

# Outils et méthodes de développement

Johann Chopin

Premier semestre de la 3ème années de la licence  
Informatique et ingénierie du web.



## Introduction à GIT

# Plan

---

## 1. Les différents types de systèmes existants

- 1.1. Système de contrôle de versi...
- 1.2. Système de contrôle de versi...
- 1.3. Système de contrôle de versi...
- 1.4. Brève histoire de GIT

## 2. Les bases de GIT

- 2.1. Installation
- 2.2. Configuration
- 2.3. Fonctionnement
  - 2.3.1. Différences entre Snapshot et Delt...
  - 2.3.2. Les commits
  - 2.3.3. Détection des fichiers
  - 2.3.4. Workflow typique
  - 2.3.5. CLI version vs GUI version

## 3. Local GIT repository

- 3.1. Créer un GIT repository
- 3.2. Enregistrement des modificati...
  - 3.2.1. Syntaxe Fileglob
  - 3.2.2. Création d'un commit
- 3.3. Annulation des modifications

## 3.4. Comparaison des versions

- 3.4.1. Renommer un fichier
  - 3.4.2. Supprimer un fichier
- ## 3.5. Historique des commits
- ## 3.6. Réécrire l'historique - 3.6.1. Modification du dernier commit - 3.6.2. Modifier des commits plus anciens...

## 4. GIT branching

- 4.1. Introduction
- 4.2. Créer une nouvelle branche
- 4.3. Switcher sur une branche
- 4.4. git log --graph
- 4.5. Supprimer une branche locale
- 4.6. Renommer une branche locale
- 4.7. Exercice
- 4.8. Merger des branches
  - 4.8.1. Fast forward merge
  - 4.8.2. Exercice
  - 4.8.3. Résolution des conflits lors des me...

## 5. Remote GIT repository

- 5.1. Introduction
- 5.2. Authentification SSH

## 5.3. Ajout d'un remote repository

- 5.4. Cloner un remote repository
- 5.5. Lister vos remotes repositories
- 5.6. Fetching: télécharger du rem...
- 5.7. git pull
- 5.8. Pushing: Uploader du local ve...

## 6. Tags

- 6.1. SemVer

## 7. TP: Utilisation d'une GUI

## 8. Git workflow

- 8.1. GitFlow
- 8.2. Trunk-based development
- 8.3. GitFlow vs Trunk-based devel...

## 9. Conventional Commits

- 9.1. Syntaxe
- 9.2. Pourquoi l'utiliser?
- 9.3. Exemple de CHANGELOG
- 9.4. Exercice

## 10. Git Cheatsheet

## 11. Ressources

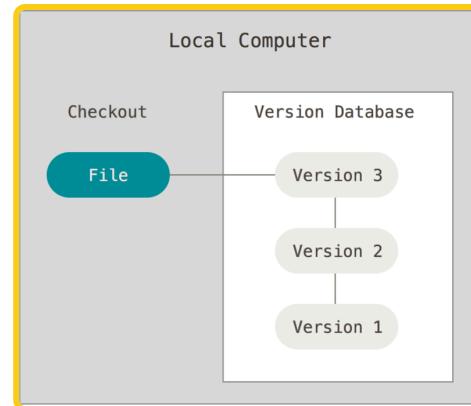
# Les différents types de systèmes existants

*Le contrôle de version est un système qui enregistre tous les changements faits sur un fichier ou un ensemble de fichiers à travers le temps. Le but est de pouvoir comparer différentes versions de code entre elles, ou/et, de pouvoir revenir en arrière en cas d'incidents.*

# Système de contrôle de versions local

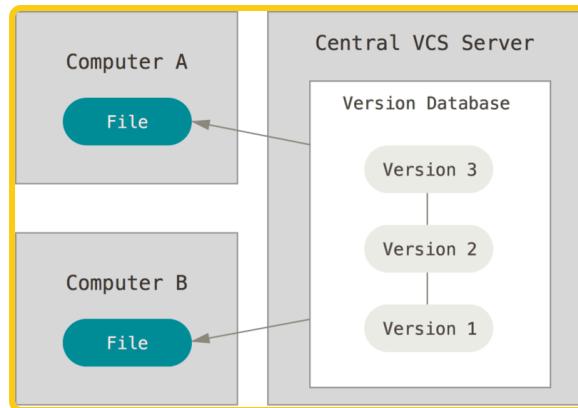
- Utilisation en local uniquement par un seul développeur
- Chaque version d'un fichier est sauvegardé dans une base de données.

RCS (Revision Control System) est un exemple de VCS (Version Control System).



# Système de contrôle de versions centralisé

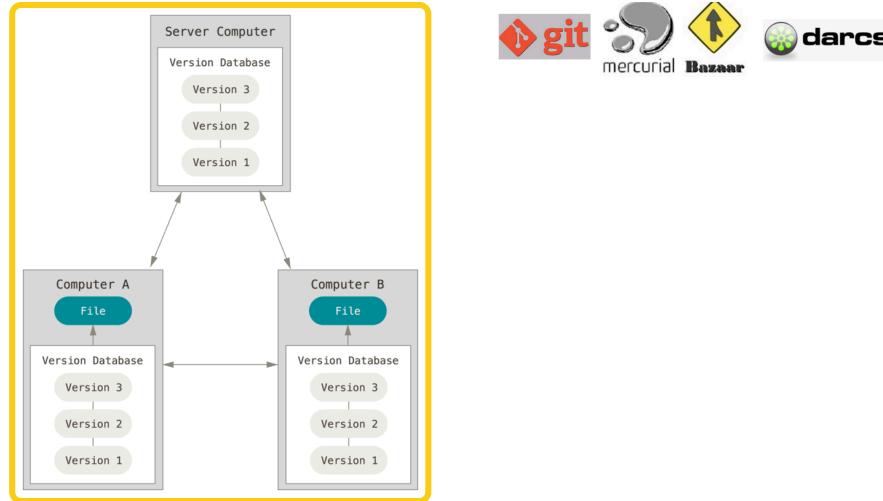
- Utilisation collaborative entre plusieurs développeurs
- 1 seul server contient toutes les versions des fichiers versionnés
- Plusieurs clients peuvent faire un check-out d'une version du code et travailler dessus en même temps



- Tortoise
- SVN
- Perforce

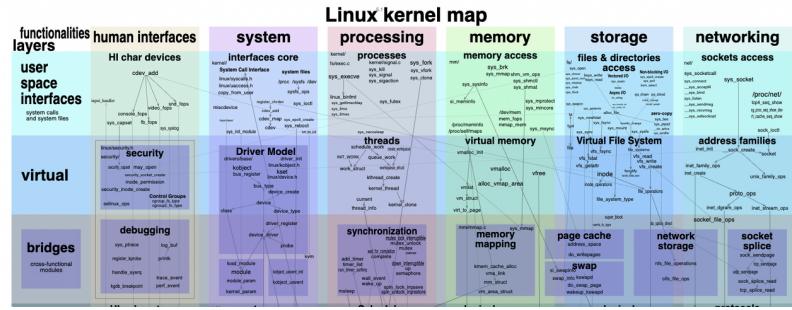
# Système de contrôle de versions distribué

- Utilisation collaborative entre plusieurs développeurs
- Les clients ne font pas un simple check-out du dernier snapshot (version) des fichiers, mais un mirroring (une copie) de tout le **repository**, avec tout son historique.
- Ainsi si le serveur meurt alors que ces systèmes collaboraient avec lui, n'importe lequel des client repositories peut être copié sur le nouveau serveur et ainsi le rétablir dans sa totalité.
- Chaque clone est vraiment un **full backup** de toutes les données.



## Brève histoire de GIT

- Le noyau de Linux est un projet open-source avec un scope très large. De 1991 à 2002, le code n'était versionné cependant qu'à l'aide de patchs et de fichiers archivés



- En 2002, le projet du noyau de Linux commence à utiliser un DVCS appelé BitKeeper.
  - En 2005, BitKeeper était devenu payant arrêtant brutalement les relations avec la communauté qui développait le noyau de Linux.

## Les différents types de systèmes existants



- Cette situation poussa la communauté qui développait Linux et en particulier son créateur, **Linus Torvalds**, à développer leur propre outil basé sur les leçons apprises pendant qu'ils utilisaient BitKeeper.
- En avril 2005, **GIT** est né

## **GIT est un outil devant être:**

- rapide au design simple
- fort à gérer de nombreuses branches parallèles
- complètement distribué
- capable de gérer de grands projets comme le noyau de Linux
  
- En 2016, GIT devient le logiciel de gestion de version le plus populaire en étant utilisé par plus de 12 millions de personnes !

## Quelle est la signification de Git ?

- La première raison pourrait être considérée comme pratique. Une commande de trois lettres pratique est rapide et facile à dire et à taper.
- Le mot `git` n'est utilisé par aucune autre commande Unix existante.
- La première version du code de Git était très simple, à tel point qu'il la trouve digne d'être insultée.
- Un acronyme pour **Global Information Tracker**, lorsqu'il fonctionne correctement.
- Un acronyme pour **Goddamn Idiotic Truckload of sh\*t**, lorsqu'il ne fonctionne pas correctement.

## Les différents types de systèmes existants

Linus Torvalds fait son premier commit du code de Git le 7 avril 2005.

Dans ce commit, il a inclus un fichier appelé README :

```
GIT - the stupid content tracker
```

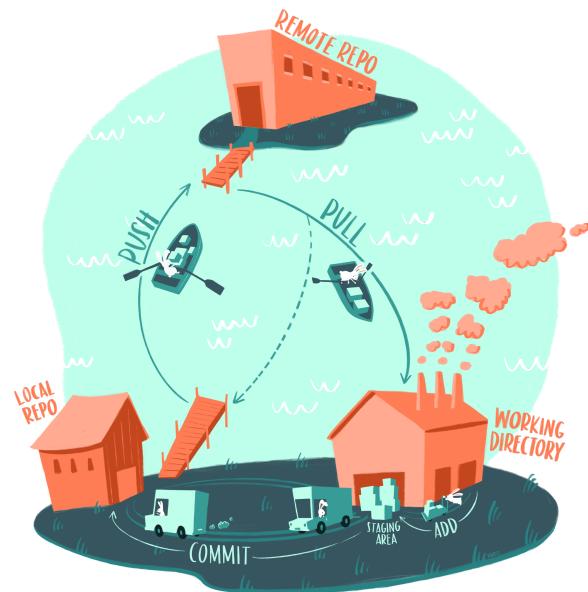
```
"git" can mean anything, depending on your mood.
```

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "goddamn idiotic truckload of sh\*t": when it breaks

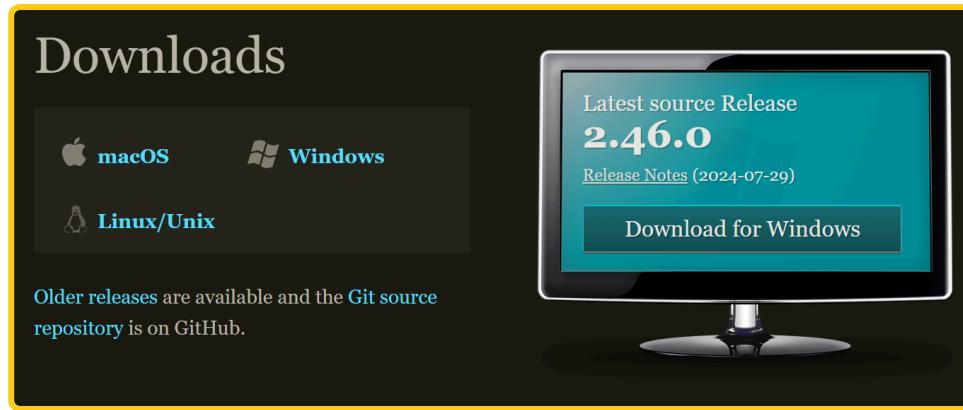
```
This is a stupid (but extremely fast) directory content manager. It doesn't do a whole lot, but what it _does_ do is track directory contents efficiently.
```

# Les bases de GIT

---



# Installation

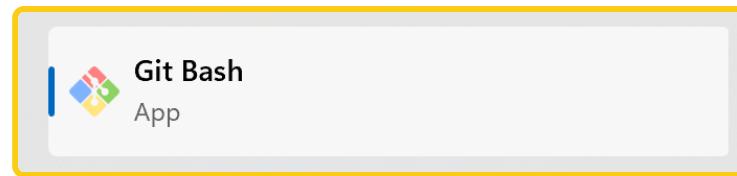


🔗 <https://git-scm.com/downloads>

```
$ git --version  
git version 2.46.0
```

## Les différents types de systèmes existants

- Sous Windows, ouvrez l'application Git Bash. Il s'agit d'une console MinTTY.
- MinTTY reconnaît les principales commandes Unix/Linux :
  - `cd` - Changer de dossier
  - `ls` - Liste des fichiers et des dossiers
  - `mv` - Déplacer/renommer des fichiers et des dossiers
  - `cp` - Copier des fichiers et des dossiers
  - `rm` - Supprimer des fichiers et des dossiers
  - `pwd` - Affiche le path du dossier
  - ...



# Configuration

---

- La première fois qu'on installe GIT, vous voudrez faire quelques setups afin de personnaliser votre installation.
- Git s'installe avec un outil appelé `git config` qui vous permet de configurer certaines variables qui contrôlent la façon dont GIT s'affiche et fonctionne. Ces variables se trouvent à 3 niveaux :
  - `/etc/gitconfig` : ce fichier contient les valeurs de chaque utilisateur sur le système et tous leurs repositories. Vous pouvez lire et écrire dans ce fichier grâce à l'option `--system`.
  - `~/.git/config` ou `~/.config/git/config` : ce fichier contient vos valeurs personnelles. Vous pouvez lire et écrire dans ce fichier grâce à l'option `--global` qui affectera tous les repositories avec lesquels vous travaillez sur votre système.
  - `config` : ce fichier se trouve dans le répertoire GIT (`.git/config`). Vous pouvez forcer GIT à lire et écrire dans ce fichier grâce à l'option `--local` qui est aussi l'option par défaut. Vous devez être présent dans le Git repository pour que cette option fonctionne correctement.
- Chaque niveau inférieur écrase les paramétrages du niveau supérieur. Par exemple, les paramétrages faits dans `.git/config` écrasent ceux dans `/etc/gitconfig`.

## Les différents types de systèmes existants

**À vous de jouer:** Paramétrez votre nom d'utilisateur et votre email:

```
$ git config --global user.name "Prénom Nom"  
$ git config --global user.email monemail@example.com
```

- En utilisant l'option `--global`, vous n'aurez besoin de le faire qu'une seule fois car GIT utilisera toujours cette information.
- Si vous voulez remplacer ces informations avec un nom différent ou un email différent pour certains projets spécifique, il vous suffit de lancer la commande sans l'option `--global` quand vous êtes dans ce projet.

## Les différents types de systèmes existants

- Commande pour afficher la configuration: `git config --list`
- Commande pour voir la valeur associée à une entrée spécifique: `git config user.name`
- Pour avoir accès à l'aide sur une option git :
  - `git help <mon_option_git>`
  - Exemple pour avoir l'aide sur config: `git help config`
  - Cette commande affichera dans votre navigateur web le détail de la documentation sur cette option Git à partir de la doc HTML fournie avec Git lors de son installation.
- Pour accéder à l'aide rapide et concise sur une option git utilisez : `git <mon_option_git> -h`

```
git add -h
usage: git add [<options>] [--] <pathspec> ...
```

```
-n, --dry-run      dry run
-v, --verbose     be verbose
```

# Fonctionnement

---

L'une des différences majeures entre Git et les autres VCS (RCS, CVS, SVN...) est sa façon de penser à ses données.



## Différences entre Snapshot et Delta storage

- **Δ Delta storage:** enregistre uniquement les modifications apportées à un fichier ou à un système plutôt que l'état complet. Chaque modification est stockée sous la forme d'un delta ou d'une différence par rapport à la version précédente, et l'état actuel du système est reconstruit en combinant tous les deltas.
-  **Snapshot storage:** enregistre l'état complet d'un fichier ou d'un système sous la forme d'une instance unique, appelée **snapshot**. Chaque instantané représente le système à un moment donné et sert de point de référence pour les modifications futures.

## Les différents types de systèmes existants



Récupération du solde du jour 6:

- **Delta:**  $\Delta 6 + \Delta 5 + \Delta 4 + \Delta 3 + \Delta 2 + \Delta 1 = 1,450$
- **Snapshot:**  $\text{camera} 6 = 1,450$

Comparaison du solde entre les jours 4 et 2:

- **Delta:**  $(\Delta 4 + \Delta 3 + \Delta 2 + \Delta 1) - (\Delta 2 + \Delta 1) = -50$
- **Snapshot:**  $\text{camera} 4 - \text{camera} 2 = -50$

Git utilise une forme de Snapshot storage.

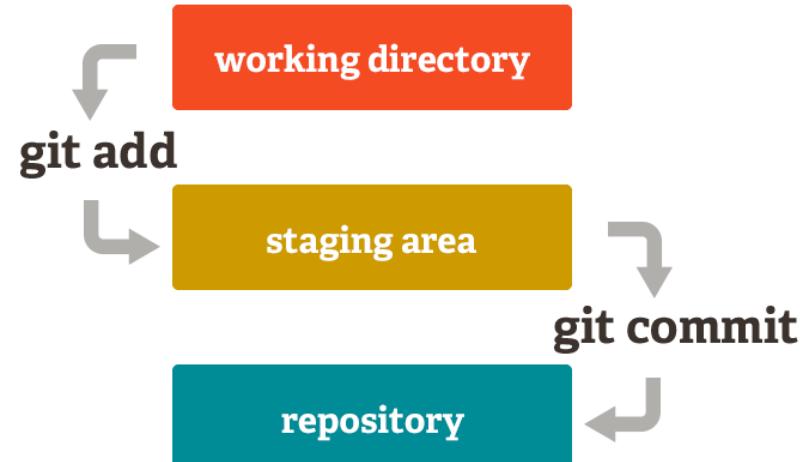
- Très efficaces pour gérer de nombreuses versions
  - GIT ne stocke pas les fichiers non modifiés mais utilise un lien
- Permet de facilement analyser 2 versions différentes
- Permet de travailler facilement en parallèle
- Gère mal les fichiers volumineux

## Les commits

Dans Git, les "snapshots" sont appelés des **commits**.

Git a trois **états fondamentaux** à connaître :

- **Modified:** le fichier a été modifié dans votre **Working directory**, mais n'est pas encore **Staged** ou **Committed**.
- **Staged:** le fichier est poussé dans la **Staging Area**, afin qu'il fasse partie de votre prochain commit.
- **Committed:** les données sont stockées dans le **.Git Repository**, c'est à dire la base de données locales.



- **Working Directory:** est un check-out d'une version du projet. Les fichiers sont extraits du .git repository et copiés dans ce répertoire pour être utilisés ou modifiés.
- **Staging Area:** est l'endroit où sont indiqués les fichiers prêts à partir dans le .git repository lors du prochain commit.
- **.git Repository:** est l'endroit où GIT stocke ses metadonnées et ses objets de base de données pour votre projet. C'est la partie la plus importante de GIT et c'est ce qui est copié quand vous clonez un répertoire d'un autre ordinateur.

La plupart des opérations fait par GIT ont besoin seulement des fichiers en local. Aucune ressource réseau n'est nécessaire contrairement aux autres systèmes de contrôle de versions centralisés où il y a toujours une latence réseau.

## Détection des fichiers

### GIT est consistant et intègre!

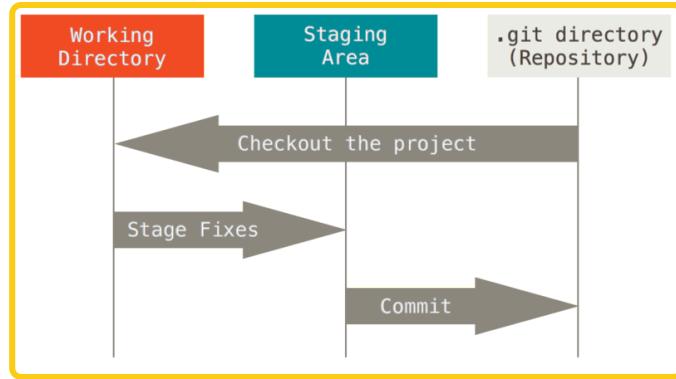
- avant de stocker un fichier ou une référence vers l'un d'eux, GIT effectue des check-sum (**SHA-1 hash**) des fichiers.
- il est impossible de modifier un fichier ou dossier sans que GIT ne le sache ce qui garantit l'intégrité du système.

**SHA-1** est une fonction de hachage cryptographique conçue par la NSA.

SHA1("Wikipédia, l'encyclopédie libre et gratuite") = 6153A6FA0E4880D9B8D0BE4720F78E895265D0A9

SHA1("Wikipédia, l'encyclopédie libre et gratuitE") = 11F453355B28E1158D4E516A2D3EDF96B3450406.

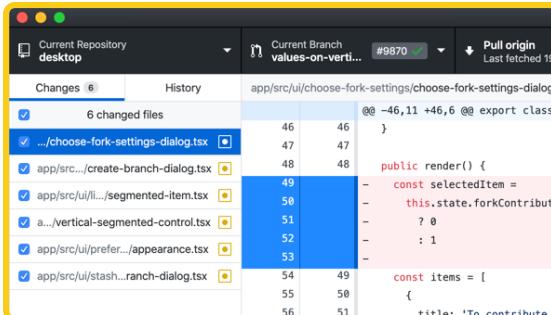
## Workflow typique



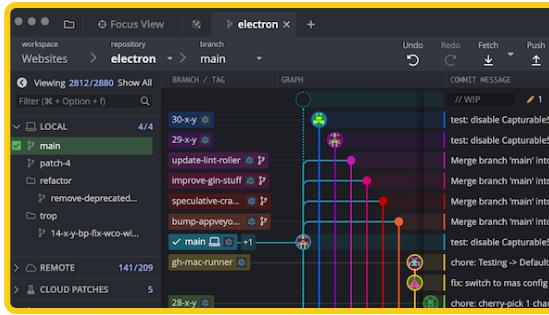
1. Vous faites un **checkout** du **.git Repository** dans votre **Working Directory**
2. Vous modifiez des fichiers dans votre **Working Directory**
3. Vous les sélectionnez pour les **stager** afin qu'ils fassent partie du prochain commit ce qui les ajoute dans la **Staging Area**
4. Vous faites un **commit**. Les fichiers présents dans la Staging Area sont **archivés comme snapshot permanent** (version permanente) dans votre **.git Repository**.

## CLI version vs GUI version

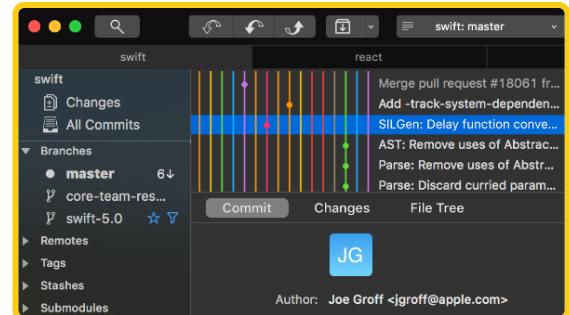
- Git est un outil CLI (Command Line Interface)
- Il existe de nombreuses versions de GIT en mode GUI mais elles ne couvrent qu'une partie des features de GIT en mode CLI.



Github Desktop



Gitkraken



Fork

↗ <https://git-scm.com/downloads/guis>

# Local GIT repository

---

# Créer un GIT repository

Il y a deux façons d'obtenir un GIT repository, dans les deux cas vous aurez un Git repository en local sur votre machine, prêt à être utilisé.

1. Transformer un dossier local en GIT repository avec la commande `git init`.
2. Cloner un Git repository à partir d'une URL
  - Supposons que vous vouliez cloner le projet `MonSuperProjet` situé à l'URL `http://github.com/MonSuperProjet`, il suffira de taper la commande : `git clone http://github.com/MonSuperProjet`
  - Si vous vouliez le renommer `MonSuperProjetLocal`, il suffirait de taper la commande : `git clone http://github.com/MonSuperProjet MonSuperProjetLocal`

Dans les deux solutions, cela créera un sous-répertoire appelé ` .git` qui contiendra tous les fichiers nécessaires au Git repository.

## Les différents types de systèmes existants

**À vous de jouer:** Utiliser la console Git pour créer un dossier que vous transformerez en Git repository.



- **Fichier HEAD:** Suivi de votre branche actuelle
- **Fichier config:** Stockage des informations de configuration pour Git
- **Fichier index:** Suivi de la Staging Area
- **Dossier objects:** Stockage de la data sous la forme d'une série de Snapshots
- **Dossier refs:** Dossier stockant les références aux commits et aux branches
- **Dossier hooks:** Exécution de scripts à des moments précis du flux de travail Git

# Enregistrement des modifications

---

Chaque fichier dans le Working Directory peut être dans 2 états:

- **Tracked:** ce type de fichier est dans votre dernier commit et il peut être **unmodified**, **modified** ou **staged**. En résumé GIT le connaît déjà.
- **Untracked:** ce type de fichier est dans votre Working Directory mais il n'est **pas dans votre dernier commit, ni dans la Staging Area**. En résumé, GIT ne le connaît pas encore.

## Les différents types de systèmes existants

Pour connaître le statut de vos fichiers, utilisez la commande `git status` :

```
$ git status
On branch master

No commits yet

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

- Cela signifie que vous avez un Working Directory propre c'est à dire qu'aucun de vos fichiers tracked n'est modifié
- La commande vous indique aussi dans quelle branche vous êtes « **On branch master** ». Pour l'instant la branche est toujours `master` ce qui est l'état par défaut.

💡 Ce terme `master` provient de Bitkeeper, un précurseur de Git. Bitkeeper désignait la source de vérité comme le "master repository" et les autres copies comme des "slave repositories". Vous verrez de plus en plus le terme `main` pour des raisons politiques.

## À vous de jouer:

1. Créez un nouveau fichier `liste-de-courses.txt`
2. Ajouter le premier élément `* 3 tomates`
3. Sauvegardez et tapez la commande `git status`. Que voyez-vous ?

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    liste-de-courses.txt ●

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

## Les différents types de systèmes existants

4. Ajoutez le fichier dans la liste des fichiers **tracked** grâce à la commande `git add liste-de-courses.txt`.

```
$ git status
On branch master

No commits yet

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

Le fichier `liste-de-courses.txt` est-il dans la **Working Area**, la **Staging Area** ou le **.git Repository** ?

- **Working Area et Staging Area** uniquement. La commande `git add` permet d'ajouter le fichier dans la **Staging Area**. Il sera dans le **.git Repository** seulement après que le commit ait été réalisé via la commande `git commit`.

## Syntaxe Fileglobs

La commande `git add [<options>] [--] <pathspec>` peut prendre comme valeur de `pathspec` un filepath ou alors un **Fileglob** (Glob pattern syntax):

- `/` pour séparer les segments de chemin d'accès
  - **Exemple:** `path/to/file.txt`
- `*` pour faire correspondre zéro ou plusieurs caractères dans un segment de chemin d'accès
  - **Exemple:** `path/to/*` correspond à tous les fichiers dans `path/to/`
- `?` pour correspondre à un seul caractère dans un segment de chemin d'accès
  - **Exemple:** `?at.txt` correspond à `Bat.txt`, `cat.txt`, `Rat.txt`, ...
- `**` pour faire correspondre un nombre quelconque de segments de chemin, y compris aucun
  - **Exemple:** `path/**/*` correspond à tous les fichiers et dossiers dans `path/`

## Les différents types de systèmes existants

- `{}` pour regrouper des conditions
  - **Exemple:** `{**/*.html,**/*.txt}` correspond à tous les fichiers **HTML** et **texte**
- `[]` pour déclarer une plage de caractères à prendre en compte
  - **Exemple:** `exemple.[0-9]` correspond à tous les fichiers `exemple.0` , `exemple.1` , ...
- `[ ! ... ]` pour annuler une plage de caractères à rechercher
  - **Exemple:** `exemple.[!0-9]` correspond à tous les fichiers comme `exemple.a` , `exemple.b` , mais pas `exemple.0`

## Création d'un commit

---

Utilisez la commande `git commit` pour réaliser un commit:

- Avec un message: `git commit -m "Message du commit"`
- Avec un message et une description: `git commit -m "Message du commit" -m "Description"`

💡 Si vous n'utilisez pas l'option `-m`, GIT lance votre éditeur de texte, préalablement paramétré, et vous demande d'entrer un commentaire pour ce commit. Après avoir fermé votre éditeur de texte, GIT exécute le commit.

## Les différents types de systèmes existants

**À vous de jouer:** Créez un commit avec un message pertinent.

```
$ git commit -m "Add the grocery list"  
[master (root-commit) ee3bc50] Add the grocery list  
 1 file changed, 1 insertion(+)  
create mode 100644 liste-de-courses.txt
```

- GIT vous indique le fichier qui a été commité ainsi que le commentaire associé à ce commit, ici: « **Add the grocery list** ».
- Le fichier `liste-de-courses.txt` a été ajouté à un snapshot archivé/versionné dans le .git repository avec la checksum SHA-1 : `ee3bc50` .

## Les différents types de systèmes existants

Ajoutez maintenant un nouvel aliment dans la liste comme `* 2 abricots` et affichez le status.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   liste-de-courses.txt ●

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

GIT indique en rouge que le fichier `liste-de-courses.txt` a été modifié mais qu'il n'est pas encore **staged** : « **Changes not staged for commit** ». Comme ce fichier est déjà Tracked, on remarquera la présence du mot **modified**, en rouge devant le fichier, indiquant que ce fichier a déjà fait l'objet d'un archivage dans le .git repository

## Les différents types de systèmes existants

Ajoutez cette nouvelle version du fichier dans la Staging Area puis ajoutez encore un alimenter et afficher le status:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file> ..." to unstage)
    modified:   liste-de-courses.txt ●

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   liste-de-courses.txt ●
```

- GIT indique en **vert** que le fichier a été **staged** et est prêt pour être commité dans le .git repository (la liste contient 2 items)
- GIT indique en **rouge** que le fichier a été modifié mais qu'il n'est pas encore **staged** « Changes not staged for commit » (la liste contient 3 items)

## Les différents types de systèmes existants

**À vous de jouer:** Faites un commit de la version du fichier contenant 2 aliments.

- `git status` vous indique en rouge que le fichier `liste-de-courses.txt` a été modifié (il contient 3 items) mais qu'il n'est pas encore staged.
- Cependant `git status` ne vous dit pas exactement quelles sont les différences entre les versions.

# Annulation des modifications

- Pour de-stagger (**unstage**) un fichier, utilisez la commande `git reset HEAD monFichier`
- Pour supprimer (**discard**) des changements fait à un fichier et revenir à la dernière version committée utilisez la commande `git checkout -- monFichier`

**À vous de jouer:** Ajoutez un nouvel aliment à la liste que vous ajouterez au dernier commit. Veuillez renommer le commit.

```
$ git add *
$ git commit --amend -m 'Add the abricots and the cerises'
```

## Comparaison des versions

La commande `git diff` compare la version de la Working Area avec la dernière version présente dans la Staging Area.

**À vous de jouer:** Affichez les différences en utilisant `git diff`

```
$ git diff
diff --git a/liste-de-courses.txt b/liste-de-courses.txt
index f2f170d..5ea5ee9 100644
--- a/liste-de-courses.txt
+++ b/liste-de-courses.txt
@@ -1,2 +1,3 @@
 * 3 tomates
-* 2 abricots
\ No newline at end of file
+* 2 abricots
+* 5 cerises
\ No newline at end of file
```

## Les différents types de systèmes existants

Pour comparer la **Staging Area** avec le **.git repo**, utilisez la commande:

```
$ git diff --staged
```

## Renommer un fichier

À vous de jouer:

1. Renommez votre fichier affichez le status
2. Stagez les modifications et affichez le status

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file> ..." to unstage)
    renamed:   liste-de-courses.txt → liste.txt
```

💡 La documentation suggère que Myers est l'algorithme de différence par défaut. En théorie, un nouveau fichier doit être plus de la moitié identique à un fichier supprimé dans un commit donné pour être considéré comme un renommage.

## Supprimer un fichier

**À vous de jouer:** Supprimez le fichier et affichez le status.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    liste-de-courses.txt ●
```

# Historique des commits

La commande `git log` permet d'afficher l'historique de nos commits.

**À vous de jouer:** Affichez votre historique.

```
$ git log  
commit 9cb875d6ff6a2f6f84991e202d3ad01e687b1a05 (HEAD → master)  
Author: johannchopin <johannchopin@pm.me>  
Date:   Tue Aug 6 16:20:16 2024 +0200
```

Add the cerises

```
commit 88d904d6624373e440a9e8d6c9910180994c8ba6  
Author: johannchopin <johannchopin@pm.me>  
Date:   Tue Aug 6 15:14:21 2024 +0200
```

Add the abricots

 `git log` ouvre un programme `less` permettant de visualiser un fichier texte page par page. Appuyer sur `q` pour le fermer.

## Les différents types de systèmes existants

- `git log -p` : montre les différences (**patch**) dans chaque commit
- `git log -p -2` : montre les 2 derniers commits seulement
- `git log --stat` : montre des abréviations de stats pour chaque commit
- `git log --pretty=format:"%h - %an, %cd : %s"` : affiche le détail des commits en pretty print : SHA-1 code – nom de l'auteur, date du commit : commentaire du commit
- `git log --pretty=format:"%h %s" --graph` : affiche un graphique ASCII montrant les différentes *branches* et l'historique des merges

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
  \
  | * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
  /
* d6016bc require time for xschema
* 11d191e Merge branch 'defunkt' into local
```

## Les différents types de systèmes existants

- `git log --since=2.weeks` : montre les commits réalisés depuis 2 semaines
- `git log --since=2024-09-01` : montre les commits réalisés depuis YYYY-MM-DD

# Réécrire l'historique

- La principale tâche de Git est de s'assurer que vous ne perdez jamais un changement commité.
- Git est également conçu pour vous donner le contrôle total de votre workflow de développement et notamment de définir avec exactitude l'apparence de votre historique de projet.
- ⚠ Cela crée des conditions pouvant entraîner la **perte de commits**.



## Modification du dernier commit

- La commande `git commit --amend` permet de modifier le commit le plus récent.
- **Cette commande est à utiliser avec prudence** et dans des cas très précis.

**Exemple:** Oublie d'un fichier:

```
$ git commit -m 'initial commit'  
$ git add forgotten_file  
$ git commit --amend --no-edit
```

**Exemple:** Mauvais message de commit:

```
$ git commit --amend -m 'new message'
```

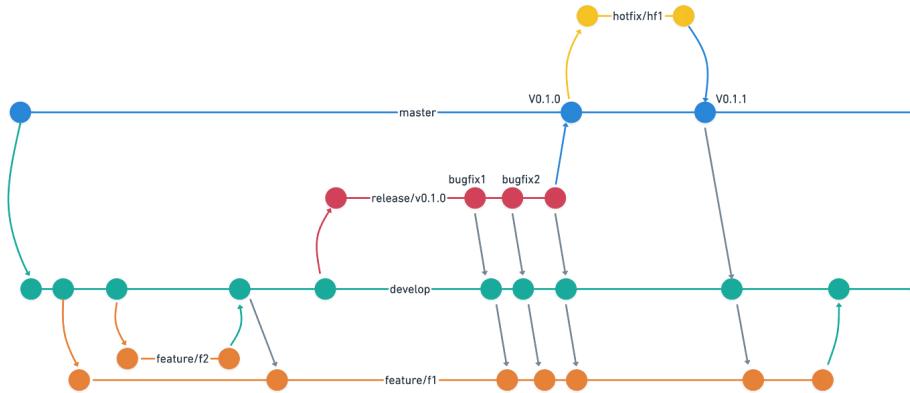
 Les commits modifiés sont en réalité des commits entièrement nouveaux.

## Modifier des commits plus anciens ou plusieurs commits

---

- La commande `git rebase` permet de combiner une séquence de commits dans un nouveau commit de base.
- Le rebasage est plus utile et facilement visible dans le contexte d'un workflow de branches.

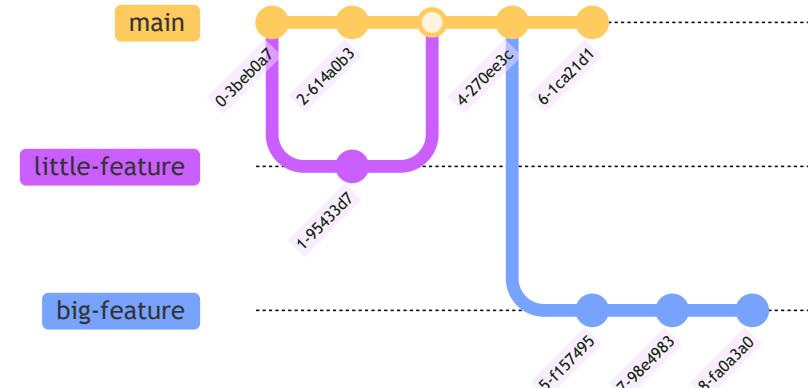
# GIT branching



# Introduction

---

- Une branche représente une ligne de développement indépendante.
- Les branches servent d'abstraction pour le processus d'édition/stage/commit.
- Vous pouvez les considérer comme un moyen de demander un tout nouveau répertoire de travail, une zone de transit et un historique du projet.
- Les nouvelles validations sont enregistrées dans l'historique de la branche en cours, ce qui entraîne une bifurcation dans l'historique du projet.

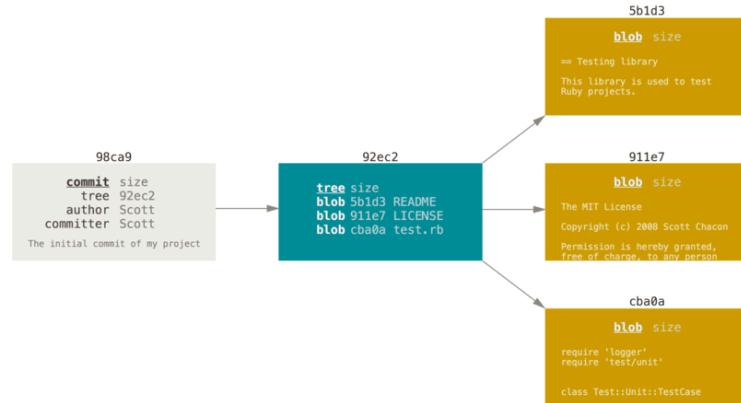


## Les différents types de systèmes existants

- Pour bien comprendre comment Git gère les branches, **il faut se rappeler comment Git stocke ses données.**
- Lors d'un commit, Git enregistre un **objet commit** qui contient un **pointeur** vers le snapshot du contenu que vous avez staggé.
- Cet objet contient entre autre:
  - nom de l'auteur
  - email
  - commentaire de commit
  - pointeurs vers le ou les commit(s) qui ont précédé directement ce commit (son ou ses parents)
    - zéro parent pour le commit initial
    - un parent pour un commit normal
    - plusieurs parents pour un commit résultant d'une fusion de deux branches ou plus.

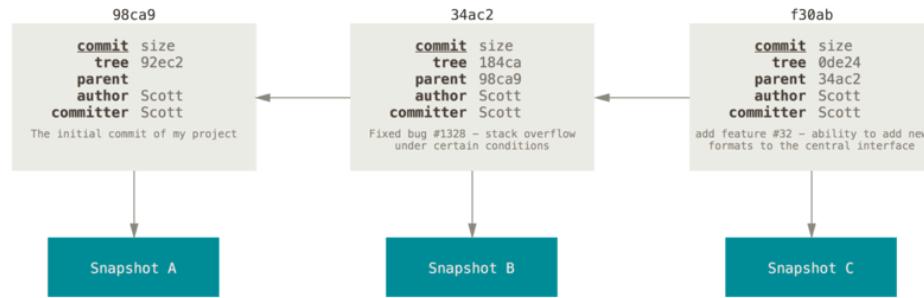
## Les différents types de systèmes existants

- **Exemple:** Supposons que vous avez un Working Directory contenant 3 ( README.md , LICENCE.md et test.rb ) fichiers et que vous les stagiez et commités tous.
- Votre Git Repo contiendra maintenant 5 objets:
  - **3 blobs** chacun représentant le contenu de l'un des 3 fichiers
  - **1 arbre** spécifiant quels fichiers sont stockés en tant que blobs
  - **1 commit** avec le pointeur qui pointe vers cet arbre et toutes les metadonnées de commit



## Les différents types de systèmes existants

- Si vous faites des changements et commitez à nouveau, les prochains commits enregistreront un pointeur vers le commit qui vient immédiatement avant lui.



Une branche dans Git est simplement un pointeur léger et mobile vers l'un de ces commits.

## Créer une nouvelle branche

- Utiliser la commande `git branch <branch_name>` pour créer une nouvelle branche.
- Cela crée un nouveau pointeur sur le même commit sur lequel on se trouve actuellement.

⚠ `git branch` crée uniquement la branche mais nous déplace pas dessus.

## Switcher sur une branche

---

Il existe 2 commandes possibles pour aller sur une branches:

- `git switch <branch>` (recommandé)
- `git checkout <branch>`

Pour créer directement une branche et s'y déplacer, utiliser:

- `git switch -c <branch>` (recommandé)
- `git checkout -b <branch>`

## git log --graph

```
$ git log --oneline --decorate --graph --all
```

Cette commande affiche l'historique de vos commits, et aussi où sont vos pointeurs de branche et comment votre historique a divergé:

```
$ git log --oneline --decorate --graph --all
```

```
* 74f5681 (HEAD → master) Add more cerises
| * 4d23f91 (bbq) Barbecue list
|/
* dc42194 (test) Add the cerises
* 88d904d Add the abricots
* ee3bc50 first commit
```

## Supprimer une branche locale

---

- Utilisez la commande `git branch -d <branch>` pour supprimer une branche sans commits.
- La commande `git branch -D <branch>` supprimera une branche qui contient des commits.

## Renommer une branche locale

---

- Utilisez la commande `git branch --move <name> <new-name>` pour renommer une branche.

## Exercice

---

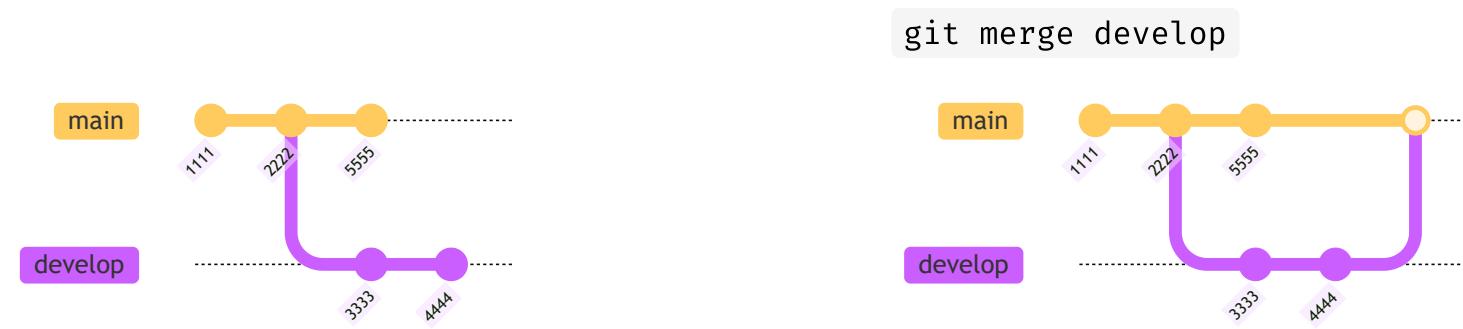
Nous voulons organiser un barbecue, nous avons donc besoin d'une liste de courses dédiée. Dans votre git repository contenant le fichier `liste-de-courses.txt` :

1. Créez une nouvelle branche `bbq` et ajoutez y le nécessaire
2. Vous avez oublié dans la liste initiale (`main`) une salade. Ajoutez-la dans un commit sans **passez par la Staging Area**.
3. Il semble qu'un anniversaire aura lieu lors de votre barbecue. Créez une nouvelle branche à partir de `bbq` pour y ajouter les idées de cadeaux .
4. Malheureusement, l'anniversaire est annulé. Supprimez la branche.

Notez dans votre bloc-notes toutes les commandes utilisées pour la correction.

# Merger des branches

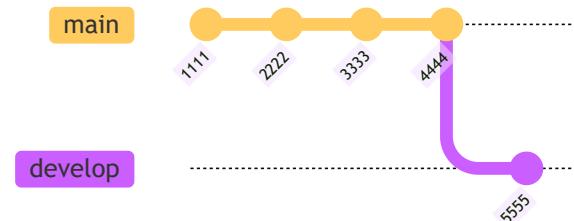
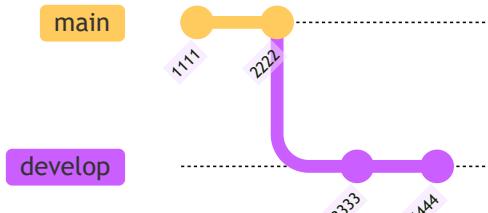
La commande `git merge <branch>` permet de merger une branche dans la branche actuelle. La commande combinera plusieurs séquences de commits en un seul historique unifié.



## Fast forward merge

- Par défaut, Git va essayer de merger 2 branches en faisant une fusion en avant rapide (**fast-forwards**).
- Cela est possible lorsqu'il y a un chemin linéaire entre l'extrémité de la branche courante et la branche cible.
- Au lieu de fusionner "réellement" les branches, Git déplace l'extrémité de la branche courante jusqu'à l'extrémité de la branche cible.
- Il est possible de désactiver cela et d'avoir toujours un merge commit avec l'option `--no-ff`.

git merge develop



## Exercice

---

Nous allons faire des courses:

1. Créez et placez vous sur une nouvelle branche `grocery-shopping`
2. Supprimez des articles dans notre `liste-de-courses.txt`
3. Les courses sont finies: mergez la branche `grocery-shopping` dans `main`
4. Supprimez la branche `grocery-shopping`

**Question:** Quel est le schéma du git graph une fois ces actions réalisées?

## Résolution des conflits lors des merges

De temps en temps, **le process de merge ne se passe pas bien** ! Cela se produit quand vous modifiez la même partie du même fichier dans les deux branches que vous voulez merger. Dans ce cas, GIT est incapable de merger et indiquera un **conflit**:

```
$ git merge bbq
Auto-merging liste-de-courses.txt
CONFLICT (content): Merge conflict in liste-de-courses.txt
Automatic merge failed; fix conflicts and then commit the result.
```

## Les différents types de systèmes existants

Pour savoir à tout moment quels fichiers n'ont pas été mergé et posent des conflits, utilisez la commande `git status`.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:  liste-de-courses.txt
```

## Les différents types de systèmes existants

**À vous de jouer:** Mergez la branche `bbq` dans `main` et ouvrez le fichier `liste-de-courses.txt`.

```
* 3 tomates
<<<<< HEAD
* 2 abricots
=====
* 2 abricots
* 5 cerises
* 3 saucisses
>>>> bbq
```

## Les différents types de systèmes existants

- Pour tous les fichiers qui ont des conflits de merge non résolus, GIT ajoute des **marqueurs de conflit** ( <<<<<, \_\_\_\_\_, et >>>> ) dans les fichiers afin que vous les éditez et **résolviez les conflits manuellement**.
- En général, le contenu avant le marqueur \_\_\_\_\_ est la branche réceptrice et la partie après est la branche de fusion.

Pour résoudre un conflit, modifiez le fichier impacté pour conserver ce qui est nécessaire tout en supprimant les marqueurs de conflit. **Stagger** ensuite le fichier pour indiquer à GIT que le conflit est résolu.

### Visualisation du conflit:

```
* 3 tomates
<<<<< HEAD
* 2 abricots
_____
* 2 abricots
* 5 cerises
* 3 saucisses
>>>>> bbq
```

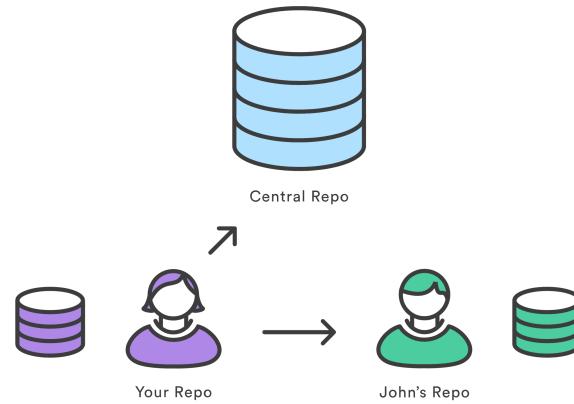
### Résolution du conflit:

```
* 3 tomates
* 2 abricots
* 3 saucisses
```

Les différents types de systèmes existants

**À vous de jouer:** Résolvez le conflit. Quel est le schéma du git graph une fois ce merge réalisé?

# Remote GIT repository



# Introduction

Les **.git Remote Repositories** sont des versions de votre projet qui sont hébergées sur internet par un service provider.



- L'action d'uploader de votre .git Repo local vers un remote repository se nomme `push`.
- L'action de télécharger d'un remote repository vers votre .git Repo local se nomme `fetch`.

## Les différents types de systèmes existants

Pour ce cours, il vous sera demandé d'utiliser un compte GitHub pour réaliser les travaux pratiques.



 <https://github.com/login>

# Authentification SSH

---

- Une clé SSH est un identifiant d'accès pour le protocole réseau **SSH** (Secure Shell). Ce protocole réseau sécurisé authentifié et chiffré est utilisé pour la communication à distance entre des machines sur un réseau ouvert non sécurisé.
- SSH utilise une paire de clés pour établir une liaison sécurisée entre des parties distantes.
  - **Clé privée:** une clé gardée secrète par l'utilisateur SSH sur sa machine cliente permettant le déchiffrement.
  - **Clé publique:** une clé publiquement donnée aux parties distantes permettant le chiffrement.

**À vous de jouer:** Si nécessaire, générez une clé SSH pour vous connecter à GitHub:

1. **Génération d'une nouvelle clé SSH:** <https://docs.github.com/fr/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>
2. **Ajout de la clé SSH à votre compte GitHub:**  
<https://docs.github.com/fr/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>

# Ajout d'un remote repository

Pour push votre .git Repository local sur votre remote sur GitHub:

1. Votre *git\_repo* doit exister préalablement sur GitHub ! S'il n'existe pas encore, créez-le en lui donnant le nom *git\_repo* depuis <https://github.com/new>!
  2. Positionnez-vous dans votre .git Repository local
  3. Tapez la commande: `git remote add origin git@github.com:<user>/<project>.git`
  4. Tapez enfin: `git push -u origin main`
- 
- Le nom ajouté par défaut par git au remote repository est **origin**. Mais vous pouvez aussi le redéfinir et lui donner un nom plus explicite avec la commande:

```
git remote add <alias> <url_repo>
```

## Les différents types de systèmes existants

**À vous de jouer:** Créez un repository public GitHub pour y pusher votre .git Repository local.

## Cloner un remote repository

---

Si un projet est déjà configuré dans un dépôt centralisé, la commande `git clone` est la plus courante pour obtenir une copie de développement. 2 types d'adresses URL existent pour cloner un projet:

- Une URL SSH: `git clone git@github.com:<user>/<project>.git` (recommandé)
- Une URL HTTPS: `git clone https://github.com/<user>/<project>.git`

## Lister vos remotes repositories

---

- `git remote` - Pour voir vos remotes (shortname seulement)
- `git remote -v` - Pour voir vos remotes (shortname + URL)

# Fetching: télécharger du remote en local

- `git fetch` : télécharge uniquement les données du remote sans modifier votre HEAD.

**À vous de jouer:** Ajoutez un aliment dans `liste-de-courses.txt` depuis l'interface de GitHub.  
Faites un `git fetch` et affichez le git graph.

```
$ git log --oneline --decorate --graph --all
* 04d8095 (origin/main) Add 2 bananes
* 34ca472 (HEAD → main) Add more cerises
* dc42194 Add the cerises
```

- Les branches distantes sont exactement comme les branches locales, sauf qu'elles correspondent à des modifications provenant du remote de quelqu'un d'autre.
- Les branches distantes sont **préfixées par le remote auquel elles appartiennent** afin de ne pas les confondre avec les branches locales.

## git pull

La commande `git pull` est utilisée pour récupérer et télécharger le contenu d'un remote repository et mettre immédiatement à jour le repository local pour qu'il corresponde à ce contenu.

**À vous de jouer:** Ajoutez à nouveau un aliment dans `liste-de-courses.txt` depuis l'interface de GitHub. Faites un `git pull` et affichez le git graph.

```
$ git log --oneline --decorate --graph --all
* 03fg423 (HEAD → main, origin/main) Add 2 kiwis
* 04d8095 Add 2 bananes
* 34ca472 Add more cerises
* dc42194 Add the cerises
```

## Pushing: Uploader du local vers le remote

La commande `git push` est utilisée pour uploader le contenu d'un repository local (les commits) vers un remote repository.

```
$ git push <remote> <branch>
```

- Si la branche remote a divergé de la votre, vous devez **pull** la remote branche et la **merger** avec votre branche locale, puis réessayer de **push**.
- L'option `--force` fait en sorte que la branche du remote repository corresponde à la branche locale, en supprimant toutes les modifications qui ont pu avoir lieu depuis le dernier pull.
- **⚠ Soyez sûr de ce que vous faites, vous pourriez perdre de nombreuses heures de travail.**

# Tags

---

GIT permet de **tagger** dans l'historique des commits qui ont une **importance**. Pour ce faire, placez-vous dans le commit que vous souhaitez tagger et exécutez la commande appropriée :

- `git tag <version>` - Crée un **lightweight tag**, un simple pointeur vers un commit spécifique.
- `git tag -a <version> -m "<description>"` - Crée un **annotated tag** contenant d'avantages d'informations comme le nom, l'email et la date de l'auteur du tag en plus d'une description si définie.
- `git show <version>` - Affiche le détail des données relatives au tagging d'une version particulière.
- `git tag -a <version> <commit-checksum>` - Crée un **annotated tag** sur le commit avec le checksum `commit-checksum`
- `git push <origin> <tag-version>` - Permet de pusher un tag sur un remote repository
- `git push <origin> --tags` - Permet de pusher tous les tags sur un remote repository



## SemVer

---

Lorsque vous créez un logiciel, vous devez être en mesure de suivre son évolution afin de pouvoir continuer à l'utiliser sans risques au fil de son développement.

Une convention est alors apparue pour donner des numéros de version normalisés, **SemVer** (Semantic Versioning).

**MAJOR.MINOR.PATCH** (exemple: 1.4.55 )

Lors d'une nouvelle version, incrémentez le numéro de version:

- **MAJOR** quand il y a des changements non rétrocompatibles
- **MINOR** quand il y a des ajouts de fonctionnalités rétrocompatibles
- **PATCH** quand il y a des corrections d'anomalies rétrocompatibles

# TP: Utilisation d'une GUI

---

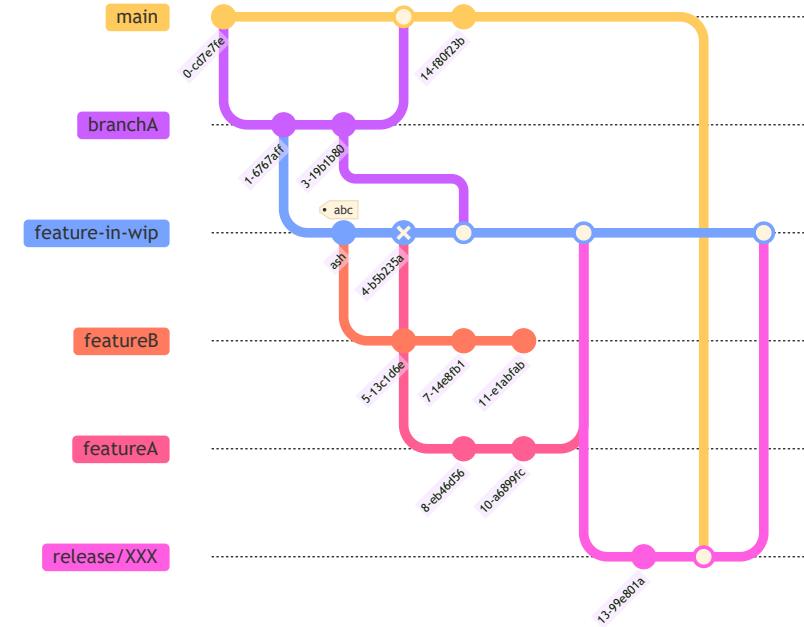


↗ <https://git-fork.com>

Clonez le repository <https://github.com/isfates-l3-outils-et-methodes-de-dev/liste-de-courses>

# Git workflow

Dans Git, un modèle de branches peut vite devenir ... chaotique !

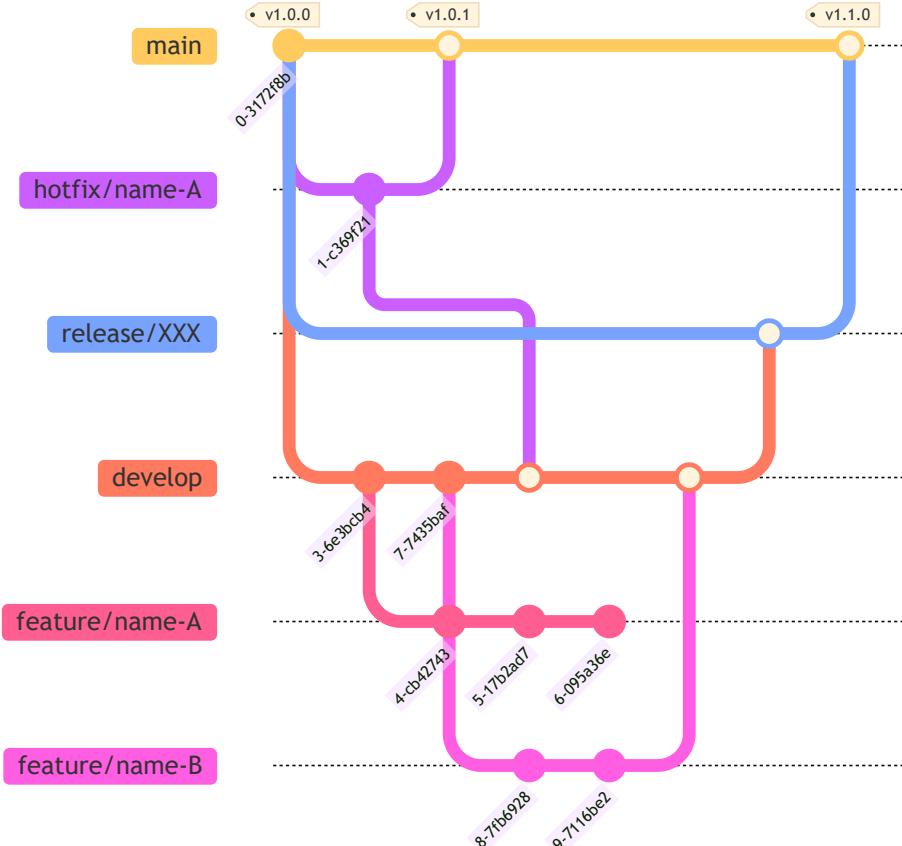


## GitFlow

---

- GitFlow est un workflow Git existant qui, à l'origine, était une stratégie innovante de gestion des branches Git.
- GitFlow a de nombreuses branches à durée de vie plus longue et des commits plus nombreux. Dans ce modèle, les développeurs créent une **feature branch** et retardent le merge avec la branche principale jusqu'à ce que la fonctionnalité soit terminée.
- GitFlow attribue des rôles très spécifiques à différentes branches et définit comment et quand elles doivent interagir

## Les différents types de systèmes existants



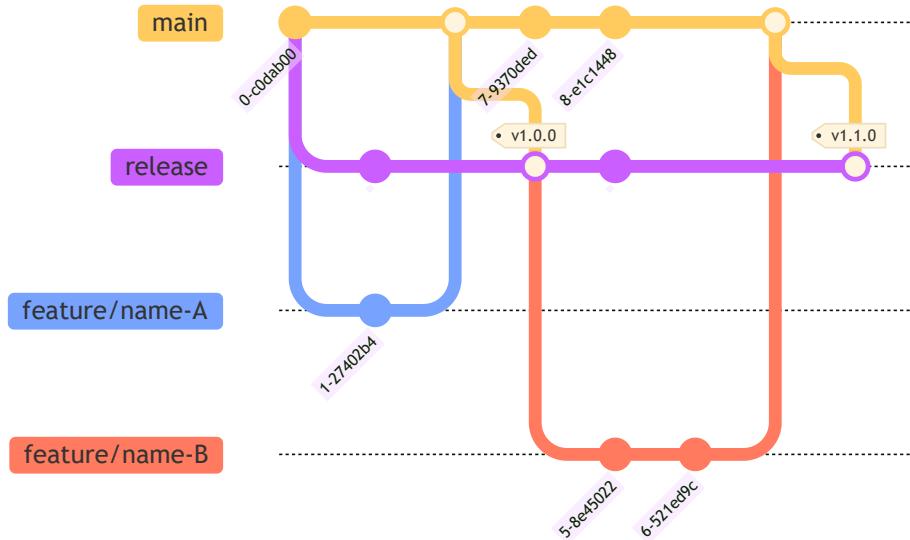
- **main** : branche reliée à la mise en production de l'application
- **develop** : branche contenant toutes les features finalisées
- **feature/XXXX** : branches permettant de travailler sur une nouvelle feature
- **release/XXXX** : branches permettant de faire la liaison entre **develop** et **main**
- **hotfix/XXXX** : branches permettant de publier rapidement une correction depuis **main**

# Trunk-based development

---

- Dans un workflow de trunk-based development, tout est basé sur la branche `main`, qui peut être appelée **trunk** et qui signifie tronc (penser à un tronc d'arbre duquel toutes les branches partent)
- Les développeurs créent des feature branches **qui durent peu de temps**, avec peu de commits et mergent très fréquemment de petits changements sur la branche principale.
- Cela assure une grande fluidité même quand la taille de l'équipe et la complexité du code augmentent.
- Afin de déployer une nouvelle version en production, une release branch est tiré du trunk, taguée avec le nouveau numéro de version.

## Les différents types de systèmes existants



# GitFlow vs Trunk-based development

---

## Quand utiliser Gitflow ?

- Le projet est conséquent
- Le cycle de release est régulier
- La qualité est primordiale
- L'équipe de développement est grande

## Quand utiliser trunk-based development ?

- Le project est plutôt récent et de petite taille
- Besoin de délivrer de la feature rapidement
- Les développeurs sont expérimentés
- L'équipe de développement est assez restreinte

# Conventional Commits

---

```
* 7d0fc3e typo
* 8fc509a more changes
* efe5fc5 add test
* 447c5a0 updates
* 1189cf0 update condition
* 68abca0 updates
* 29f73ed more changes
* cefaa18 add file
```



- **Conventional Commits** est une spécification ajoutant une signification lisible pour l'humain et pour la machine dans les messages des commits.
- Elle fournit un ensemble simple de règles pour créer un **historique de commit explicite**, ce qui facilite l'écriture d'outils automatisés.
- Cette convention suit la convention SemVer.

# Syntaxe

```
<type>[optional scope]: <description>
```

```
[optional body]
```

```
[optional footer(s)]
```

Liste des type s:

- `feat` : Une nouvelle fonctionnalité
- `fix` : Correction d'un bug
- `build` : Changements qui affectent le système de build ou les dépendances externes (webpack, maven, npm, ...)
- `ci` : Changements dans nos fichiers de configuration et scripts CI (exemples : Travis, Circle, BrowserStack, SauceLabs)
- `docs` : Changements dans la documentation uniquement
- `perf` : une modification du code qui améliore les performances
- `refactor` : une modification du code qui ne corrige pas de bug et n'ajoute pas de fonctionnalité
- `style` : Changements qui n'affectent pas la signification du code (espaces blancs, formatage, points-virgules manquants, etc.)
- `test` : Ajout de tests manquants ou correction de tests existants

## Les différents types de systèmes existants

- Un commit de type `fix` induit une nouvelle version **PATCH** dans la spécification SemVer.
- Un commit de type `feat` induit une nouvelle version **MINOR** dans la spécification SemVer.
- Un commit dont le `type / scope` est suffixé d'un `!`, ou contient dans son `footer` le mot-clé `BREAKING CHANGE:`, introduit un BREAKING CHANGE et induit une nouvelle version **MAJOR** dans la spécification SemVer.
  - Exemple: `feat!: send an email to the customer`
  - Exemple avec un `scope` : `feat(api)!: send an email to the customer`

## Pourquoi l'utiliser?

---

- Générer automatiquement des CHANGELOGs.
- Déterminer automatiquement un changement de version SemVer (en fonction des types de commits inclus).
- Communiquer la nature des changements aux membres de l'équipe, au public et aux autres parties prenantes.
- Déclencher des processus de génération et de publication.
- Faciliter la contribution des personnes à vos projets en leur permettant d'explorer un historique de commit plus structuré.

# Exemple de CHANGELOG

v16.0.0

release-please released this Sep 18, 2023 v16.0.0 · a55a85c

**16.0.0 (2023-09-18)**

**⚠ BREAKING CHANGES**

- require node 18+ ([#2069](#))

**Features**

- Fallback to root package version if package ignores github release ([#1935](#)) ([0e11d4c](#))
- Require node 18+ ([#2069](#)) ([5a50247](#))
- Support pnpm workspaces ([#2058](#)) ([13cba14](#))
- Use default updaters based on file extension ([#2072](#)) ([1ee162b](#))

**Bug Fixes**

- deps: Bump semver dependency version ([#2068](#)) ([cd0bd85](#))
- deps: Replace lerna dependency with lerna-lite ([13cba14](#))
- deps: Upgrade http-proxy-agent to v7 ([#2071](#)) ([cc50a34](#))
- deps: Upgrade https-proxy-agent to v7 ([cc50a34](#))

# Exercice

À vous de jouer: Donnez les versions des différents tags:



# Git Cheatsheet

---

 Atlassian Git Cheatsheet

# Ressources

---

Ce cours a été élaboré à partir des ressources suivantes :

- Cours précédents de [in Gary Zingraff](#)
- <https://git-scm.com/book/en/v2/>
- <https://openclassrooms.com/fr/courses/7162856-gerez-du-code-avec-git-et-github>
- <https://www.atlassian.com/fr/git/>
- <https://grafikart.fr/tutoriels/git-presentation-1090>
- <https://blog.git-init.com/snapshot-vs-delta-storage/>
- <https://dev.to/rajaniraiyn/understanding-the-contents-of-the-git-folder-eff>
- <https://chelseatroy.com/2020/05/09/question-how-does-git-detect-renames/>
- <https://blog.jcoglan.com/2017/02/12/the-myers-diff-algorithm-part-1/>
- <https://semver.org/>
- <https://www.conventionalcommits.org/en/v1.0.0/>

