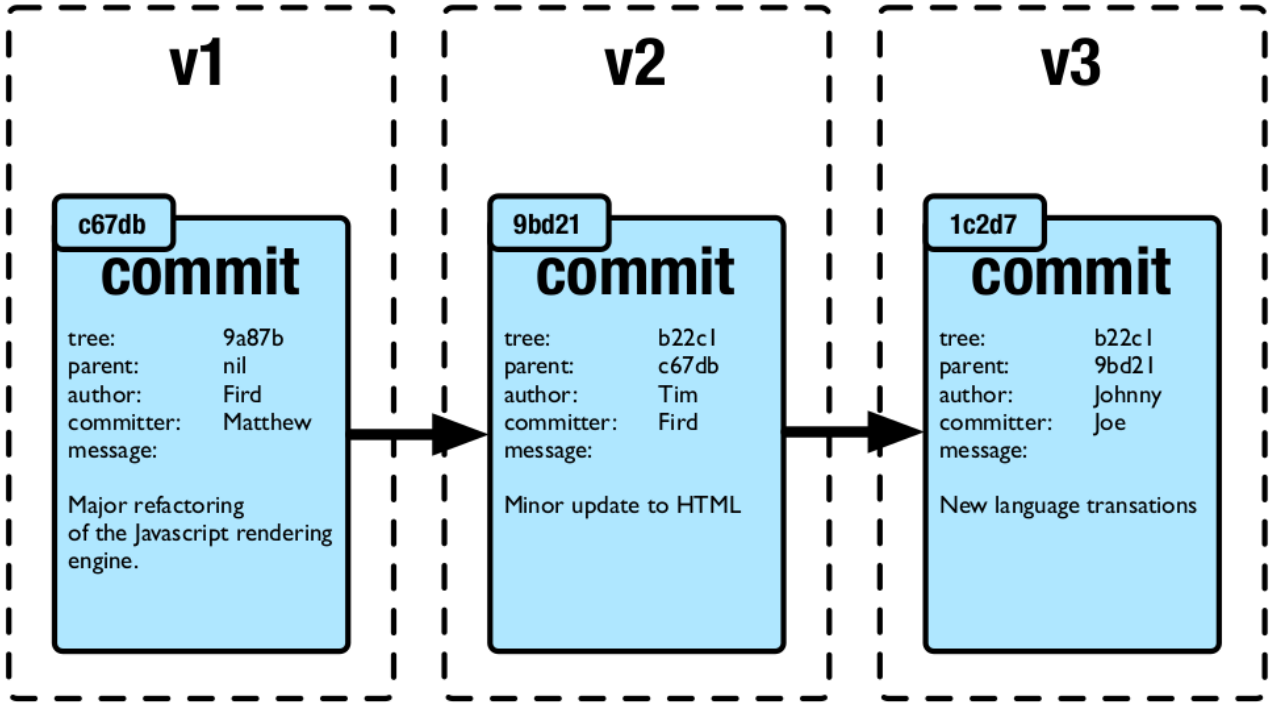
Objets manipulés par git: des fichiers au dépôt

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

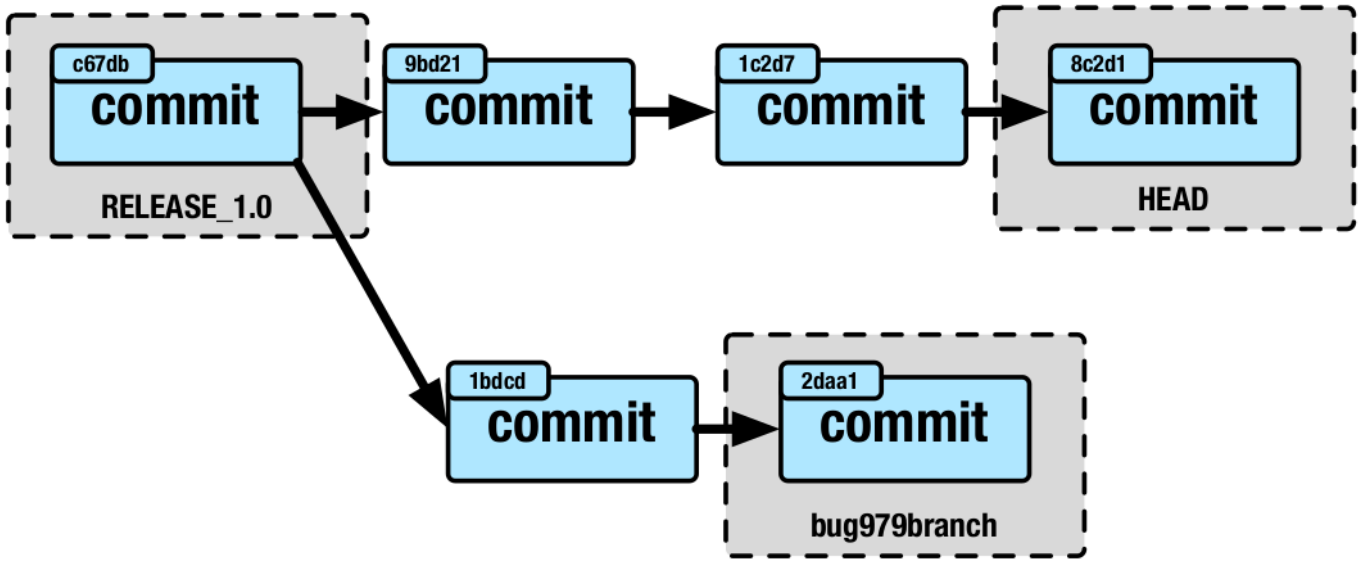
> Pour en savoir plus sur les rouages internes de git, consulter cet excellent ouvrage en ligne *Git from the bottom up*, de John Wiegley.
> Nous nous arrêterons à cette couche d'abstraction pour travailler.



Historique: l'arbre des commits



Les tags



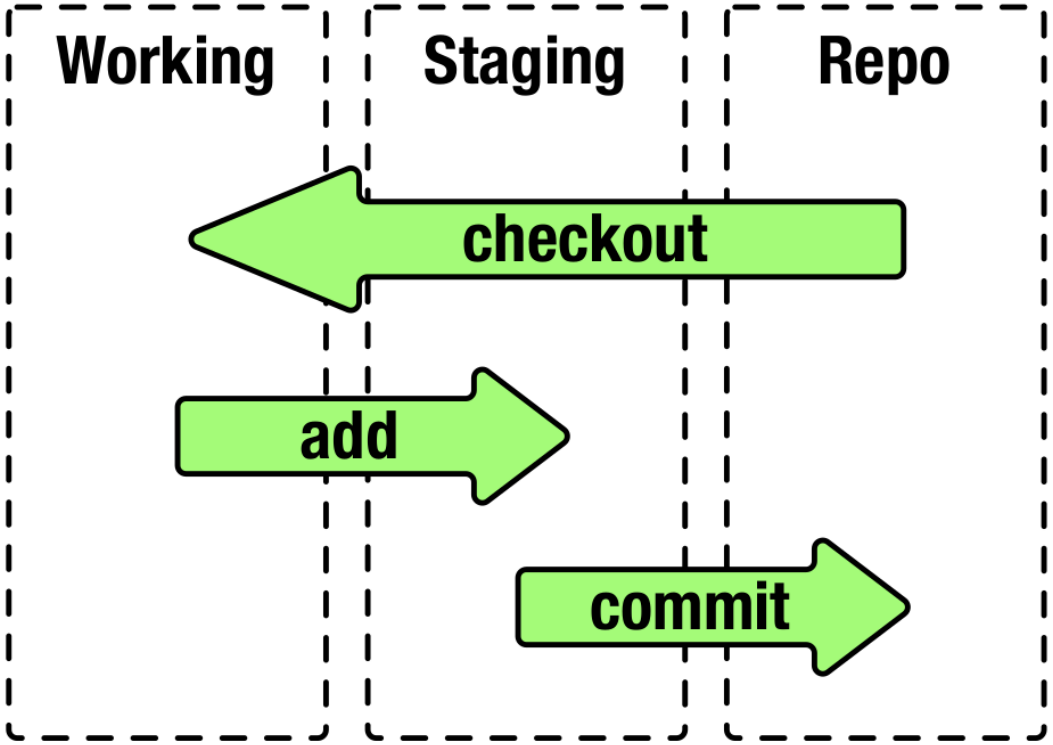
Lister les tags, ajouter un tag `v0.1` sur `HEAD` avec le message "app v 0.1"

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

> Les branches ne sont qu'un *cas particulier* de tags. Un tag est une référence (une étiquette) fixe, attachée à un commit.

Un travail en 3 temps

- on récupère un état du dépôt (`git checkout`)
- on édite les fichiers
- on les *stage*, ajoute dans l'index (`git add`)
- on commit les changements (`git commit`)

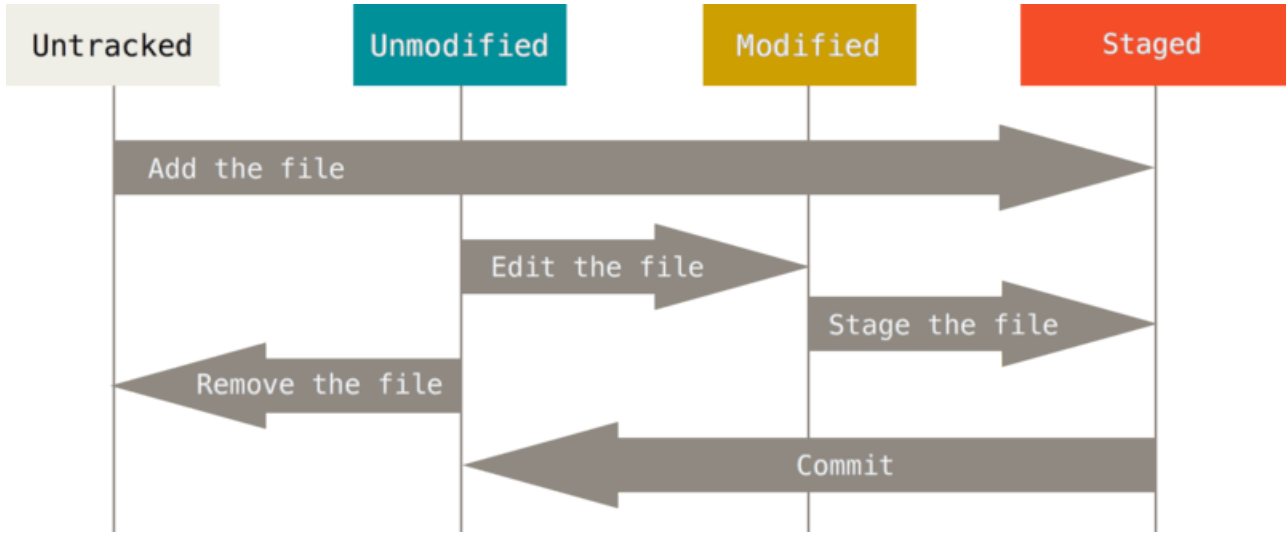


Status des fichiers

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

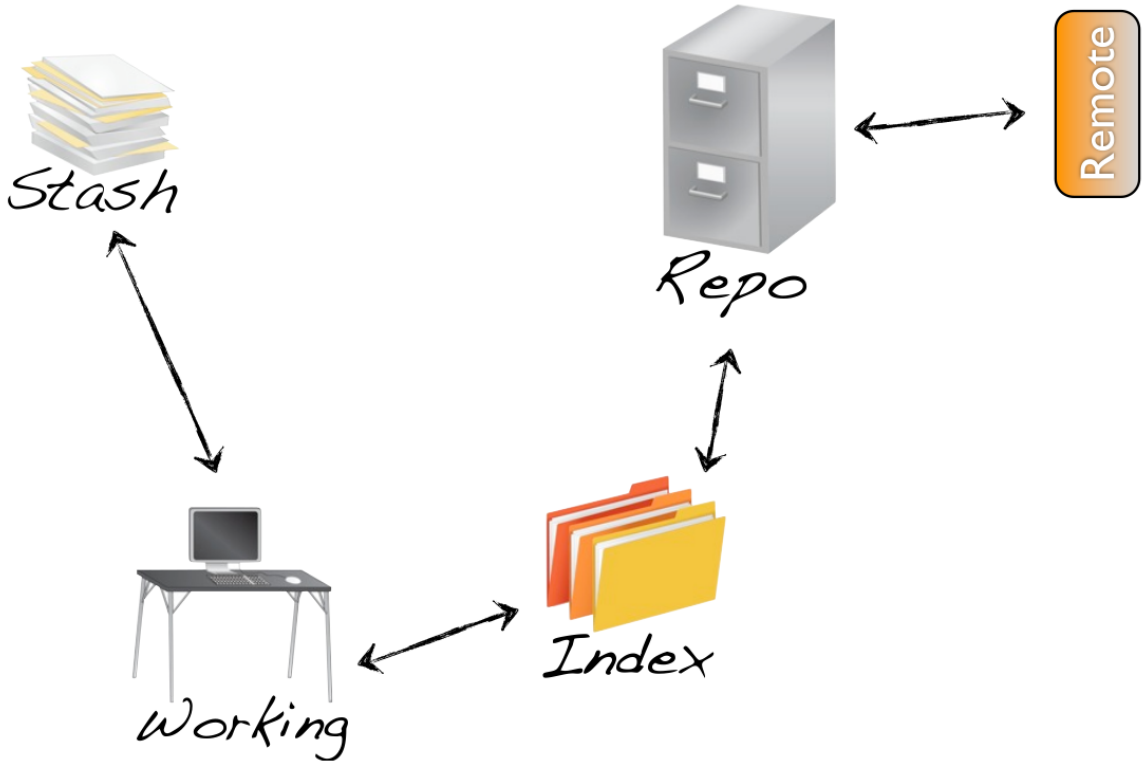
L'état *staged* correspond à la salle d'attente du commit. C'est dans l'état *staged* qu'on accumule les changements à apporter au dépôt pour un *commit*.

> staged = dans l'index



Analogie

- **`git clone`** (seule partie qui demande une connexion)
- `git checkout`
- travail (edition des fichiers)
- interruption, changer de tâche: `git stash`
- `git stash apply`
- `git add`
- `git commit`
- **`git push`** (seule partie qui demande une connexion)



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`

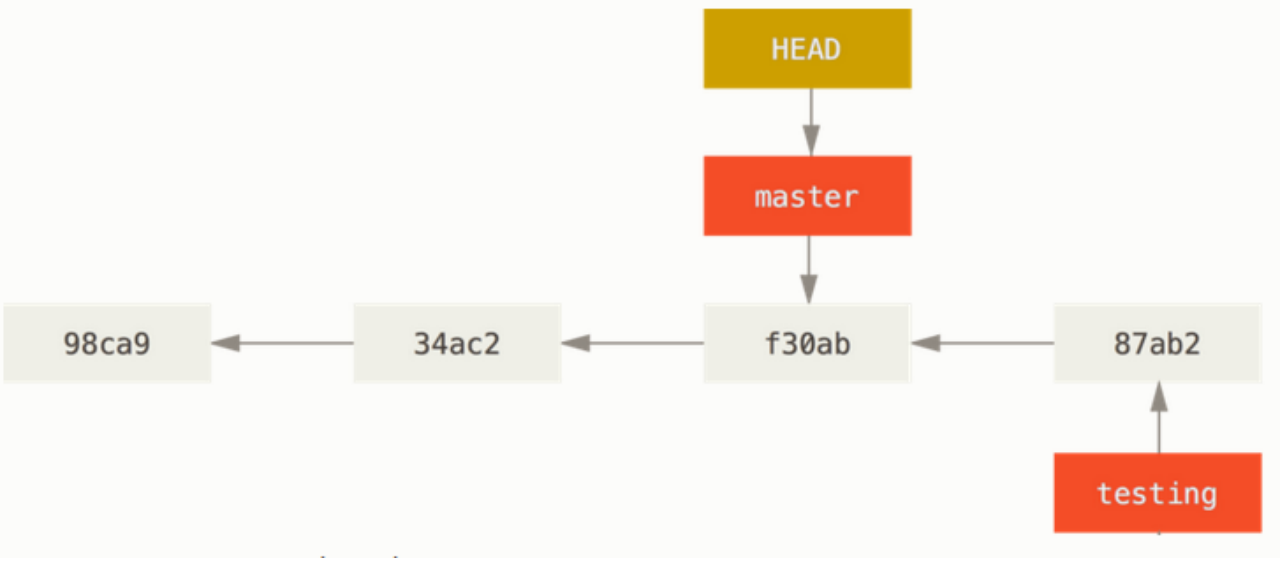
Les branches, le cœur de Git

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

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

Git permet de:

- créer des branches
- de les **fusionner** (2 méthodes: via `merge` ou `rebase`)
- de les supprimer



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
```

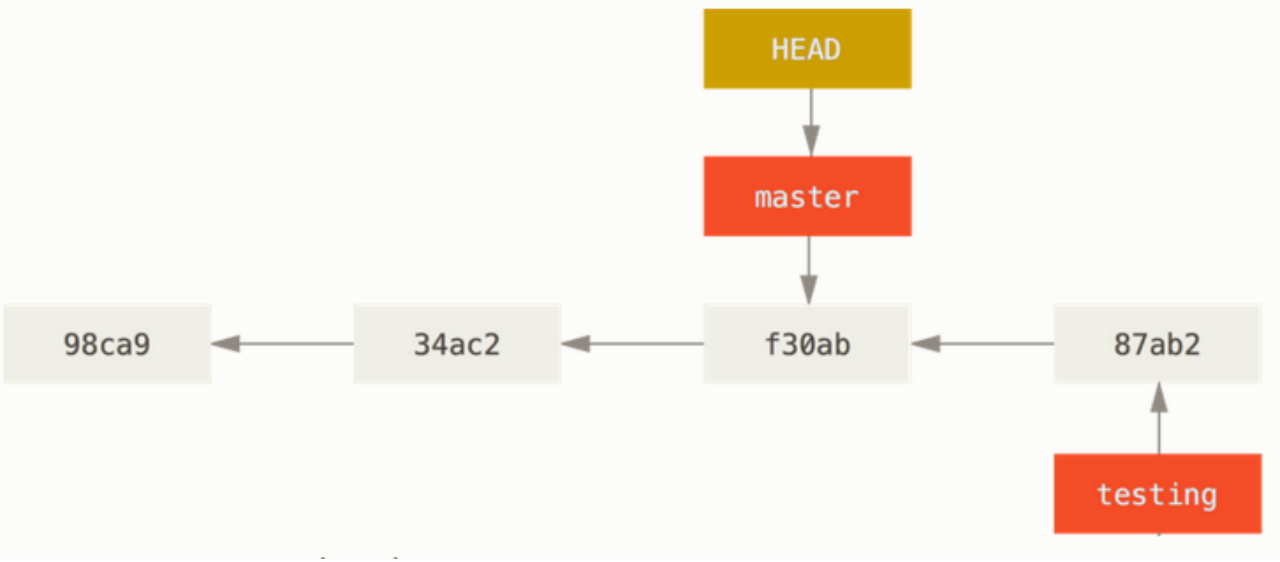
Le pointeur `HEAD`*

`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 nimporte quel commit)

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



Gérer le stash

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

Utilisez `git stash` lorsque vous voulez enregistrer l’état actuel du répertoire de travail et de l’index, mais que vous voulez revenir à un répertoire de travail propre.

La commande enregistre vos modifications locales et rétablit le répertoire de travail pour qu’il corresponde au commit `HEAD`.

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

Annuler le dernier commit 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>
```

Exercice 2 : manipulation des branches avec `merge`

1. **Créer** une branche de développement appelée `dev`.
2. **Créer** les fichiers `index.html`, `index.js` et `style.css`.
3. 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.
4. **Merger** localement cette branche dans `main` en forçant la création d'un commit de merge avec l'option trouvée précedemment.
5. **Lister** les commits et identifier le commit de `merge`.
6. **Lister** les branches locales et la branche courante.

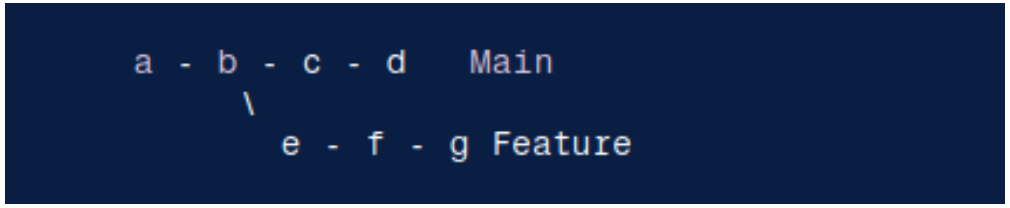
Exercice 3 : Merge vs Rebase

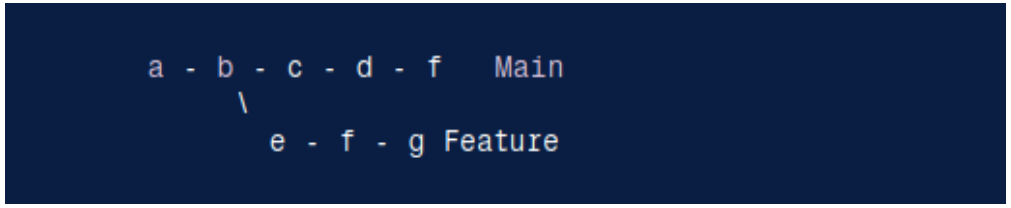
1. **Lire** la documentation de merge: `man git merge`
2. **Lire** la documentation de rebase: `man git rebase`
3. **Quelle** est la différence fondamentale entre les deux ?
4. **Refaire** le rapatriement du travail mergé précédemment avec `rebase`. Pour cela, **supprimer** le dernier commit de merge avec `reset` pour réinitialiser le dépôt avant merge.

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**).





> 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 éxecutée *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
- etc.

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 *contenu de la page web*)
    ii. **Ajouter** `console.log('hello git')` dans `index.js` (commit *js implementation*)
    iii. **Ajouter** `html{color:red}` dans `style.css` (commit *style html red*)
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 *integration js*)
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. *contenu de la page web*
    ii. *integration js*
    iii. *style html red*
    iv. *js implementation*
7. Toujours à l'aide de rebase interfactif, **fusionner** les deux commits *contenu de la page web* et *integration js* avec comme nouveau message *init markup*
8. Toujours à l'aide de rebase interfactif, **réécrire** le commentaire du commit *style html red* en *balise html couleur 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`

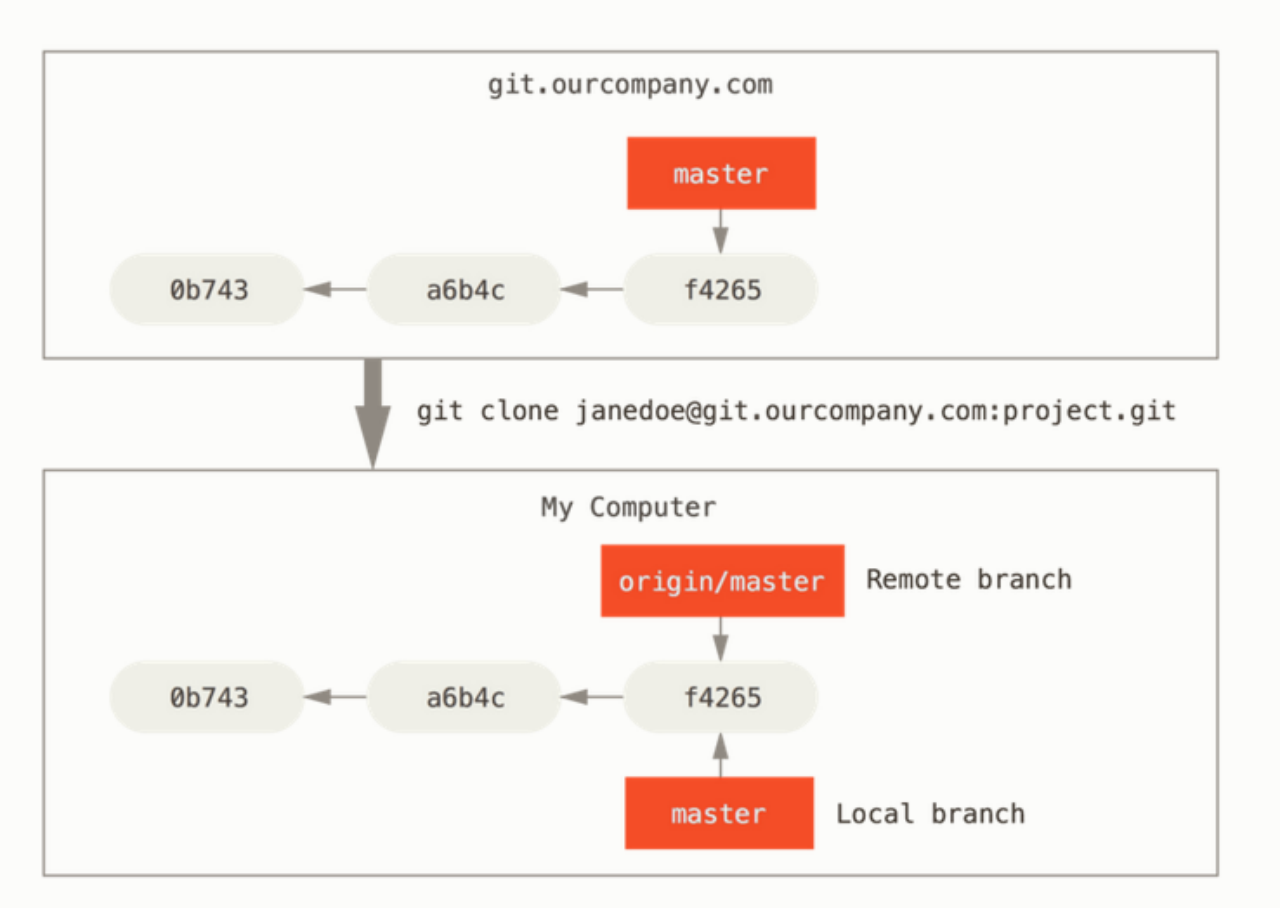
> Il existe d'autres solutions que Github : héberger un *scm* sur son serveur (gitlab, gitea, sourcehut, etc.), ou aller sur une autre plateforme (bitbucket, codeberg, sourcehut, framagit, etc.)

Différents types de branches

Il existe 3 types de branches:

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

> Commande pour savoir quelle branche distantielle (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): le dossier caché `.git` dans votre workspace.
- **working tree**: l'état des fichiers *vu depuis un commit donné* (checkout).
- **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 crée 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