

# GIT

## L'essentiel

### Table des matières

I - Principaux objectifs pédagogiques.....	6
II - Présentation de GIT.....	7
1. Présentation de GIT.....	7
1.1. Fonctionnalité d'un gestionnaire de versions.....	7
1.2. Historique et évolution (cvs , svn, git).....	7
1.3. Fonctionnement distribué de GIT.....	8
1.4. Structure décentralisée de Git (vue d'ensemble).....	9
1.5. Hébergement de référentiel GIT.....	10
1.6. Installation et configuration minimum :.....	11
1.7. GIT , Tortoise-GIT , intégration IDE et SmartGit, GitKraken.....	12
1.8. Utilitaires GIT graphiques de bases.....	13
1.9. Apprentissage progressif de GIT.....	13
III - Bases locales de GIT.....	14

1. Principales commandes de GIT (en mode local).....	14
2. initialisation d'un projet GIT (en local).....	15
3. Index (staging area).....	17
3.1. fichiers indexés par GIT.....	17
3.2. Affichage de l'historique courant.....	19
3.3. git commit et "show log" via tortoiseGit (sous windows).....	22
3.4. cycle des mises à jour.....	23
4. indispensable .gitignore.....	25
5. Commit et tags.....	26
5.1. Retour sur ancienne version.....	28
5.2. Comparaisons et affichages des modifications (git diff).....	30
5.3. Affichage d'une ancienne version sans checkout.....	31
5.4. Git add et commit d'une partie seulement des modifications effectuées sur un fichier.	31

## IV - Gestion des branches / GIT.....33

1. Gestion des branches avec GIT.....	33
2. Principales commandes de GIT pour les branches.....	35
2.1. Gestion élémentaire des branches.....	35
3. Bonnes pratiques dans la gestion des branches.....	38
4. Merge.....	39
4.1. merge rapide (fast-forward).....	39
4.2. merge sans conflits.....	40
4.3. merge avec conflits.....	41
5. Git rebase (souvent sur branche privée).....	43
6. Git cherry-pick.....	45
6.1. Comparaisons de deux branches via git diff.....	45
6.2. correctif (fixup) sur ancien commit.....	46
6.3. Fusion de plusieurs commits en un seul (squash).....	46
7. Git stash (enregistrer un travail brouillon à finir).....	49

## V - GIT en mode distant (--bare , github, ...).....50

1. Commandes de GIT pour le mode distant.....	50
2. Gérer plusieurs référentiels distants.....	51
3. initialiser git en mode distant.....	52
4. pistage (track) entre branches locales et distantes.....	53
5. Git fetch et git pull.....	55
6. Opérations diverses sur branches distantes :.....	57
6.1. Supprimer une branche distante.....	57
6.2. Envoyer des tags vers le référentiel distant.....	57

<b>VI - GitHub , GitLab et sécurité associée.....</b>	<b>58</b>
1. GitHub , GitLab et Bitbucket.....	58
2. Sécurité git (tokens , credential-manager, ... ).....	59
2.1. comptes utilisateurs, propriétaire et développeurs en équipe.....	59
2.2. Token pour opérations git distantes sans mot de passe.....	60
2.3. Url git avec token.....	62
2.4. Url git en mode ssh.....	64
2.5. Configurations de git coté client.....	65
2.6. Credential manager (sur poste de dev. , en local).....	66
2.7. Gestion simplifiée des "credentials" via l'interface graphique de tortoise git.....	67
2.8. Invitation d'un nouveau collaborateur sur un projet github.....	68
<b>VII - Git-flow et aspects divers.....</b>	<b>70</b>
1. Quelques workflows pour GIT.....	70
1.1. git-flow (évolué mais trop complexe).....	70
1.2. github-flow ( simple).....	71
1.3. Gitlab flow (moins rapide , fiabilité privilégiée).....	72
2. GitHub "pull request" et GitLab "merge request".....	73
2.1. Pull request (github).....	73
2.2. Merge request (requête de fusion) avec gitlab :.....	76
3. Fork de dépôt git.....	77
4. Git patch.....	77
4.1. Création d'un patch.....	77
4.2. Application d'un patch.....	77
5. utilisation d'un patch pour réorganiser commits.....	78
6. Réorganisation des commits lors d'un merge:.....	79
7. RAZ d'un référentiel git (RAZ de l'historique).....	80
<b>VIII - Ancienne architecture centralisée (CVS, SVN).....</b>	<b>82</b>
1. SVN.....	82
1.1. Terminologie & fonctionnement de CVS et SVN.....	82
<b>IX - Annexe – Structure interne de GIT (objets).....</b>	<b>84</b>
1. Structure interne d'un référentiel GIT.....	84
1.1. Contenu du répertoire caché .git.....	84
1.2. Blobs , Tree et références.....	84
<b>X - Annexe – GIT avec IDE eclipse.....</b>	<b>87</b>
1. Plugin eclipse pour GIT (EGIT).....	87
1.1. Actions basiques (commit , checkout , pull , push).....	87

1.2. Résolution de conflits.....	87
1.3. Sécurité GIT.....	88
1.4. Autres fonctionnalités eclipse/git.....	88
1.5. Perspective "Git" de eclipse (vue d'ensemble sur les branches).....	88

## **XI - Annexe – GIT avec IDE intelliJ.....90**

1. Bien préparer l'url du référentiel distant.....	90
2. GIT avec IntelliJ.....	90
2.1. Commit au sein de IntelliJ.....	90
2.2. Menu contextuel général "GIT" de IntelliJ.....	91
2.3. Merge Tool d'intelliJ.....	92

## **XII - Annexe – GIT avec IDE VisualStudioCode.....93**

1. Bien préparer l'url du référentiel distant.....	93
2. Git au sein de l'IDE "Visual Studio Code".....	93
2.1. commit avec VsCode.....	94
2.2. push après un commit.....	94
2.3. Merge tool (when conflict).....	95
2.4. Show git history via "git graph extension".....	95

## **XIII - Annexe – Serveur GIT "on premise".....96**

1. GitLab "On Premise".....	96
1.1. Via docker.....	96
2. Très ancienne configuration d'accès distant à un référentiel Git.....	97
2.1. Accès distant (non sécurisé) via git.....	97
2.2. Accès distant sécurisé via git+ssh.....	97
2.3. Accès distant en lecture seule via http (sans webdav).....	97
2.4. Accès distant en lecture/browsing via gitweb.....	98
2.5. Accès distant "rw" via http/https (webdav).....	99

## **XIV - Annexe – Bibliographie, Liens WEB + TP.....101**

1. Bibliographie et liens vers sites "internet".....	101
2. TP.....	101
2.1. Installation de git en mode texte.....	101
2.2. Configuration locale fondamentale de git.....	101
2.3. Eventuelle installation de compléments graphiques :.....	102
2.4. Git élémentaire en mode local.....	103
2.5. Gestion élémentaire des branches de git.....	105
2.6. Merge de fusion sans conflit.....	107
2.7. Merge git avec résolution de conflit.....	107
2.8. Git élémentaire en mode remote (clone, push, pull).....	109

---

2.9. Merge git avec résolution de conflit et branches distantes.....	110
2.10. github (ou autre) en mode distant et réel avec un accès restreint sur un référentiel préparé (compte "formateur").....	111
2.11. Git avec résolution de conflit depuis un IDE (Tp facultatif).....	112
2.12. Git en mode http (via github ou autre , tp facultatif).....	112
2.13. Expérimentation de "rebase" sur branches locales.....	113
2.14. Expérimentation de "cherry-pick" sur branches locales.....	113
2.15. Expérimentation de "stash".....	114
2.16. Autres expérimentations libres.....	114

## **I - Principaux objectifs pédagogiques**

- **Connaître les fonctionnalités apportées par un système de gestion de versions.**
- **Comprendre les mécanismes internes de GIT et le fonctionnement distribué de Git (référentiels locaux et distants).**
- **Bien savoir utiliser les principales commandes de git au quotidien.**
- **Savoir gérer les branches (variantes).**
- **Savoir travailler en équipe avec GIT (merge, gestion des conflits).**
- **Savoir configurer un référentiel distant sur github (ou autre) et établir une connexion depuis un référentiel local.**
- **Savoir installer GIT sur un poste de développement (windows ou linux) et connaître les outils complémentaires (bitbucket , tortoisegit , ...)**
- **Connaître quelques bonnes pratiques (gitflow , pull request, ...).**

## II - Présentation de GIT

### 1. Présentation de GIT

#### 1.1. Fonctionnalité d'un gestionnaire de versions

Le sigle "SCM" signifie "**S**ource **C**ode **M**anagement" et désigne généralement l'ensemble des technologies possibles qui permettent de gérer les différentes versions d'un logiciel développé en équipe .

Principales fonctionnalités d'un gestionnaire de versions :

- **mémoriser différentes versions du code source d'un logiciel** (et pouvoir revenir si besoin sur une ancienne version)
- **partager ce code versionné entre les différents développeurs d'une même équipe.**
- gérer des **branches** (différentes **variantes** du logiciel)
- autres fonctionnalités secondaires (...)

#### 1.2. Historique et évolution (cvs , svn, git)

Historique et évolution des "SCM" (Source Code Management) Tools :

<b>Depuis 1972</b> (d'origine IBM , porté sur Unix)	<b>SCCS</b> (Source Code Control System)	Projet fondateur ayant inspiré les outils postérieurs
<b>Depuis novembre 1990</b> , dernière version en 2008	<b>CVS</b> (Concurrent Version System)	<b>Système centralisé</b> et libre (licence public GNU)
<b>À partir de octobre 2000</b> (pris en charge par "apache foundation" depuis 2010) , existe encore en 2024.	<b>SVN / subversion</b>	Successeur officiel de CVS (avec meilleur implémentation). <b>Système toujours centralisé</b>
<b>A partir de avril 2005</b>	<b>GIT</b> (créé par " <i>Linus Torvalds</i> " l'inventeur de linux )	<b>Système décentralisé</b> et libre (licence public GNU2)
...	...	...

Au début des années 2000 , SVN a petit à petit remplacé CVS .  
A partir de 2008/2010 , GIT a petit à petit remplacé SVN.  
Certains anciens projets sont encore basés sur SVN.

NB : une des annexes montre la structure centralisée de CVS/SVN .

## 1.3. Fonctionnement distribué de GIT

### Présentation de GIT

**GIT** est un **système de gestion du code source** (avec prise en charge des différentes **versions**) qui fonctionne en **mode distribué** .

GIT est moins centralisé que SVN . Il existe deux niveaux de référentiel GIT (local et distant).

Un référentiel GIT est plus compact qu'un référentiel SVN.

**GIT** a été conçu par **Linus Torvalds** (l'inventeur de **linux**) .

Un produit concurrent de GIT s'appelle « **Mercurial** » et offre à peu près les mêmes fonctionnalités.

### Mode distribué de GIT

*Dans un système « scm » centralisé (tel que CVS ou SVN) , le référentiel central comporte toutes les versions des fichiers et chaque développeur n'a (en général) sur son poste que les dernières versions des fichiers.*

**Dans un système « scm » distribué (tel que GIT ou Mercurial) , le référentiel central ne sert que pour échanger les modifications et chaque développeur a (potentiellement) sur son poste toutes les versions des fichiers.**

NB : **Bazaar** est un gestionnaire de gestion de version codé en langage python . La dernière version de "Bazaar" date de 2016 . autrement dit , cet outil est aujourd'hui obsolète .

NB2 : "**Mercurial**" est un concurrent de GIT (fonctionnant également en mode distribué). Mercurial a également été créé en 2005 et est toujours maintenu en 202x .

Interêt de mercurial : syntaxe plus simple (du côté des lignes de commandes)

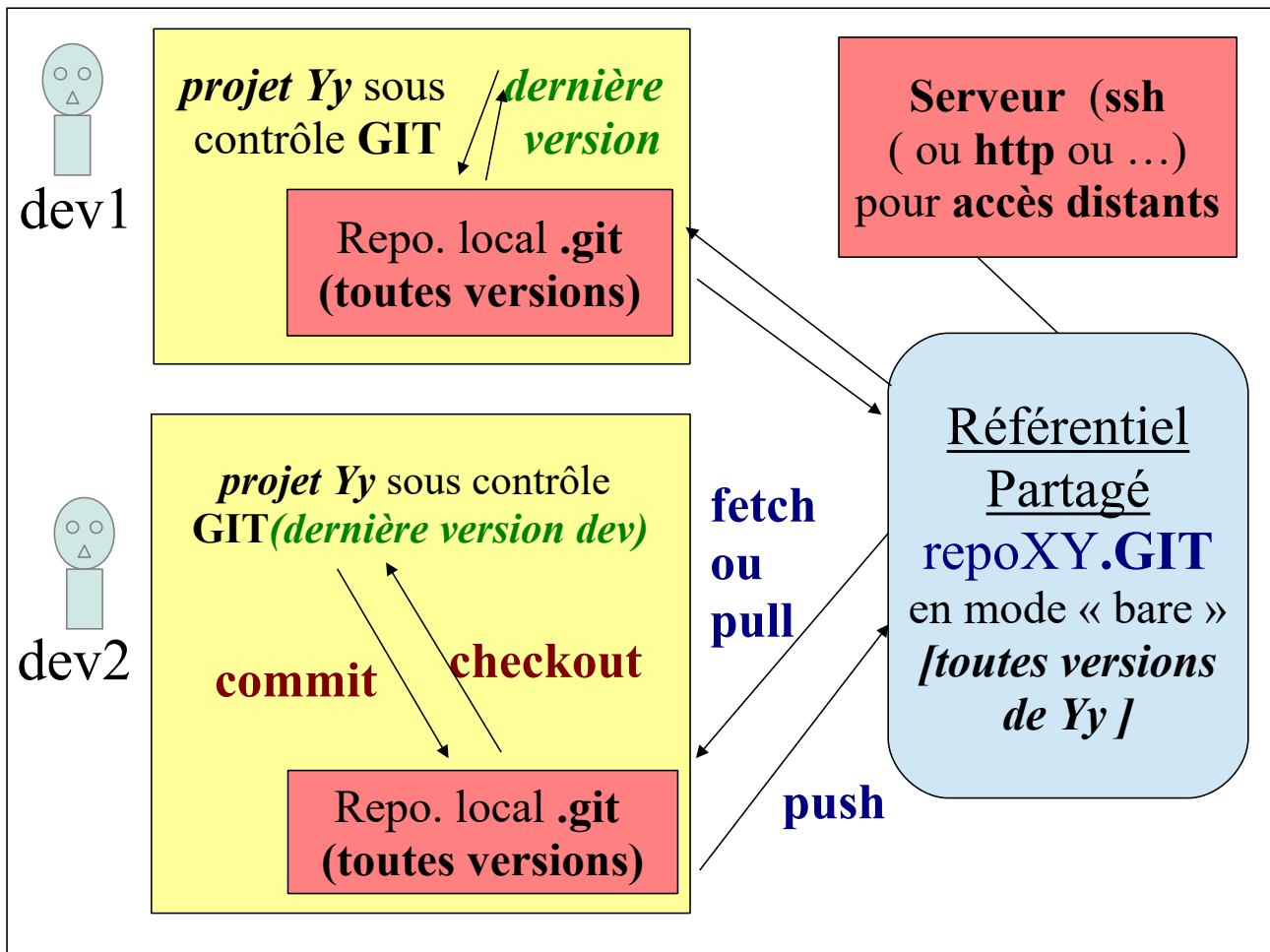
**Inconvénient de mercurial** : *moins de fonctionnalité que GIT* et **très peu utilisé** (environ 2 % des projets)

Anecdote sur l'origine du terme "GIT" (source wikipédia) :

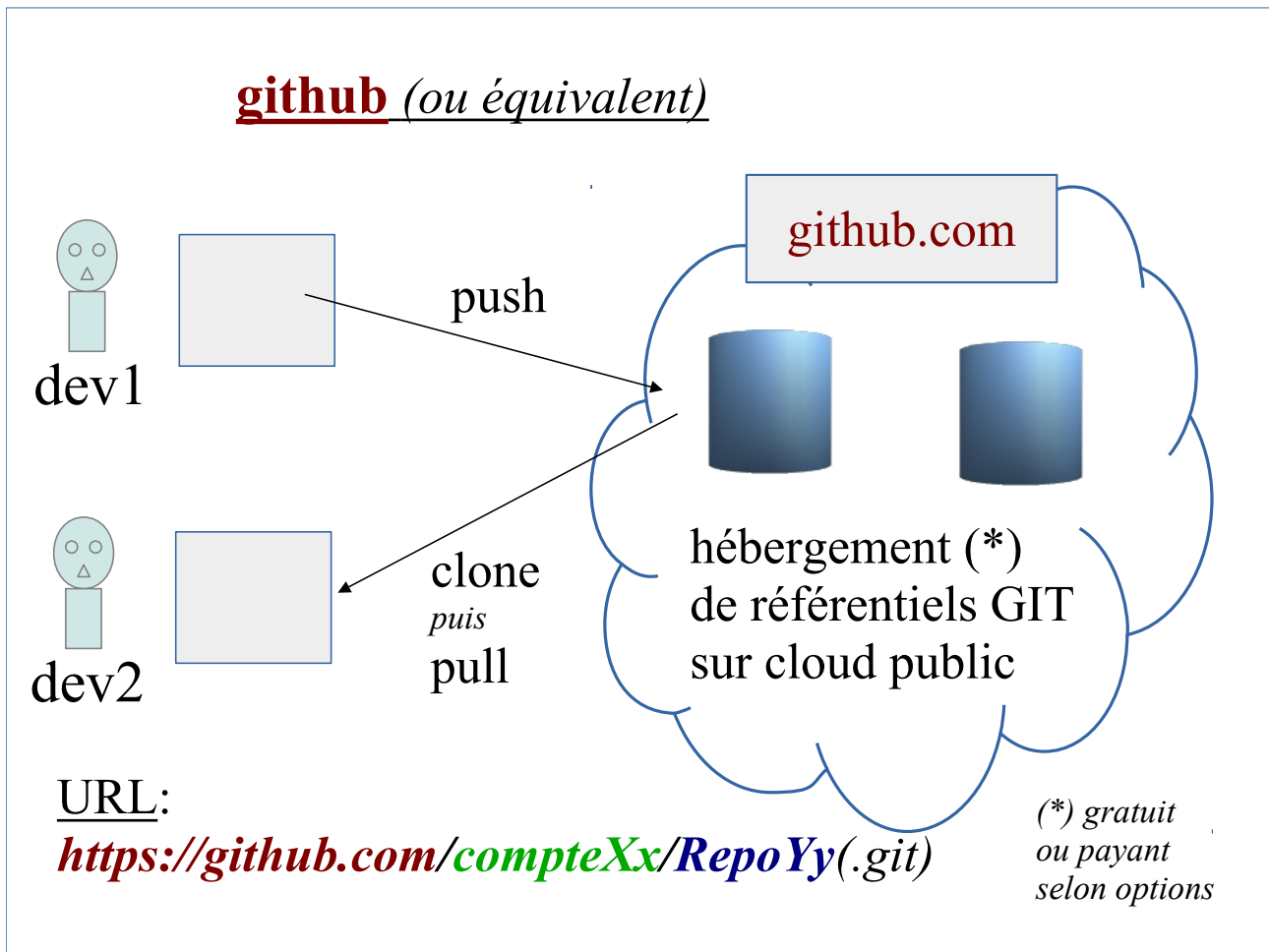
*Le magazine "[PC World](#)" nous apprend que "quand on lui a demandé pourquoi il avait appelé son logiciel "git", qui est à peu près l'équivalent de "connard" en argot britannique" Linus Torvalds a répondu "je ne suis qu'un sale égoцентриque, donc j'appelle tous mes projets d'après ma propre personne. D'abord Linux, puis Git."*



## 1.4. Structure décentralisée de Git (vue d'ensemble)



## 1.5. Hébergement de référentiel GIT



Principaux sites d'hébergement de référentiels GIT :

- **github**
- **gitlab**
- **bitbucket**
- ....

## 1.6. Installation et configuration minimum :

- 1) installer "*Git for windows*" ou bien "*Git sur linux*" (via **yum** ou **apt-get** ou autre) .
- 2) **git --help**
- 3) **git config ....**

*En bref, les commandes «**commit**» et «**checkout**» de **GIT** permettent de gérer le référentiel **local** (propre à un certain développeur) et les commandes «**push**» et «**fetch / pull**» de **GIT** permettent d'effectuer des **synchronisations** avec le **référentiel partagé distant** .*

### Configuration locale de GIT

Installation de GIT sous linux (debian/ubuntu) :

```
sudo apt-get install git-core
```

Configuration locale:

```
git config --global user.name "Nom Prénom"
git config --global user.email "poweruser@ici_ou_la.fr"
#...
```

```
# pour voir ce qui est configuré :
git config --list
```

NB: certaines commandes de git peuvent quelquefois démarrer un éditeur de fichier en mode texte de façon à pouvoir spécifier certains choix (ex : parties d'un fichier à sélectionner).

Il est possible de choisir l'éditeur de fichier à utiliser avec git via la variable d'environnement **GIT\_EDITOR** ou bien via l'entrée **core.editor** de la configuration de git :

Exemple pour windows :

```
git config --list
```

```
...
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -nosession -noPlugin
```

Exemple pour linux :

```
git config --global core.editor "/usr/bin/vim"
```

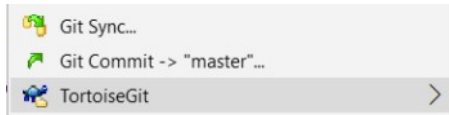
*Attention : vim (vi improved) n'est pas simple/intuitif : à n'utiliser que si l'on connaît déjà bien.*

ou

```
git config --global core.editor "/usr/bin/nano"
```

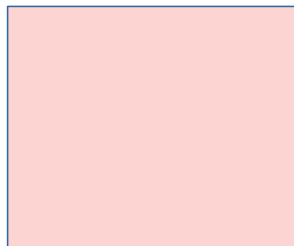
## 1.7. GIT , Tortoise-GIT , intégration IDE et SmartGit, GitKraken

### Utilisation directe ou indirecte de GIT



*Ajoute des menus contextuels à l'explorateur de fichiers*

*Menu "team" et perspective "GIT"*



...



**GIT** (en ligne de commande)

O.S. (linux ou windows ou ...)

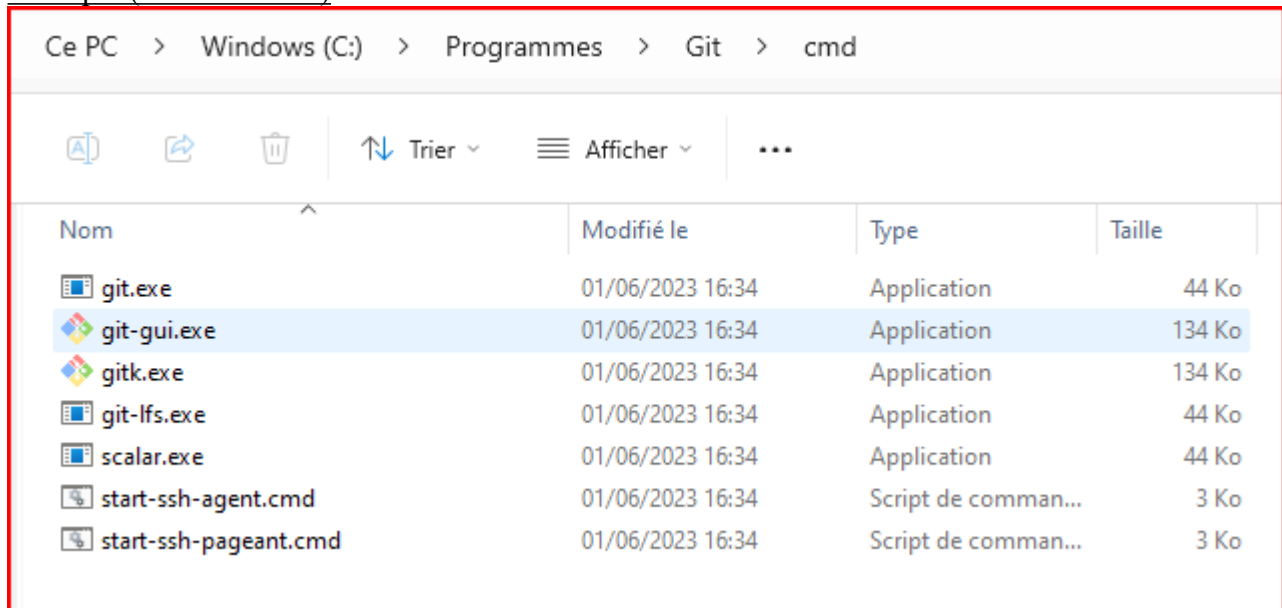
<i>Environnement git</i>	<i>caractéristiques</i>
git en <b>ligne de commande</b>	toujours accessible/utilisable (en dev, en prod , ...)
<b>tortoiseGit</b>	Bien intégré à l'explorateur de fichiers de windows , sous windows seulement (pas sur linux/mac) . Utilisable sur n'importe quel type de projets (java, javascript, python, ...) . Simple et intuitif .
<b>plugin Git pour IDE</b> (eclipse, intelliJ, Visual Studio Code)	Souvent multi-plaforme (windows, linux, mac , ...) , interface graphique au cas par cas (habitudes à prendre)
<b>GitKraken</b>	Environnement graphique évolué multi-plaforme (windows, linux, mac , ...) existant en différentes versions (free pour "projets open source" , ....=
<b>SmartGit</b>	Autre environnement graphique évolué multi-plaforme (windows, linux, mac , ...) avec un mode de licence spécial (gratuit) pour projet purement open-source (à justifier) et avec une période d'évaluation limitée à 30 jours .
<b>Encore plein d'autres "clients GIT" existants</b>	.... autres outils moins connus (moins utilisés) ...

## 1.8. Utilitaires GIT graphiques de bases

Certains utilitaires "GIT" assez basiques en mode graphique sont livrés dans l'installation standard.

**NB :** Selon le système d'exploitation et l'environnement associé (en mode texte ou mode graphique), certains utilitaires graphiques tels que **gitk** et **git-gui** qui peuvent être **automatiquement installés lors de l'installation standard de git** (sous windows ou ailleurs) sont (ou pas) utilisables.

Exemple (sous windows) :



<b>gitk</b>	Pour visualiser graphiquement les logs de git (historique d'un projet) et les différences entre plusieurs commits ou plusieurs branches. <i>bien dans son domaine</i>
<b>git-gui</b>	Interface graphique "git" assez généraliste mais un peu rudimentaire d'un point de vue ergonomie

## 1.9. Apprentissage progressif de GIT

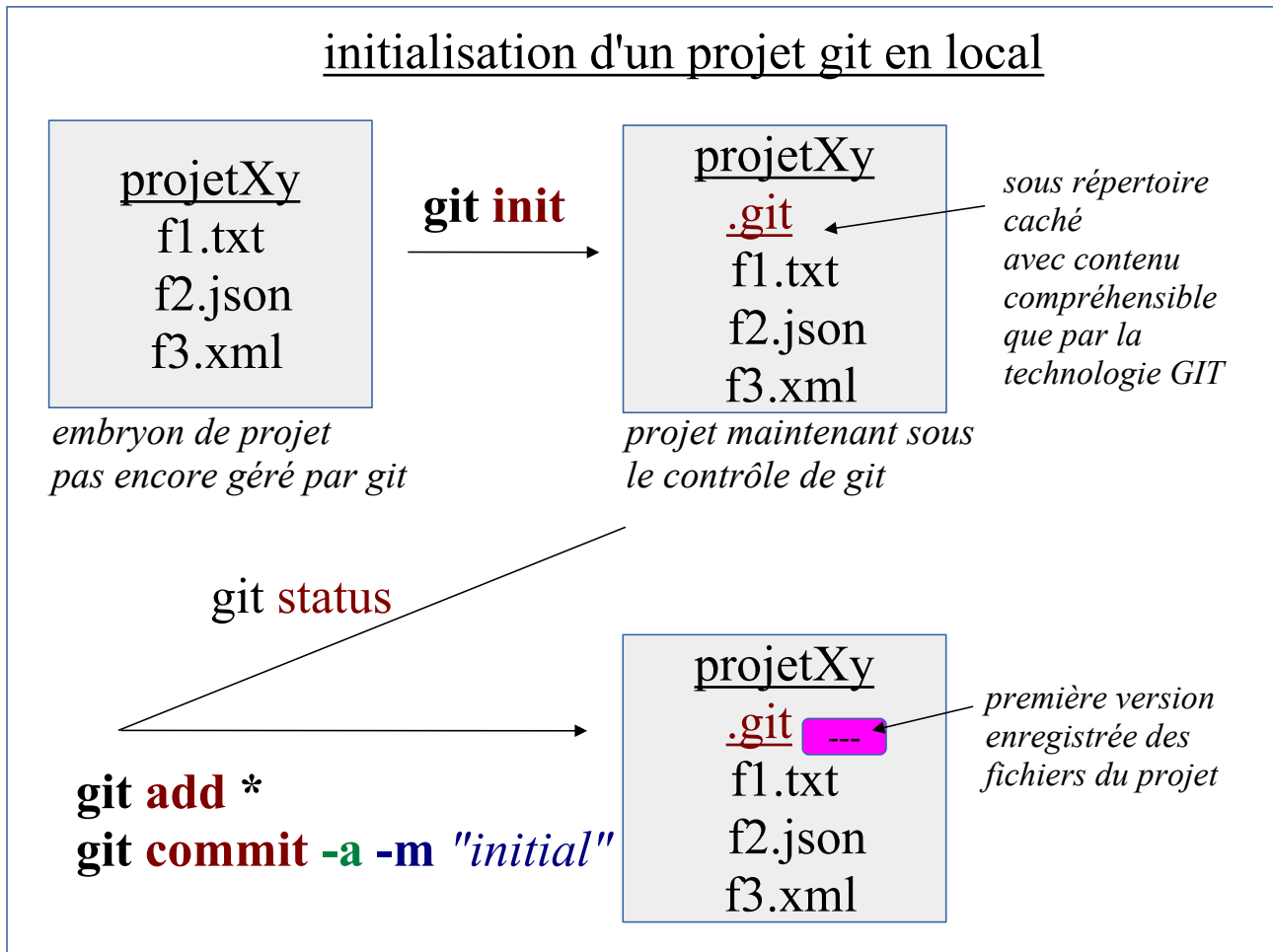
Phase 1	Git (sans branches) en mode local : init , clone , add/commit , status , log , tag , ... en ligne de commande et via quelques outils graphiques (gitk , ...)
Phase 2	Branches de GIT en mode local (branch, checkout , merge , .... , rebase , ...) en ligne de commande et via quelques outils graphiques (gitk , ...)
Phase 3	GIT en mode distant (remote, push , fetch, pull , ....) avec travail en solo puis travail en équipe (réelle ou simulée)
Phase 4	GIT workflow , "pull request" et autres aspects divers
Phase 5	Pratiquer, pratiquer et encore pratiquer

## III - Bases locales de GIT

### 1. Principales commandes de GIT (en mode local)

Commandes GIT (locales)	Utilités
<b>git init</b>	Initialise un référentiel local git (sous répertoire caché « .git » ) au sein d'un projet neuf/originel.
<b>git clone</b> <i>url_referentiel_git</i>	Récupère une copie locale (sous le contrôle de GIT et avec toutes les versions des fichiers) d'un référentiel git existant (souvent distant)
<b>git status</b>  <b>git diff</b> <i>fichier</i>	Affiche la liste des fichiers avec des changements (pas encore enregistrés par un commit) et git diff affiche les détails (lignes en + ou -) dans un certain fichier.
<b>git add</b> <i>liste_de_fichiers</i>	Ajoute un répertoire ou un fichier dans la liste des éléments qui seront pris en charge par git (lors du prochain commit).
<b>git commit</b> <i>-m message [-a]</i>	Enregistre les derniers fichiers modifiés ou ajoutés dans le référentiel git local (ceux préalablement précisés par <i>add</i> et affichés par <i>status</i> ) . si option <i>-a</i> tous les fichiers modifiés (ou supprimés) qui étaient déjà pris en charge par git seront enregistrés
<b>git checkout</b> <i>idCommit (ou branche)</i>	Récupère les (dernières ou ....) versions depuis le référentiel local
<b>git --help</b> <b>git cmde --help</b>	Obtention d'une aide (liste des commandes ou bien aide précise sur une commande)
<b>git log</b>	Affiche l'historique des mises à jour
<b>git branch</b> , <b>git checkout</b> <i>nomBranche</i> , <b>git merge</b>	Travailler (localement et ...) sur des branches ( <i>approfondissement dans prochain chapitre</i> )
<b>git grep</b> <i>texte_a_rechercher</i>	Recherche la liste des fichiers contenant un texte
<b>git tag</b> <i>NomTag IdCommit</i>	Associer un <b>tag</b> parlant(ex: <b>v1.3</b> ) à un id de commit .
<b>git tag -l</b>	Visualiser la liste des tags existants
<b>git checkout tags/NomTag</b>	Récupère la version identifiée par un tag

## 2. initialisation d'un projet GIT (en local)



### Exemples :

#### #initialisation

```
cd projetXy; git init
```

Initialized empty Git repository in C:/temp/demo-git/projetXy/.git/

#### #affichage des éléments non enregistrés

```
cd projetXy ; git status
```

→ affiche:

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

f1.txt

f2.json

f3.xml

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

# add all files in index (staging area) :

```
git add *
git status
```

→affiche :

```
...
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
   new file:   f1.txt
   new file:   f2.json
   new file:   f3.xml
```

#commit files in index :

```
git commit -m "initial"
```

→affiche :

```
[master (root-commit) ce8f4f7] initial
3 files changed, 7 insertions(+)
create mode 100644 f1.txt
create mode 100644 f2.json
create mode 100644 f3.xml
```

#vérification du changement d'état :

```
git status
```

---> affiche:

```
On branch master
nothing to commit, working tree clean
```

#affichage de l' historique:

```
git log --stat
```

---> affiche:

```
commit ce8f4f7245ea03d3314d35c1177400ebccc6b4ef (HEAD -> master)
Author: DidierDefrance <didier@d-defrance.fr>
Date:   Fri Feb 23 13:03:46 2024 +0100

    initial

f1.txt | 1 +
f2.json | 1 +
f3.xml | 5 +++++
3 files changed, 7 insertions(+)
```

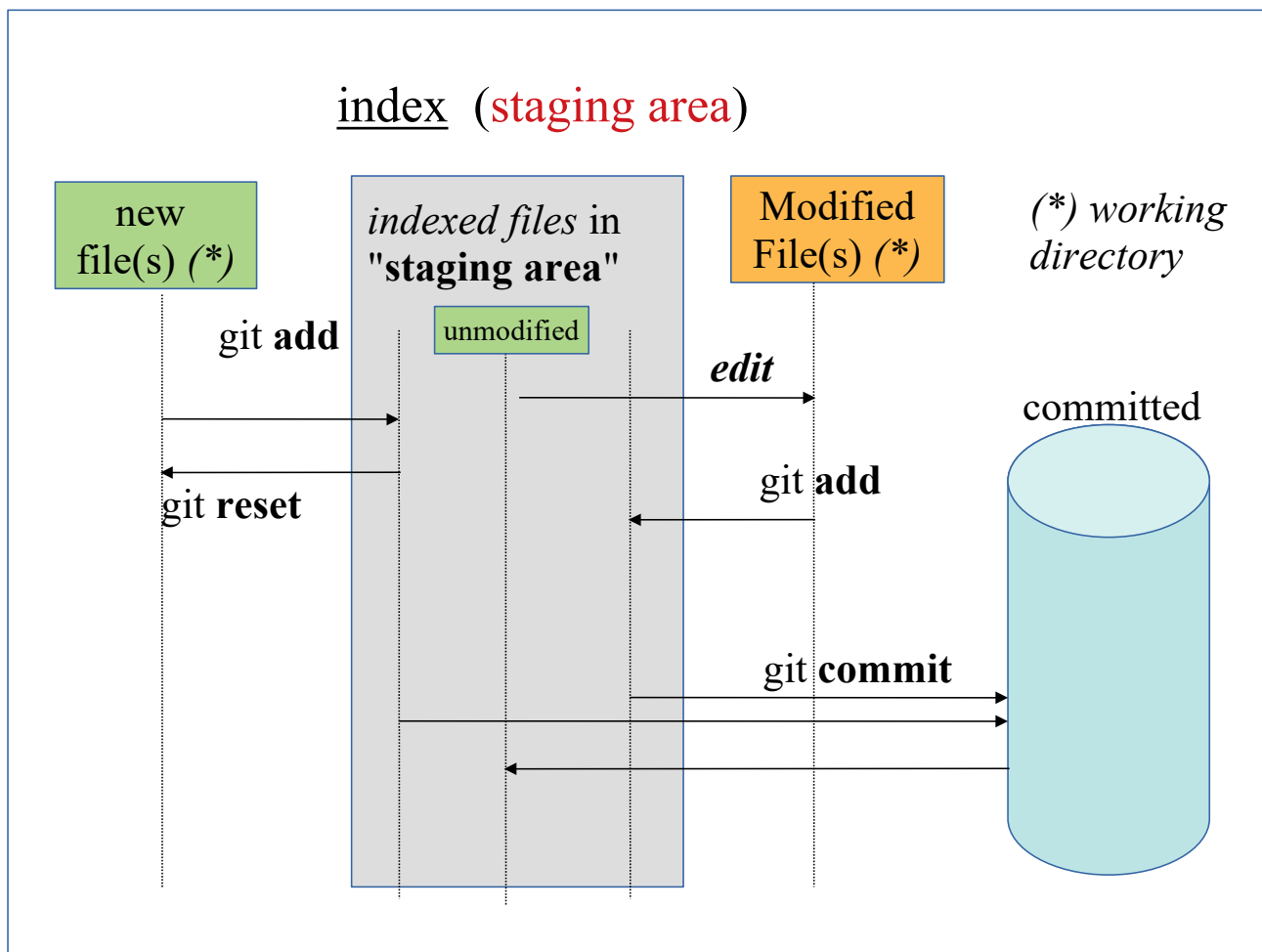


### 3. Index (staging area)

Les mécanismes internes de **git** ne travaillent pas systématiquement sur tous les fichiers du répertoire d'un projet mais sur la sous partie paramétrable appelée "**working tree**" ou "**staging area**" ou "**git index**".

"**git add**" est la principale commande qui permet de modifier le contenu de cet index.

#### 3.1. fichiers indexés par GIT



**NB :**

- Tous les fichiers appartenant à la zone "staging area" seront commités au sein d'un futur "git commit"
- **Un nouveau fichier ne sera ajouté à l'index "staging area" que s'il est explicitement ajouté via la commande "git add"** (ceci permet de contrôler des fichiers que l'on ne veut pas committer, par exemple pas finis, avec bugs, temporaires, ...)
- **Si un fichier préalablement commité est soit modifié, soit supprimé, il pourra alors :**
  - soit être ultérieurement *automatiquement commité* via l'*option -a* de "*git commit*"
  - soit être explicitement ajouter en zone "staging area" via "git add"
  - soit être restauré dans son ancien état via "git restore" ou autre
  - soit être laissé hors zone d'index (et par défaut pas commité lors du prochain commit)
- Si besoin, **git reset fl.txt** permet d'annuler **git add fl.txt**

#### Exemple :

On va considérer que f1.txt (préalablement commité) a été supprimer , que f4.txt est un nouveau fichier du projet et que f2.json (préalablement commité) a été modifié.

#### **git status**

→ affiche :

*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: f1.txt*

*modified: f2.json*

#### **Untracked files:**

*(use "git add <file>..." to include in what will be committed)*

*f4.txt*

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

#### **git commit -a -m "modif et suppression"**

-->affiche :

*[master ab8a7b4] modif et suppression*

*2 files changed, 1 insertion(+), 2 deletions(-)*

*delete mode 100644 f1.txt*

#affichage des éléments non enregistrés

#### **git status**

→ affiche:

*On branch master*

#### **Untracked files:**

*(use "git add <file>..." to include in what will be committed)*

*f4.txt*

...

# add and commit new file :

#### **git add f4.txt**

**git status**

#### **git commit -m "new file f4.txt"**

**git status**

## 3.2. Affichage de l'historique courant

### via git log (en mod texte)

#### **git log**

→affiche (du plus récent au plus ancien) :

```
commit f9ba33b7578fe012239d430be6748a9bfeaf6c44 (HEAD -> master)
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 14:44:23 2024 +0100
```

new file f4.txt

```
commit ab8a7b49ee7c314bc66bb07a7a8f0ad552934c16
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 14:36:54 2024 +0100
```

modif et suppression

```
commit ce8f4f7245ea03d3314d35c1177400ebccc6b4ef
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 13:03:46 2024 +0100
```

initial

### Principales options de git log

- **--stat** pour visualiser des statistiques sur chaque commit (nb d'ajouts , de suppressions , de modifications)

```
git log --stat
```

-->affiche :

...

```
commit ab8a7b49ee7c314bc66bb07a7a8f0ad552934c16
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 14:36:54 2024 +0100
```

modif et suppression

**f1.txt | 1 -**

**f2.json | 2 +-**

**2 files changed, 1 insertion(+), 2 deletions(-)**

- **--oneline** pour afficher chaque commit en abrégé sur une seule ligne

```
git log --online
```

-->affiche :

```
f9ba33b (HEAD -> master) new file f4.txt  
ab8a7b4 modif et suppression  
ce8f4f7 initial
```

```
git log -n 3
```

→ affiche que les 3 dernières révisions

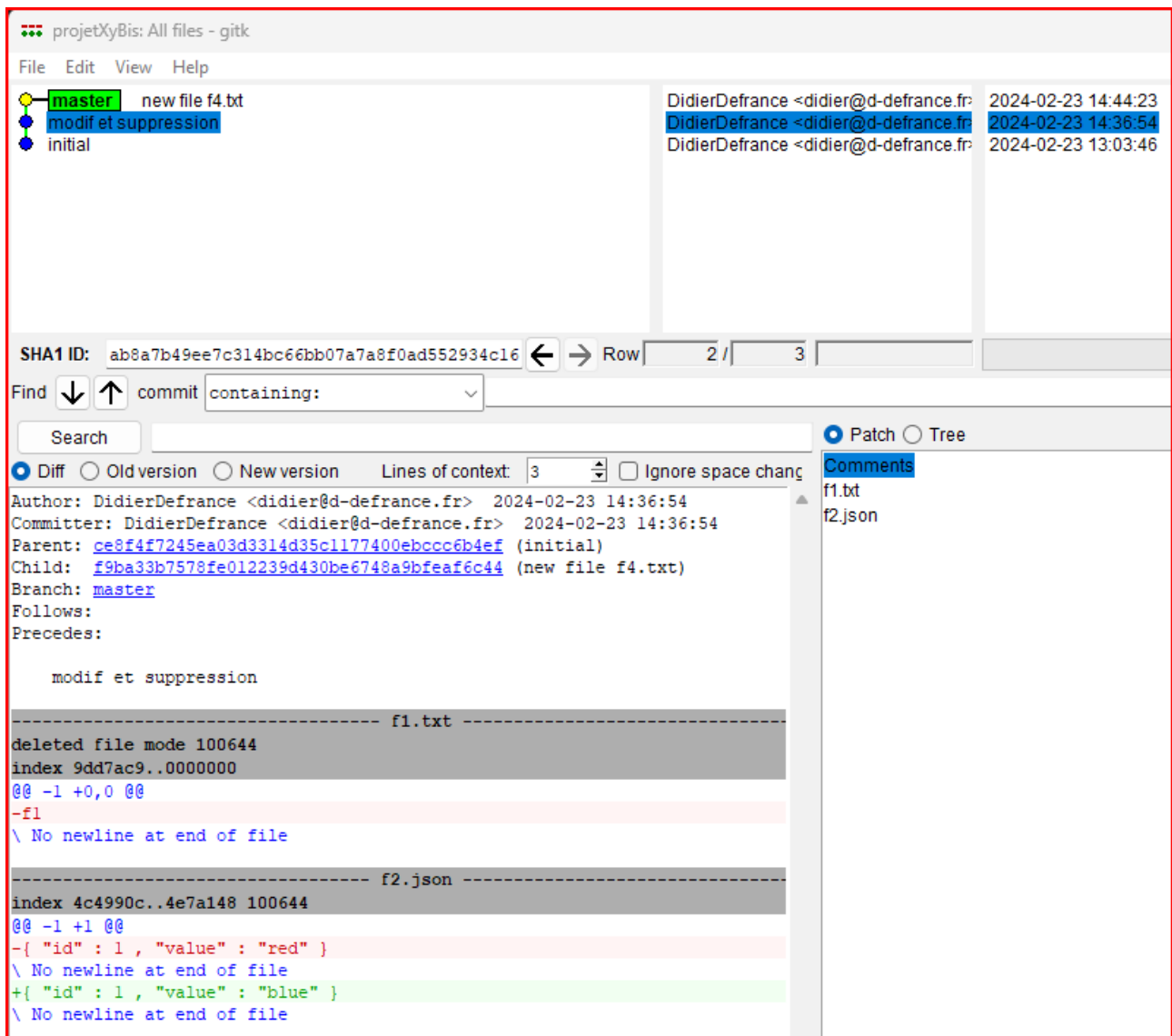
```
git log --all --graph
```

→ affiche toutes les branches (avec des pseudo-graphiques en mode texte)

Exemple :

```
git log -n 6 --graph --all --online
```

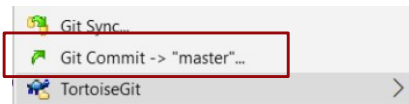
```
*    fb271e2 (HEAD -> main, origin/main, origin/HEAD) Merge branch 'f1' into 'main'  
| \  
| * 4ec08bc (origin/f1) f2Bis  
| * 66adf57 f2Enplus  
* | d1a5ae5 f3EnPlus  
|/  
* 33a9791 s  
* 2e6a5e1 22septembre
```

via gitk (en mode graphique):

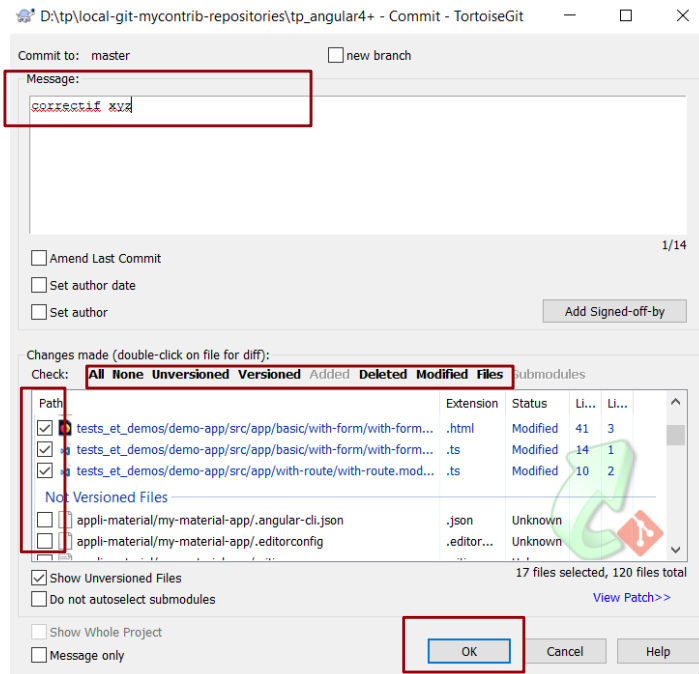
### 3.3. git commit et "show log" via tortoiseGit (sous windows)

#### commit via tortoiseGit

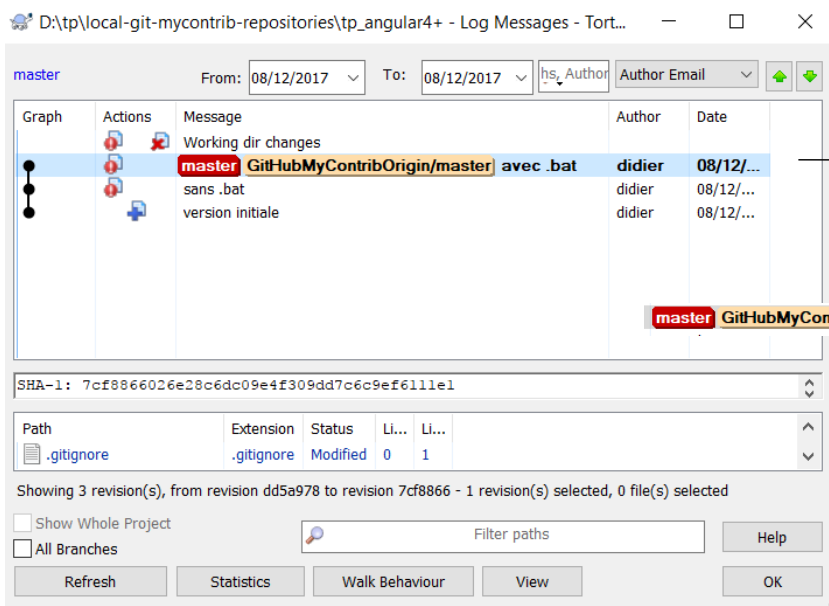
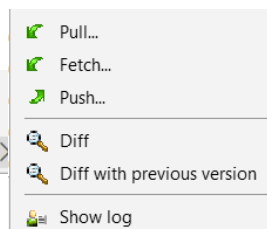
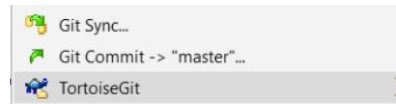
dans l'explorateur de fichiers à partir du répertoire du projet déclencher le menu contextuel **commit** --> ...



*choix des fichiers à "commiter"*



#### show-log et tag via tortoiseGit

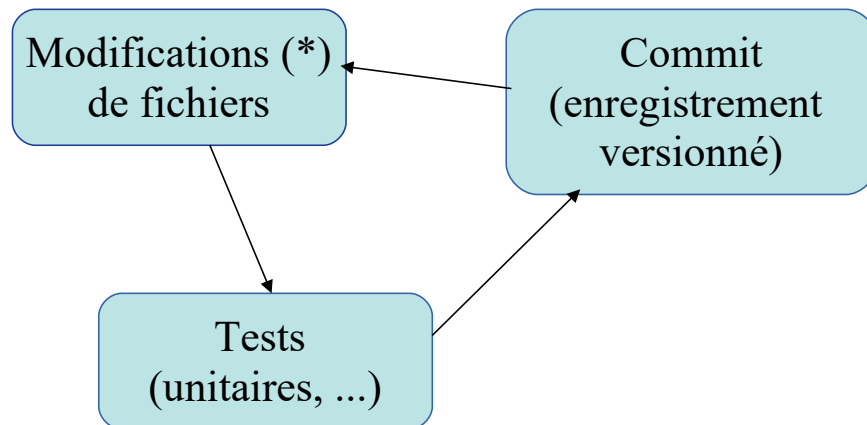


*click droit, create tag at this version*

**master** GitHubMyContribOrigin/master 1.0.RELEASE

### 3.4. cycle des mises à jour

#### Cycle de mises à jour



(\*) Un fichier existant déjà lors du dernier commit et re-modifié depuis fait d'office partie de la "**staging area**" et sera par défaut re-commité. Les commandes `git add ...`, `git rm ...`, `git rm --cached ...` permettent d'ajouter ou retirer des fichiers à commiter ultérieurement avec l'option `-a`.

`git commit --amend -m nouveauMessage` permet si besoin de remplacer le message de commit

#### Quelques cas particuliers :

La commande `git commit --amend` (ou `git commit -a --amend -m "meilleur_message"`) permet d'améliorer le dernier commit effectué en enregistrant les dernières modifications indexées de la "**staging area**". Cela permet d'éviter une multitude de petits commits successifs en cas d'oubli et au final l'historique sera moins long (plus clair). Si l'on ne précise pas l'option `-m`, un éditeur de texte est automatiquement lancé pour que l'on puisse peaufiner le message du dernier commit. Et pour une éventuelle propagation distante immédiate on pourra utiliser `git push --force-with-lease origin main_ou_autre_branche`

Si un fichier a été modifié sur un projet (ex : `f2.json`) et que finalement les modifications sont jugées pas bonnes (inutiles ou néfastes ou autres) on peut alors restaurer le fichier dans son ancien état (celui enregistré au sein dernier commit) via la commande `git restore f2.json`

Si un fichier a été supprimé sur un projet (ex : `f1.txt`) et que finalement cette suppression est jugée pas bonne (inutile ou néfaste ou accidentelle ou autre) on peut alors restaurer le fichier dans son ancien état (annuler la suppression de ce fichier) via la commande `git restore f1.txt`

Rappel : `git reset *` annule `git add *`





## 4. indispensable .gitignore

Le fichier caché **.gitignore** (à placer à la racine d'un référentiel git) est indispensable pour préciser la liste des fichiers à ne pas stocker dans le référentiel git (ex : fichiers temporaires , spécifiques à un IDE , fichiers binaires générés , ....) .

Exemple de fichier ".gitignore" pour java / maven /eclipse :

**.gitignore**

```
target/  
*.class  
*/.settings/  
.settings/*  
.settings  
*.jar  
*.war  
*.ear
```

Exemple de fichier ".gitignore" pour npm/javascript :

**.gitignore**

```
node_modules  
node_modules/*  
dist/*  
dist
```

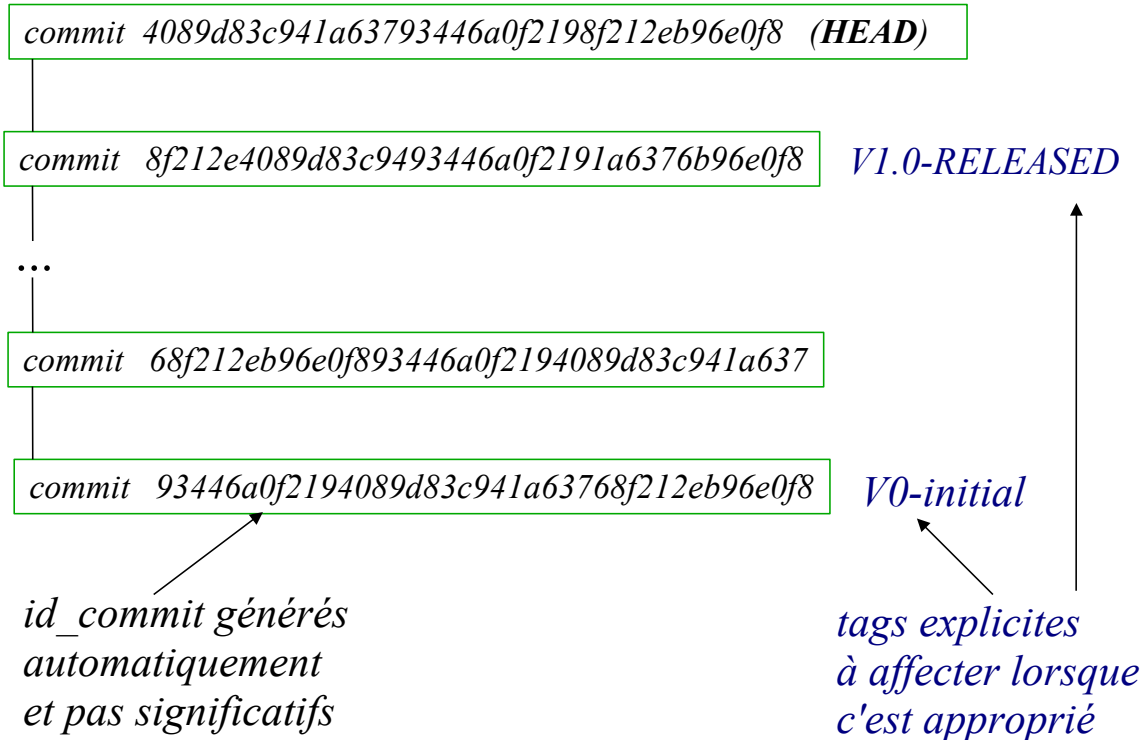
### Gestion des répertoires vides

**NB :**

- **Par défaut git ne sauvegarde (n'enregistre pas) les répertoires vides.**
- **En plaçant un fichier spécial de nom ".gitkeep" dans un répertoire vide , on peut alors faire en sorte que ce répertoire vide soit par défaut enregistré/sauvegardé (suite à `git add *` et `git commit`) .**

## 5. Commit et tags

### commits et tags



Chaque commit est identifié par une clef au format sha-1 (160 bits , 20 octets , 40 caractères en hexadecimal) .

Exemple d'identifiant de commit : `ce8f4f7245ea03d3314d35c1177400ebccc6b4ef`

On ne contrôle pas la valeur d'une clef sha-1 générée/produite . On peut voir cela comme une sorte super-évoluee de UUID ou d'auto-incrémentation .

Chaque commit est heureusement associé à un message obligatoire (à fixer via l'option -m ).

Il est possible (même longtemps après) d'associer un ou plusieurs tags/étiquettes à certains commits importants.

Un tag (étiquette) a une valeur libre que l'on peut contrôler/décider (ex : **V0-initial** , **V1.0-RELEASED** , ....)

L'ajout/association d'un tag à un certain commit peut se faire avec la commande `git tag -a`

Exemple :

```
git tag -a V0-initial ce8f4f7245ea03d3314d35c1177400ebccc6b4ef -m tag_V0-initial
ou bien git tag V0-initial ce8f4f7245ea03d3314d35c1177400ebccc6b4ef (light tag)
```

```
git tag -a V1.0-RELEASED f9ba33b7578fe012239d430be6748a9bfeaf6c44 -m tag_V1
```

NB : l'utilitaire **gitk** permet d'ajouter un tag simplement via un click droit et "create tag".  
Idem avec "tortoiseGit" et la plupart des outils graphiques tels que SmartGit ou GitKraken .

la vérification peut s'effectuer via

**git log** (avec ids de commit complets)

ou bien

**git log --oneline**

(avec ids de commit abrégés (début seulement) )

```
f9ba33b (HEAD -> master, tag: V1.0-RELEASED) new file f4.txt  
ab8a7b4 modif et suppression  
ce8f4f7 (tag: V0-initial) initial
```

La commande **git tag** (par défaut équivalent à **git tag -l** ou **git tag --list**) permet de visualiser la liste des tags existants.

```
git tag -l
```

affiche

```
V0-initial  
V1.0-RELEASED
```

On peut éventuellement supprimé un ancien tag inutile via l'option -d (--delete) :

```
git tag -d t2
```

*Deleted tag 't2' (was f9ba33b)*

## 5.1. Retour sur ancienne version

Il est éventuellement possible de visualiser l'état du projet dans une révision (numéro de commit bien précis ) via la commande **checkout** *numero\_de\_commit\_precis* .

Ceci dit , il ne vaut mieux pas éditer et enregistrer des fichiers juste après car on serait alors en mode "detached HEAD" (détaché d'une tête de branche) ce qui pose plein de problème par la suite .

---> pour effectuer des modifications à un niveau antérieur de l'historique , il faudra idéalement créer une nouvelle branche démarrant au niveau d'un commit\_bien\_precis .

....

Exemples :

```
git checkout ce8f4f7245ea03d3314d35c1177400ebccc6b4ef
```

affiche :

```
Note: switching to 'ce8f4f7245ea03d3314d35c1177400ebccc6b4ef'.
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
  git switch -c <new-branch-name>
...
Or undo this operation with:
  git switch -
```

Retour sur la dernière version du code :

```
git switch -
git log
```

Checkout sur une ancienne version taguée/étiquetée :

```
git checkout tags/V0-initial
```

affiche :

```
Note: switching to 'tags/V0-initial'.
You are in 'detached HEAD' state.
...
```

Pour une simple visualisation d'une ancienne version  
et encore une fois , **git switch -** pour revenir sur la dernière version (dernier commit)

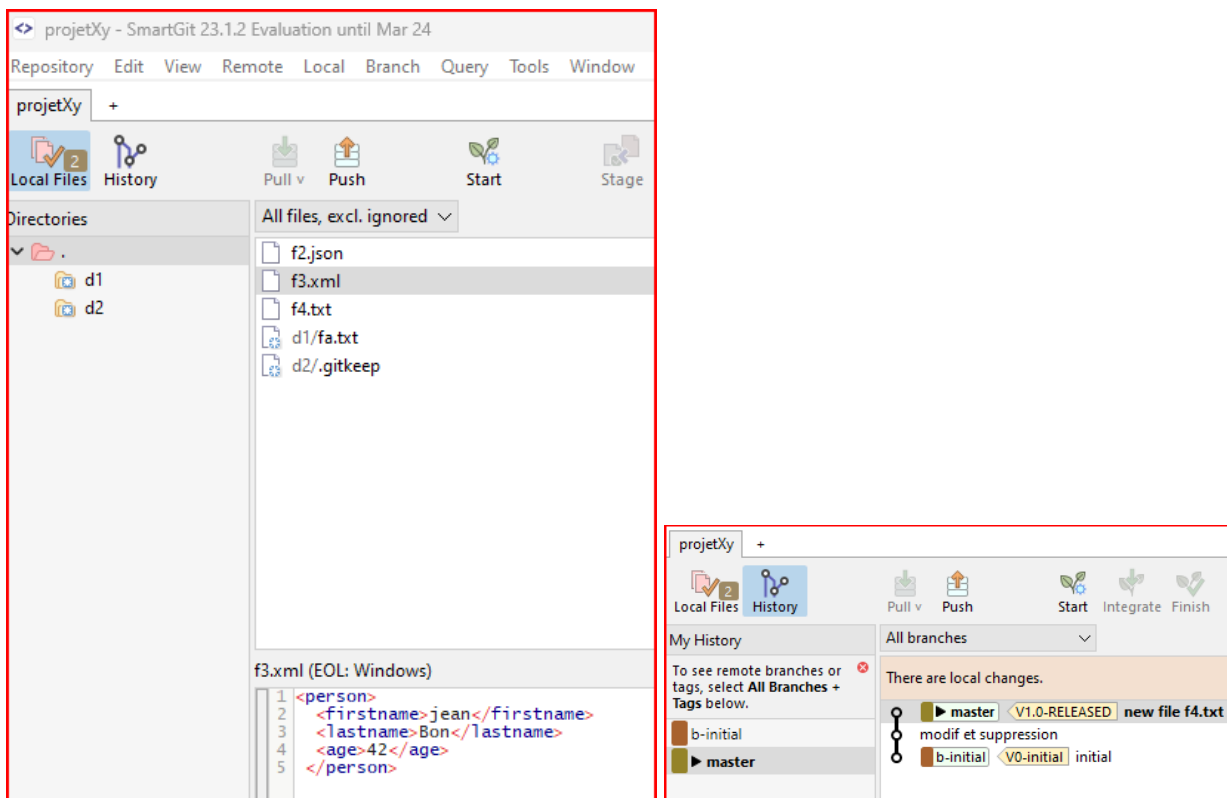
Si l'on souhaite créer une nouvelle branche à partir d'un ancien commit tagué , on pourra utiliser la commande suivante :

```
git checkout tags/<tag> -b <branch>
```

Exemple :

```
git checkout tags/V0-initial -b b-initial
git branch
git checkout master
git checkout b-initial
git checkout master
```

Aperçu avec l'outil "SmartGit" :



## 5.2. Comparaisons et affichages des modifications (git diff)

Après avoir modifié les fichiers existants f4.txt et f5.txt

la commande **git status** permet de visualiser la liste des fichiers qui ont changé :

*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: f4.txt*

*modified: f5.txt*

et la commande **git diff** permet de visualiser les différences (les modifications effectuées depuis le dernier commit par défaut) :

```
diff --git a/f4.txt b/f4.txt
index 3b0729e..351c17f 100644
```

```
--- a/f4.txt
```

```
+++ b/f4.txt
```

```
@@ -1 +1,2 @@
```

```
-f4 v2
```

```
\ No newline at end of file
```

```
+f4 v3
```

```
+ligne2 f4
```

```
\ No newline at end of file
```

```
diff --git a/f5.txt b/f5.txt
```

```
index 968c356..0e64c01 100644
```

```
--- a/f5.txt
```

```
+++ b/f5.txt
```

```
@@ -1 +1,2 @@
```

```
-f5
```

```
\ No newline at end of file
```

```
+f5 v2
```

```
+ligne 2 f5
```

```
\ No newline at end of file
```

Selon les capacités du terminal texte, les lignes supprimées peuvent quelquefois apparaître en rouge et les lignes ajoutées en vert .

NB: la commande **git diff** comporte beaucoup de variantes et d'options.

**git diff f4.txt**

n'affiche que les modifications effectuées sur le fichier f4.txt (et pas sur l'ensemble des fichiers modifiés)

**git diff HEAD~1 -- f4.txt**

affiche les modifications effectuées sur le fichier f4.txt entre l'état actuel du fichier et un ancien commit identifié par son id sha1 ou bien HEAD ou HEAD~1 , HEAD~n (HEAD=tête de branche = dernier commit , HEAD~1 = avant dernier commit , ...)

### 5.3. Affichage d'une ancienne version sans checkout

- 1) via **git log** on repère l'identifiant SHA d'une version (commit) intéressante.
- 2) via la commande **git show num\_commit:path\_to\_file** on affiche la version spécifiée d'un des fichiers du référentiel git

Exemple :

```
git show 0bbd619fbf8645678cfc30e85e091e5d7811dcff:basicHtmlCssJs/notes.txt
```

ou

```
git show HEAD^^:basicHtmlCssJs/notes.txt
```

note 13h36

NB : via une redirection de la sortie de la commande git show, il est possible de stocker l'ancienne version d'un fichier dans un nouveau fichier temporaire à consulter tranquillement :

```
git show 0bbd619fbf8645678cfc30e85e091e5d7811dcff:basicHtmlCssJs/notes.txt > temp_old_notes.txt
more temp_old_notes.txt
del_or_rm temp_old_notes.txt
```

NB: la commande git show peut être déclenchée avec d'autres options/syntaxes .

### 5.4. Git add et commit d'une partie seulement des modifications effectuées sur un fichier

Dans certains cas (assez rares) , on peut avoir effectué plusieurs modifications bien distinctes sur un même fichier (par exemple : ajout de quelques lignes au début et autre ajout de quelques lignes à la fin) .

Via l'option **--patch** (ou **-p**) ou bien l'option **--edit** (ou **-e**) de la commande **git add** on peut indexer seulement certaines sous parties (appelées "hunk") d'un fichier de manière à ne commiter/versionner qu'une partie des modifications apportées sur un fichier .

Mode opératoire :

- 1) nouveau fichier (ex : a.txt) + git add ordinaire + premier git commit ordinaire.  
car la commande git add -p ne semble pas vouloir travailler sur un fichier pas encore commité .
- 2) modifier le fichier a.txt (ajout de ligne au début et autre ajout de ligne à la fin)
- 3)

```
git add -p a.txt
```

et répondre souvent "e" (edit via default editor) à la question posée

ou bien éventuellement directement **git add -e a.txt**

- 4) au sein de l'éditeur supprimer les lignes ajoutées à la fin (et garder celles ajoutées au début) selon les modalités expliquées dans la note finale , enregistrer les modifs et fermer l'éditeur

- 5) **git commit** et **git show ^HEAD:a.txt** pour vérifier l'enregistrement partiel (à comparer avec le fichier a.txt complet).

NB: en cas d'erreur (de sélection des parties du fichier à ultérieurement commiter) on peut annuler

les sélections effectuées via `git status` puis **git restore --staged a.txt**

Exemple du mode edition ("e" de git add -p) :

Soit a.txt déjà enregistré/commité avec ce contenu initial :

```
a
bb
ccc
```

soit a.txt modifié avec le nouveau contenu suivant :

```
nouvelle premiere ligne
a
bb
ccc
nouvelle derniere ligne
```

**git add -p a.txt**

et "e"

.git/addp\_hunk-edit.diff

```
# Manual hunk edit mode -- see bottom for a quick guide.
@@ -1,3 +1,5 @@
+nouvelle premiere ligne
a
bb
-ccc
\ No newline at end of file
+ccc
+nouvelle derniere ligne
\ No newline at end of file
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
```

→ si on ne change rien, toutes les modifications apportées au fichier seront sélectionnées et ultérieurement committées

→ si on supprime certaines parties (par exemple toute la ligne **+nouvelle premiere ligne**) et que l'on enregistre ce fichier de "diff" temporaire interne de git, on aura désélectionner la première partie . Autrement dit, on aura garder que les ajouts en fin de fichier comme partie à ultérieurement commiter.

Après cela, **git status** affiche à peu près cela :

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   a.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:   a.txt
```

et **git commit -m "mon commit partiel de la fin du fichier a.txt"** permet d'effectuer un enregistrement que des modifications sélectionnées (ligne ajoutée à la fin).

Autre solution/alternative :

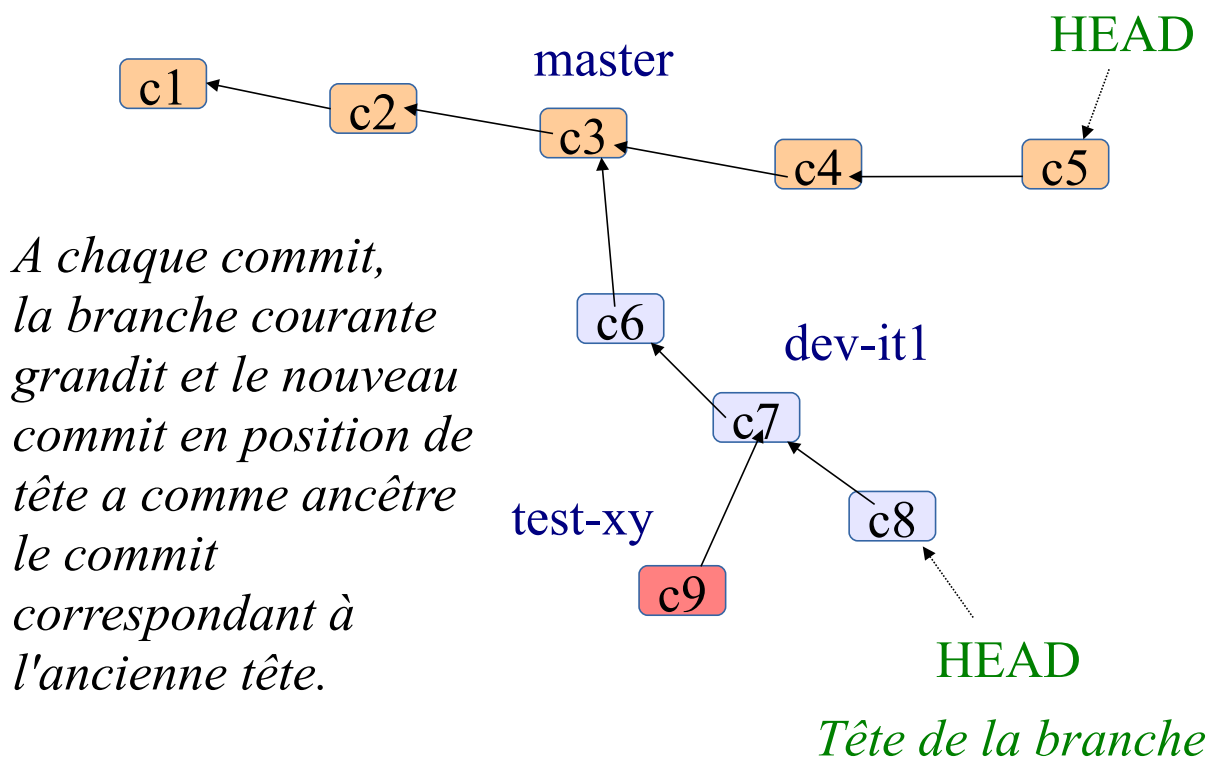
on peut quelquefois utiliser **git gui** et sélectionner des lignes puis les marquer comme "à commiter" via le menu contextuel (click droit) **"stage lines for commit"**



## IV - Gestion des branches / GIT

### 1. Gestion des branches avec GIT

#### Structures des branches de GIT



NB : au sein des anciennes versions de git, la branche principale s'appelait souvent **master** . Certains projets récents basés sur de git préfèrent une branche principale s'appelant **main** .

Le choix du nom de la branche principale est contrôlable via l'option suivante :

```
git config --global init.defaultbranch master
git config --list
git config --global init.defaultbranch main
git config --list
```

## Annulation d'un commit erroné

En local, la commande **git reset --hard *idCommit*** permet de repositionner la tête de la branche courante sur un ancien commit **et toute les modifications apportées par le(s) tout/s dernier(s) commit(s) seront effacées/perdues** .

Si l'option **--soft** est utilisée à la place de **--hard**, les fichiers modifiés ne sont pas effacés du répertoire de travail et on peut les modifier avant d'effectuer un nouveau commit.

La syntaxe spéciale **HEAD~1** correspond à **l'avant dernier commit** et donc **git reset --hard HEAD~1** annule le dernier commit.

HEAD^ ou HEAD~1 : avant dernier commit

HEAD^^ ou HEAD~2 : avant avant dernier commit

Tout projet commence avec une seule branche «**master**» (ou bien "**main**")

## 2. Principales commandes de GIT pour les branches

Commandes GIT (branches)	Utilités
<b>git branch</b>	Affiche la liste des branches et précise la branche courante (*) .
<b>git branch</b> <i>nomNouvelleBranche</i>	Créer une nouvelle branche (qui n'est pas automatiquement la courante)
<b>git checkout</b> <i>nomBrancheExistante</i>	Changement de branche (avec mise à jour « checkout » des fichiers pour refléter le changement de branche) .
<b>git checkout</b> <i>master</i> <b>git merge</b> <i>autreBranche_a_fusionner</i>	Modifie la branche courante (ici «master») en fusionnant le contenu d'une autre branche
<b>git branch -d</b> <i>ancienneBrancheAsupprimer</i>	Supprime une ancienne branche (avec -d : vérification préalable fusion, avec -D : pas de verif , pour forcer la perte d'une branche morte)
<b>git commit -m</b> <i>message [-a]</i>	Faire pousser/grandir la branche courante ou bien entériner certaines résolutions de conflits après un merge
<b>git rebase</b>	Réajustement de l'historique
....	
<b>git log</b> et <b>git status</b>	Affiche de l'historique et de l'état courant
<b>git stash -u</b> et <b>git stash pop</b>	Mémoriser travail en cours/pas fini (à <i>reprendre</i> )

### 2.1. Gestion élémentaire des branches

*#création d'une nouvelle branche b1 (en étant placé sur la dernière version de la branche parente)*

**git branch b1**

*#affichage des branches existantes (\* marque la branche courante)*

**git branch**

affiche :

*b1*

*\* master*

*#création rare d'une nouvelle branche b-original (à partir d'une ancienne version taguée)*

**git checkout tags/V0-initial -b b-initial**

*affichage des branches existantes (\* marque la branche courante)*

**git branch**

affiche :

*\* b-initial*

```
b1
master
```

#changement de branche :

```
git checkout master
git branch
```

affiche :

```
b-initial
b1
*master
```

#changement de branche :

```
git checkout b1
```

#visualisation :

```
git branch
```

affiche :

```
b-initial
* b1
master
```

#modification de f4.txt et création d'un nouveau fichier f5.txt

#puis ajout/commit pour faire pousser la branche b1

```
git status
git add *
git status
git commit -a -m "modif f4 plus f5.txt"
```

affichage de l'historique (avec la branche b1) :

```
git log
```

affiche :

```
commit fladb1feb9e1a088a71ef71c617b141159e680b0 (HEAD -> b1)
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 18:29:28 2024 +0100
    modif f4 plus f5.txt

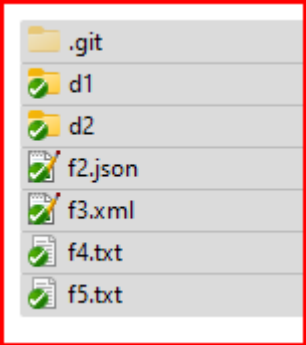
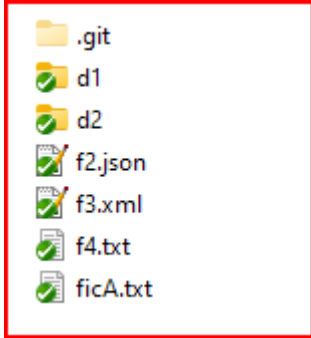
commit 3e34752dbedcf7ea3c08216e0a146a80fa96f060 (master)
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 17:58:58 2024 +0100
    with _sub_directories
```

retour sur branche master , ajout de ficA.txt et add , commit :

```
git checkout master
#...ajout de ficA.txt
```

```
git add ficA.txt
git commit -m "new ficA.txt"
```

Contenu du répertoire du projetXy selon la branche courante

Après "git checkout b1" (avec <i>f5.txt</i> )	Après "git checkout master" (avec <i>ficA.txt</i> )
	

*#Evolution de la branche master en fusionnant le contenu de la branche b1 :*

```
git checkout master
git merge b1
git log
```

→ affiche :

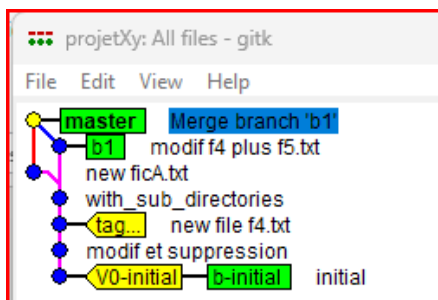
```
commit ebbc002dd7e8357e2423c9571de6b58e0ca36aca (HEAD -> master)
Merge: 27c811a fladb1f
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 18:45:03 2024 +0100
    Merge branch 'b1'

commit 27c811a17d8d9fccb1f31d59adb438fca53838ab
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 18:33:13 2024 +0100
    new ficA.txt

commit fladb1feb9e1a088a71ef71c617b141159e680b0 (b1)
Author: DidierDefrance <didier@d-defrance.fr>
Date: Fri Feb 23 18:29:28 2024 +0100
    modif f4 plus f5.txt
...
```

*Visualisation graphique via gitk :*

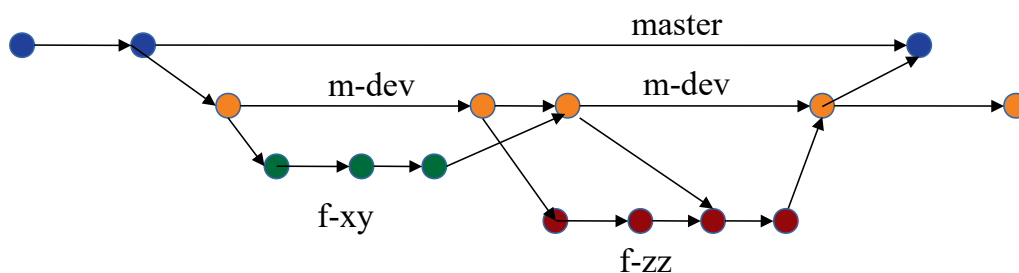
```
gitk
```



### 3. Bonnes pratiques dans la gestion des branches

#### Bonnes pratiques élémentaires (branches GIT)

- ne pas directement programmer sur la branche "master" (à considérer comme la branche stable à déployer en production) mais sur une branche de développement .
- Un certain développeur peut éventuellement créer une sous branche d'ajout de fonctionnalité purement locale (sans push) pour expérimenter et tester une extension, effectuer un merge local et ultérieurement effectuer un push sur la branche parente/commune de développement .
- Attention à bien organiser (en équipe) la gestion des branches distantes (pour éviter pagaille et gros problèmes)

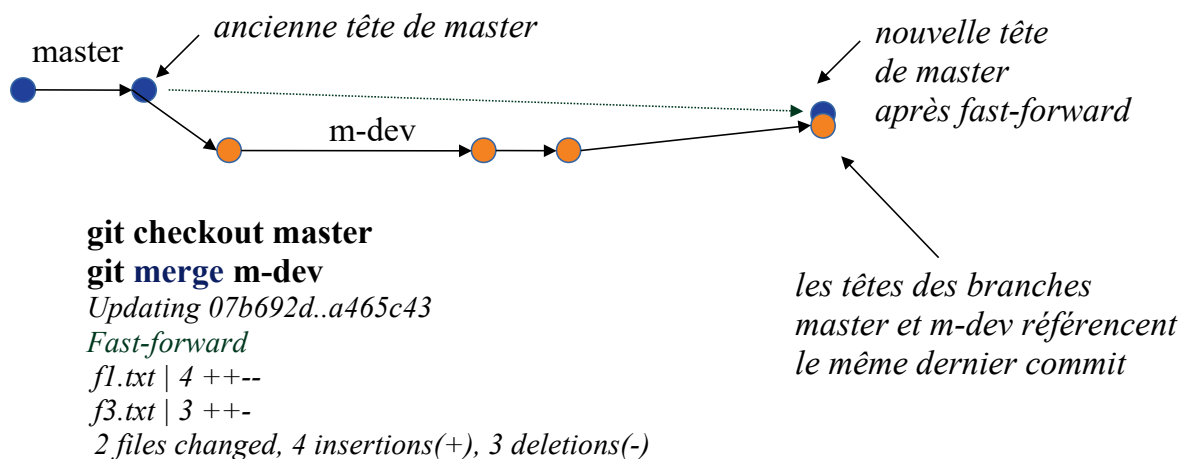


## 4. Merge

### 4.1. merge rapide (fast-forward)

#### Fast-forward (simple merge)

- Lorsque la branche à fusionner comporte quelques commits et que la branche actuelle (réceptrice de la fusion) ne comporte aucun autre commit depuis l'origine de la branche à fusionner, GIT peut effectuer un merge extrêmement rapide et simplifié appelé "fast-forward".
- Ce "fast-forward" consiste à actualiser la référence de la tête de la branche réceptrice pour qu'elle coïncide avec la tête de la branche à fusionner.

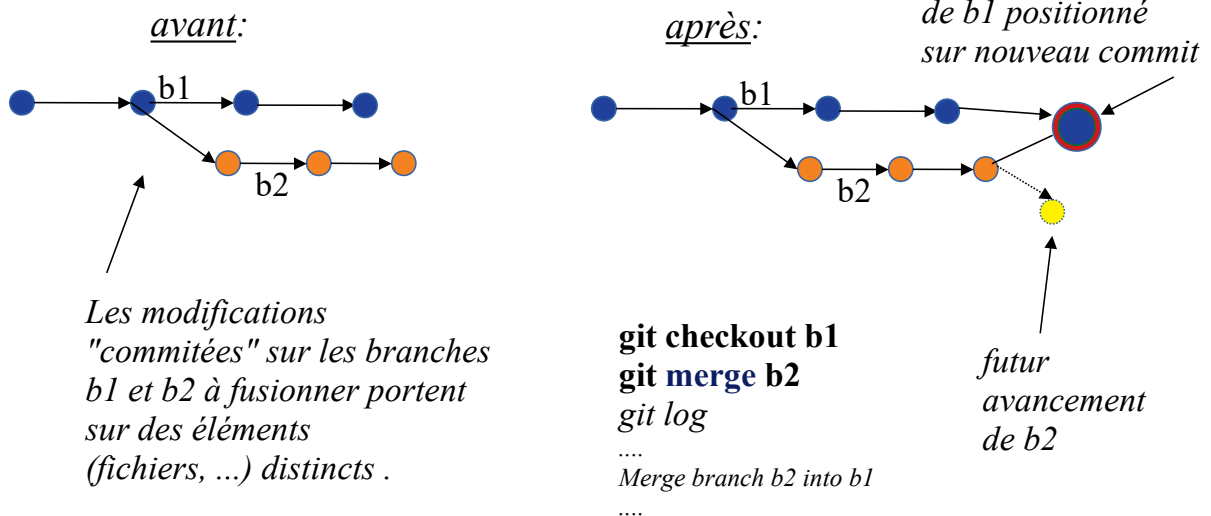


Mis à part ce cas très particulier, les autres sortes de "merge" conduiront à la création d'un nouveau commit de fusion (avec 2 ancêtres).

## 4.2. merge sans conflits

### Merge automatique (sans conflit)

- Lorsque les 2 branches à fusionner comportent quelques commits associés à des modifications/ajouts/suppressions de fichiers différents, GIT peut alors effectuer un merge automatique.
- La branche courante avance et sa nouvelle tête référence **un nouveau commit de fusion** (avec 2 ancêtres).



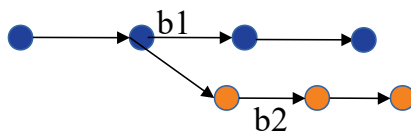
**NB :** Git est capable de gérer automatiquement un merge dans le cadre de modifications (de type "remplacements") apportées sur des lignes différentes d'un même fichier. Ceci n'étant qu'une heuristique, il convient de relancer certains tests unitaires (ou de relire certains fichiers) pour s'assurer que la fusion automatique de git a bien fonctionné.



### 4.3. merge avec conflits

#### Merge avec **conflit** à résoudre

- Lorsque les 2 branches à fusionner comportent toutes les 2 quelques commits associés à des modifications/ajouts/suppressions qui portent sur des fichiers communs , GIT ne peut alors pas décider de ce qu'il faut garder et les conflits doivent être résolus de manière interactive .
- via "git status" on peut connaître la liste des fichiers en conflit lors du merge
- soit le merge doit être annulé via "**git merge --abort**"  
soit **les conflits doivent être résolus** (éditions, add , commit)  
pour que le "merge" puisse aboutir .



← Les modifications  
"commitées" sur les branches  
b1 et b2 à fusionner portent  
sur quelques éléments  
(fichiers, ...) communs .

#### **git status (après merge)**

*On branch m-dev*

*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: fl.txt*

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

Résolution de **conflit** (merge git)

début "git merge" avec conflits  
et "git status"

*f1.txt , ...*

```

<<<<<<< HEAD
f1 v2
aaa
=====
DEBUT
f1 v2
ABC
DEF
>>>>>> master
  
```

*éditions*  
*et tests*

*nouveaux contenus f1.txt , ...*

```

DEBUT
f1 v2
aaa bbb ccc
DEF
  
```

**git commit -m "fin merge xyz"**

**git add f1.txt ...**

**git add f1.txt**

>**git status**

*On branch m-dev*

*All conflicts fixed but you are still merging.*

*(use "git commit" to conclude merge)*

*Changes to be committed:*

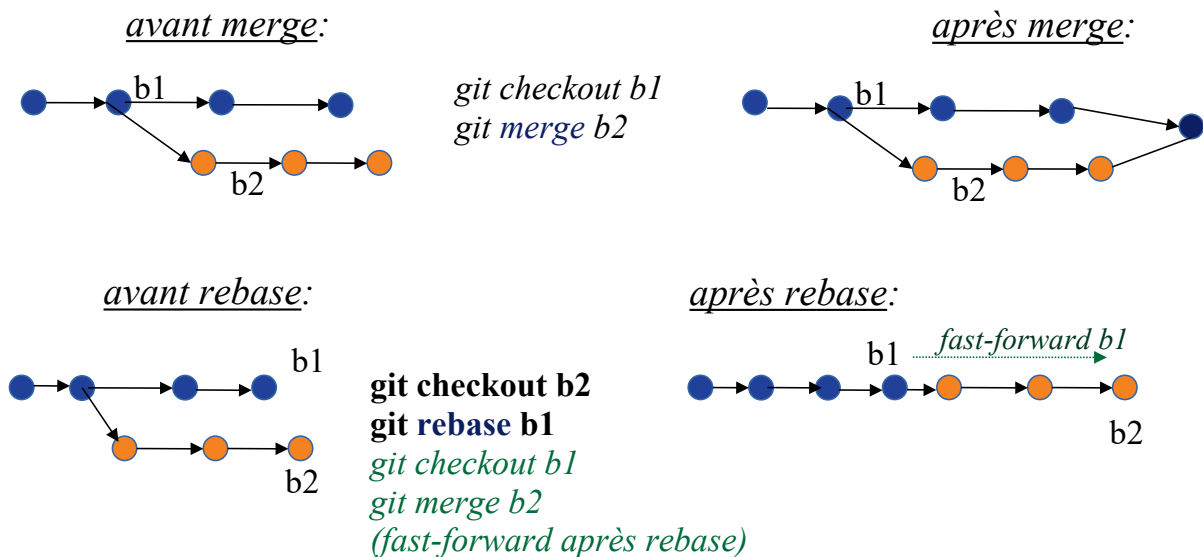
*modified: f1.txt*

**git commit -m "fin merge"**

## 5. Git rebase (souvent sur branche privée)

### git rebase

- La commande "git rebase" permet de reprendre toutes les modifications qui ont été validées sur une branche et de les rejouer sur une autre branche
- Ceci permet d'obtenir un historique linéaire (sans le nouveau commit de fusion d'un merge ordinaire)



**NB** : étant placé sur une branche fille , on déclenche

**git rebase branche\_mere\_avec\_nouvelle\_tete**

pour rebaser la branche fille vers la nouvelle tête de la branche mère

et ceci a souvent pour effet d'intégrer des modifications apportées par d'autres développeurs.

Si lors du "rebase" certains conflits sont détectés , l'opération de "rebase" est alors temporairement stoppée. On peut alors :

- soit annuler le "rebase" via "**git rebase --abort**"
  - soit résoudre le conflit de la même manière que pour un merge (édition, add ...)
- puis déclencher "**git rebase --continue**"

**NB** : en interne la commande **git rebase** va tenter d'avancer de commit en commit (et si conflits , ceux-ci sont gérés au fur et à mesure , git rebase --continue pour passer au commit(s) suivant(s))

Un "rebase" est assez déconseillé sur une branche publique partagée en lecture/écriture car les réorganisations locales pourront une fois propagées mettre la pagaille chez les autres développeurs d'une équipe . *Un git rebase est toutefois envisageable sur une branche publiée/poussée si les développeurs respectent des règles très strictes sur son réaménagement.*



## 6. Git cherry-pick

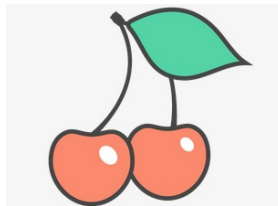
### git cherry-pick (pour cas exceptionnel)

La commande "git **cherry-pick**" permet de récupérer des modifications d'un commit spécifique (d'une autre branche ou commit perdu/détaché) de manière à l'intégrer dans la branche courante .

Cette opération exceptionnelle peut s'avérer pratique pour réintégrer un "hotfix" (correction de bug en urgence) ou pour corriger certains problèmes.



*refCsha* est ici une référence sha sur le commit rouge de la branche b2



### 6.1. Comparaisons de deux branches via git diff

**git diff b1 --**

affiche toutes les différences entre la branche courante et la branche b1

**git diff b1 -- f4.txt**

affiche toutes les différences sur le fichier f4.txt entre la branche courante et la branche b1

```
...
@@ -1,2 @@
-f4 v2
\ No newline at end of file
+f4 v3
+ligne2 f4
\ No newline at end of file
```

**git diff b1 b-initial -- f4.txt**

affiche toutes les différences sur le fichier f4.txt entre la branche b1 et la branche b-initial

## 6.2. correctif (fixup) sur ancien commit

Lorsque l'on s'aperçoit (longtemps après) que l'on a introduit du code en erreur au sein d'un ancien commit , on peut effectuer une amélioration et effectuer un nouveau commit .

Pour que ce nouveau commit ait automatiquement un message de type

"**fixup!** message\_ancien\_commit" , on peut éventuellement utiliser l'option `--fixup id_ancien_commit` plutôt que `-m "...."` .

Exemple :

**git add \***

**git commit --fixup 31d4ab7ff247bf4357ade83a523f000e3444abd8**

**git log**

commit 05759d6222578d56db9904c8404fc84de006b98b (HEAD -> b1)

Author: DidierDeFrance <didier@d-defrance.fr>

Date: Mon Feb 26 11:42:08 2024 +0100

**fixup! f6\_modif**

commit 8cfb37cd28a6f85d6de96091129249a01576327a

Author: DidierDeFrance <didier@d-defrance.fr>

Date: Mon Feb 26 11:40:20 2024 +0100

f5\_modif

commit **31d4ab7ff247bf4357ade83a523f000e3444abd8**

Author: DidierDeFrance <didier@d-defrance.fr>

Date: Mon Feb 26 11:39:32 2024 +0100

**f6\_modif**

## 6.3. Fusion de plusieurs commits en un seul (squash)

Au sein de git, la notion de "squashing" permet de **transformer une succession de plusieurs petits commits en un seul**.

Cette opération peut soit s'effectuer lors d'un merge très particulier avec une option `--squash` pas simple à maîtriser , soit tranquillement s'effectuer sur une branche de "features" via

"**git rebase -i HEAD~n**" (avec n valant 1 ou 2 ou 3 ou ...) .

NB: cette opération peut souvent s'effectuer de manière très simple et très intuitive avec un outil graphique (plugin pour IDE ou autre).

Exemples :

Soit 3 petits commits venant d'être effectués sur la branche b1 (ajout des lignes aaa ,bbb et ccc sur f6.txt) .

### git log -n 3

permet d'afficher l'historique partiel (les 3 derniers commits)

```
commit 77aab3450ced1af68ad7a6958af70e4d1f460784 (HEAD -> b1)
```

```
Author: DidierDefrance <didier@d-defrance.fr>
```

```
Date: Mon Feb 26 10:20:51 2024 +0100
```

```
f6_ccc
```

```
commit c935b00c2e847f5a98f628abc99fb3c4d9a1d432
```

```
Author: DidierDefrance <didier@d-defrance.fr>
```

```
Date: Mon Feb 26 10:16:08 2024 +0100
```

```
f6_bbb
```

```
commit e380a602164e452df2af0bf720c7730284a2e855
```

```
Author: DidierDefrance <didier@d-defrance.fr>
```

```
Date: Mon Feb 26 10:15:32 2024 +0100
```

```
f6_aaa
```

### git rebase -i HEAD~3

fait apparaître un éditeur (car -i signifie mode interactif) dans lequel il faut remplacer

```
pick abc56s commit_...
```

```
pick abc56s commit_...
```

```
pick abc56s commit_...
```

```
#...
```

par

```
pick abc56s commit_...
```

```
s abc56s commit_...
```

```
s abc56s commit_...
```

```
# s or squash
```

dans un second éditeur on pourra choisir

le message du nouveau commit condensé (3\_en\_1\_ex: f6\_aaa\_bbb\_ccc ).

Vérification de l'opération effectuée :

### git log

```
commit 9549989407fb7e073ee43918dfd24e96cd8c1c60 (HEAD -> b1)
```

```
Author: DidierDefrance <didier@d-defrance.fr>
```

```
Date: Mon Feb 26 10:15:32 2024 +0100
```

```
f6_aaa f6_bbb f6_ccc
```

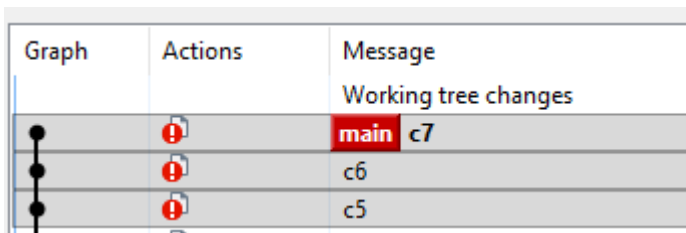
```
...
```

NB : la fonctionnalité "squash" est intéressante mais assez fastidieuse à mettre en place en mode texte lorsque l'on manque d'habitude .

**Il est bien plus naturel d'effectuer un squash avec une interface graphique quand c'est possible.**

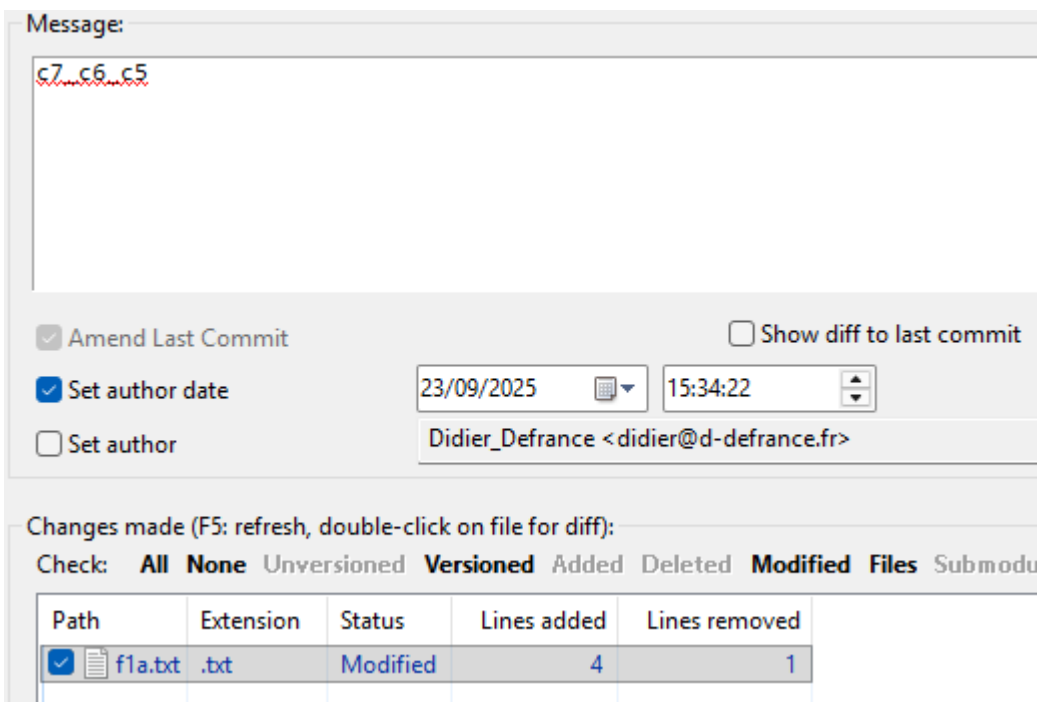
### Squash avec tortoise-git :

tortoise-git/show\_log



**sélection multiple de plusieurs commits** et click droit / "**combine to one commit**"

choix d'un message de commit combiné (ex : c4, c5, c6 ==> c4\_5\_6 )



+ validation du commit fusionné .

*L'outil git-gui trop rudimentaire ne permet malheureusement pas d'effectuer graphiquement un squash.*



## 7. Git stash (enregistrer un travail brouillon à finir)

Dans certains cas , on peut avoir commencé une tâche de moyenne priorité (et avoir modifié certains fichiers du répertoire de travail).

Et l'on a envie de changer de branche pour effectuer une tâche plus prioritaire.

On peut pour cela, lancer une commande spécialisée pour cela :

```
git stash -u
```

Cette commande enregistre dans une sorte de "*pseudo\_commit\_temporaire*" toutes les modifications apportées sur les fichiers du répertoire de travail (*même celles des "untracked-files" si option -u*) et restaure l'état du répertoire de travail dans celui où il était lors du dernier commit enregistré.

```
git checkout otherBranch
```

```
...
```

```
git checkout myBranch
```

```
git stash list
```

→ affiche

```
stash@{0}: WIP on m-dev: 51e3dd5 nomDernierCommit
```

les "stash" sont éventuellement multiples et sont gérés comme une pile.

**git stash show stash@{0}** est une sorte de "diff sur working\_directory et stash@{0}"

Pour restaurer le répertoire de travail dans un état sauvegardé dans un précédent "stash" on peut lancer la commande

```
git stash pop
```

(ou bien **git stash pop stash@{0}** )

D'autres possibilités existent autour de git stash (voir doc officielle et autres articles).

**NB**: dans le cas où l'on voudrait changer de branche sans enregistrer préalablement les modifications effectuées , on obtient alors le message d'erreur suivant :

```
error: Your local changes to the following files would be overwritten by checkout:
```

```
basicJava/src/main/java/tp/MyUtil.java , ...
```

```
Please commit your changes or stash them before you switch branches.
```

```
Aborting
```

Attention :

**git stash -u** (travail pas fini mémorisé et reprenable) est un peu différent par rapport à **git restore \*** (annulation / RAZ définitif depuis dernier commit).

## V - GIT en mode distant (--bare , github, ...)

### 1. Commandes de GIT pour le mode distant

Commandes GIT (mode distant)	Utilités
<b>git init --bare</b>	Initialisation d'un nouveau référentiel vide de type «nu» ou «serveur». (à alimenter par un push depuis un projet originel)
<b>git clone --bare url_repo_existant</b>	Idem mais via un clonage d'un référentiel existant
<b>git clone url_repo_sur_serveur.git</b>	Création d'une copie du projet sur un poste de développement (c'est à ce moment qu'est mémorisée l'url du référentiel « serveur » pour les futurs push et pull)
<b>git pull</b>	Rapatrie les dernières mises à jour du serveur distant (de référence) vers le référentiel local. (NB: <i>git pull</i> revient à déclencher les deux sous commandes <i>git fetch</i> et <i>git merge</i> )
<b>git push</b>	Envoie les dernières mises à jour vers le serveur distant (de référence)  <u>Attention:</u> <i>le push est irréversible et personne ne doit avoir effectuer un push depuis votre dernier pull !</i>
...	

#### Exemples:

#script de création d'un nouveau référentiel GIT (coté serveur) dans /var/scm/git ou ailleurs:

```
mkdir p1.git
cd p1.git
git init --bare
```

#récupération d'une copie du projet sur un poste de développement

```
git clone file:///var/scm/git/p1.git
```

#pull from serv:

```
cd p1
git pull
```

#push to serv:

```
cd p1
git push
```

## 2. Gérer plusieurs référentiels distants

Commandes GIT (mode distant)	Utilités
<b>git remote -v</b>	Affiche la liste des origines distantes (URL des référentiels distant)
<b>git remote add originXy url_repoXy</b>	ajoute une origine distante (alias associé à URL)
<b>git push -u originXy</b>	Effectue un push vers l'origine (upstream) précisé (alias associé à l'URL du référentiel distant).
<b>git push --set-upstream originXy master</b>	push en précisant la branche remote à pister (track) ceci est particulièrement utile pour un push initial vers référentiel distant vide (pas encore initialisé)
...	

Exemples :

*s\_list\_remote\_git\_url.bat*

```
git remote -v
pause
```

*s\_set\_git\_remote\_origin.bat*

```
git remote set-url origin Z:\TP\tp_angular1.git
git remote -v
pause
```

*s\_push\_to\_remote\_origin.bat*

```
git push -u origin master
pause
```

*s\_push\_to\_github.bat*

```
git remote add GitHubMyContribOrigin https://github.com/didier-mycontrib/tp_angular.git
git push -u GitHubMyContribOrigin master
pause
```

*commit\_and\_push.bat*

```
cd /d "%~dp0"
git add *
git commit -a -m "nouvelle version"
git push -u GitHubMyContribOrigin master
pause
```

NB : la gestion des paramètres d'authentification (username/password , token) sera abordée dans un chapitre ultérieur .

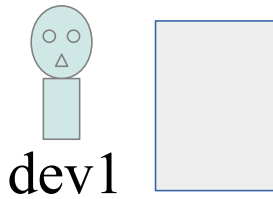
### 3. initialiser git en mode distant

#### initialiser git en mode "remote"

**1** *préparation d'un projet*

*git en mode local (java ou ...)*

*git init , .gitignore , ... , commit*



**3** *préciser url et premier push :*

**git remote add origin url\_repo**

**git push --set-upstream origin master**

**2** *préparation*

*du référentiel*

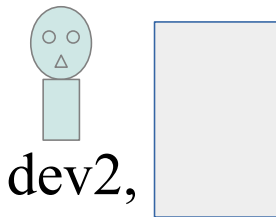
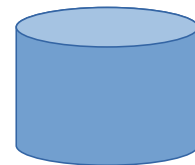
*git distant vide*

*(git init --bare)*

*ou équivalent via*

*console github*

*ou autre*



**4** *clonage(s) (avec "origin" fixée automatiquement) :*

**git clone url\_repo**

...

*Ensuite , git pull et git push de chaque coté .*

## 4. pistage (track) entre branches locales et distantes

### pistage (tracking) de branche

- Lorsqu'une branche locale est paramétrée pour pister (**track**) une branche distante , celles ci pourront ensuite être synchronisées via git push et git pull (et autres).
- Pour initialiser ce processus, il faut la première fois lancer la commande **git push** avec l'option **--set-upstream origin**
- Exemple : **git push --set-upstream origin m-dev**  
*[new branch] m-dev -> m-dev*  
*Branch m-dev set up to track remote branch m-dev from origin.*  
**git branch -vv**  
*\* m-dev 9f0d962 [origin/m-dev] ...*  
*master d41dc86 [origin/master] ...*

## **pistage (tracking) de branche (sur clones)**

- **git branch -a** permet de visualiser toutes les branches dont les branches distantes existantes :

*\* master*

*remotes/origin/HEAD -> origin/master*

*remotes/origin/m-dev*

*remotes/origin/master*

- Au niveau d'un clone secondaire , pour initialiser une branche locale pistant une branche distante existante on peut lancer la commande suivante :

**git checkout -b m-dev origin/m-dev**

*Branch m-dev set up to track remote branch m-dev from origin.*

*Switched to a new branch 'm-dev'*

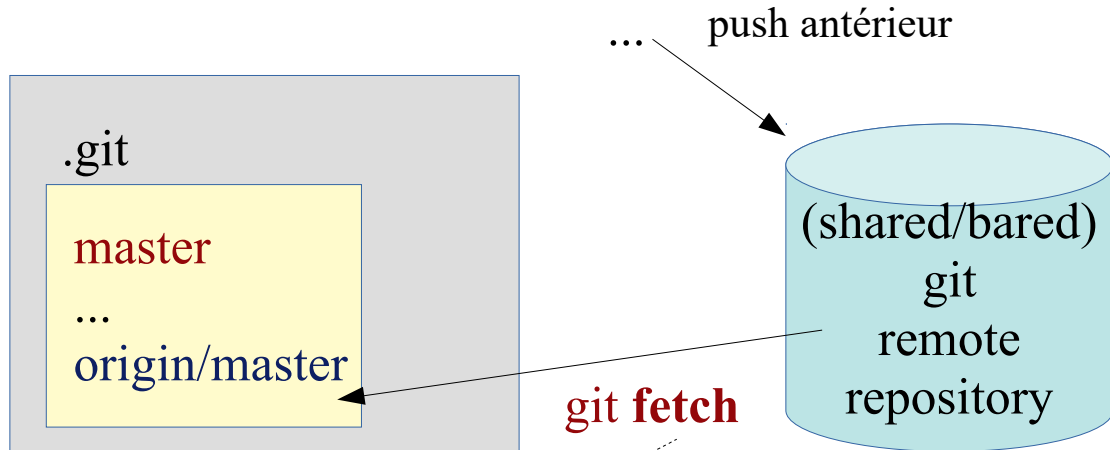
**git branch -vv**

*\* m-dev 439df8e [origin/m-dev] ...*

*master d41dc86 [origin/master] ...*

## 5. Git fetch et git pull

### git fetch



② on peut ensuite , en étant placé sur la branche locale `master` voir les différences via `git branch -vv` et via `git diff origin/master`

① rapatrie (réplique) les mises à jour du référentiel distant dans la branche ***origin/master*** du référentiel local. Rien ne change sur la branche locale "master" ni dans le répertoire de travail (du projet).

## **git pull**

- En étant placé sur la branche master pistant la branche distante origin/master, **git pull** est équivalent à **git fetch** suivit de **git merge origin/master**
- il faut quelquefois résoudre certains conflits
- Les mises à jours distantes sont alors vues / répercutées dans le répertoire de travail (du projet)

### **Remarques importantes :**

- La commande **git push** ne peut quelquefois pas être lancée s'il manque un **git pull** préalable pour rapatrier des modifications effectuées (plus rapidement) par d'autres développeurs.  
Autrement dit , un "git push" ne peut s'effectuer qu'en tête de branche (qui doit être à jour)
- Juste après un **git pull** , on peut se retrouver dans le cadre d'un paquets de conflits à gérer (avec la procédure habituelle : gérer les conflits un par un , *git add ...* et *git commit -m "merge ok ..."*)

NB : il peut arriver qu'à la suite de fausses manipulations, certaines branches soient dans un état incohérent . La commande **git pull** peut par exemple remonter ce genre d'erreurs :

*fatal: refusing to merge unrelated histories*

Dans ce cas, il est possible de débloquent la situation avec une option spéciale :  
**git pull --allow-unrelated-histories**



## 6. Opérations diverses sur branches distantes :

### 6.1. Supprimer une branche distante

Après avoir supprimé une branche locale via `git branch -d` ou `-D` (pour forcer l'opération si pas de merge effectué) on peut demander à supprimer une branche distante via

```
git push origin --delete branche_xyz
```

Commandes conseillées dans la foulée pour supprimer références obsolètes et afficher les branches restantes :

```
git fetch --prune
```

et

```
git branch -r
```

### 6.2. Envoyer des tags vers le référentiel distant

*Par défaut , les tags ne sont pas transférés lors d'un git push ordinaire .*

Si l'on souhaite transférer/envoyer un tag vers le référentiel distant, il faut lancer cette commande :

```
git push origin tag myTagName
```

*# pas souvent recommandé mais possible (push all tags) :*

```
git push --tags
```

Dans le sens inverse, les tags distants sont par défaut récupérés via `git clone` ou `git pull`.

On peut néanmoins demander une réactualisation via

```
git fetch --tags
```

## VI - GitHub , GitLab et sécurité associée

### 1. GitHub , GitLab et Bitbucket

GitHub, GitLab et BitBucket sont les 3 principaux sites d'hébergement de référentiels git (en mode saas) .

	<b>GitHub</b>	<b>GitLab</b>	<b>Bitbucket</b>
<i>Date de création</i>	<b>10 avril 2008</b>	<b>2011</b>	<b>Fin 2008</b> (ancien nom = <i>Stash</i> )
<i>Url principale</i>	<b><a href="https://github.com/">https://github.com/</a></b>	<b><a href="https://gitlab.com/">https://gitlab.com/</a></b>	<b><a href="https://bitbucket.org/">https://bitbucket.org/</a></b>
<i>Propriétaire actuel</i>	<b>Microsoft</b> (depuis 2018)	<b>GitLab Inc.</b>	<b>Atlassian</b> (depuis 2010, même propriétaire que Trello , Jira ,...)
<i>Points forts</i>	<b>Large communauté</b> (plus de 100 millions d'utilisateurs)	<b>Utilitaires en ligne pour intégration continue</b> Gitlab-CI .	<b>Communauté open source non négligeable</b>
<i>Autres caractéristiques</i>	Très connu , simple , ...	GitLab existe aussi en version "on-premise" que l'on peut être installer au sein d'un réseau interne/privé d'entreprise.	Liaisons simples avec d'autres produits de Atlassian (Jira pour le suivi de bugs)

Du côté prix et fonctionnalités , chaque offre est différente.

En général :

- gratuit pour projet "public/open-source"
- payant pour projet "privé" ou bien "avec fonctionnalités étendues"

Pour une éventuelle installation privé (on-premise) , on pourra (entre autres) se baser sur **Gitlab-CE** (gratuit sans support) ou bien Gitlab-enterprise (payant , avec support) .

## 2. Sécurité git (tokens , credential-manager, ...)

### 2.1. comptes utilisateurs, propriétaire et développeurs en équipe

Suite à une inscription sur un site tel que github, gitlab ou bitbucket on obtient un compte utilisateur permettant de créer et gérer des référentiels git distants (hébergés en mode saas).

La personne ou l'administrateur associé à ce compte peut se créer des référentiels git (elle en sera le propriétaire) et pourra partager un accès (plus ou moins restreint) à ce référentiel avec d'autres collaborateurs .

Selon le niveau (*gratuit* ou **payant**) des comptes sur github (ou autres) , on pourra éventuellement **configurer** une **organisation** avec différents membres.

Dans un schéma collaboratif purement open-source/gratuit (avec des contributeurs en mode "particuliers") , on pourra procéder en mixant une ou plusieurs de ces façons :

- chaque collaborateur/contributeur se créer un compte et paramètre si possible des partages sur la plateforme (avec l'avantage de mémoriser/tracer les contributeurs).
- le contributeur principal (à l'origine du projet) peut se créer un paquet de "**developer access tokens**" permettant de futurs accès restreints depuis tous les postes de développement où un de ces tokens sera diffusé.

Exemple :

Le référentiel git dont l'URL est <https://github.com/didier-tp/git-classroom> ne comporte que des éléments d'apprentissage (pas importants) et n'importe quel développeur pourra y apporter sa petite contribution s'il dispose d'un jeton d'accès .

## 2.2. Token pour opérations git distantes sans mot de passe

De manière à accéder en lecture/écriture à un référentiel distant géré de façon récente par GitHub ou bien GitLab on a besoin de générer et enregistrer un "**token développeur**" (alias "**access\_token**") (via la page d'administration d'un compte GitHub ou GitLab) .

Sur GitHub , **sélectionner** (*souvent en haut à droite !*) , l'**icône du profil utilisateur**(compte). Sélectionner "**Settings**" , puis (*souvent en bas à gauche !*) , "**developer settings**"

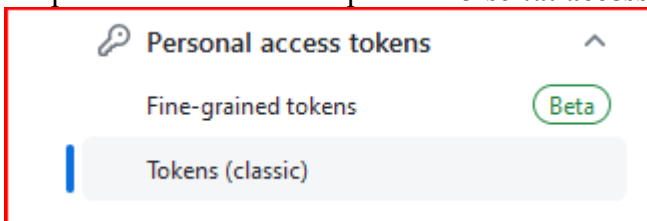
Settings / Developer settings

GitHub Apps

OAuth Apps

Personal access tokens

On peut enfin accéder à la partie "**Personal access tokens**" .



NB : en mode "***fine-grained***" , un token peut être configuré pour un accès restreint (à un certain référentiel , ....) .

## New fine-grained personal access token Beta

Create a fine-grained, repository-scoped token suitable for personal API use and for using Git over HTTPS.

Token name \*

git-classroom-march2024



A unique name for this token. May be visible to resource owners or users with possession of the token.

Expiration \*

Custom...

31 / 03 / 2024



Description

access to git-classroom repository (for didier-tp user) , expired on 2024-03-31 .

What is this token for?

Resource owner



didier-tp

## Repository access

☐ Public Repositories (read-only)

☐ All repositories

This applies to all current *and* future repositories owned by the resource owner.  
Also includes public repositories (read-only).

☒ Only select repositories

Select at least one repository. Max 50 repositories.  
Also includes public repositories (read-only).



Select repositories

Selected 1 repository.



didier-tp/git-classroom



5 permissions for 1 of your repositories	
Actions	Access: Read and write
Contents	Access: Read and write
Discussions	Access: Read and write
Metadata	Access: Read-only
Pull requests	Access: Read and write

0 Account permissions

This token will expire on Sun, Mar 31 2024.

[Generate token](#) [Cancel](#)

Une fois généré et mémorisé dans un fichier temporaire, ce "token développeur" (ressemblant à `gtNVm7wH8P9nhp_t5jIVyZs2AWx6zT2E5p5AYlsF` et valable pour une durée à choisir telle que 30 jours par exemple) sera utilisé en tant qu'élément d'authentification.

Ce "developer access token" d'authentification et d'accès restreint à un référentiel git pourra être utilisé de l'une de ces manières :

- sur demande (au sein d'une boîte de dialogue quelquefois fournie dans certains contextes)
- mémorisé au sein d'un credential-manager pour ensuite être utilisé de manière automatique
- placé en tant que partie additionnelle d'une url au référentiel git

## 2.3. Url git avec token

De manière à se connecter à un référentiel git depuis un IDE tel que eclipse, IntelliJ ou VisualStudioCode on peut utiliser des variantes d'URL GIT comportant un "developer token"

**Format d'une URL de référentiel GIT comportant un token d'authentification :**

`https://username:token_value_string@github.com/repoOwner/repoXyz.git`

Exemples (à adapter)::

`https://didier-tp:ghp_P8.....8U@github.com/didier-tp/git-classroom.git`

**git remote set-url origin** `https://didier-tp:...11AH.....U0@github.com/didier-tp/git-classroom.git`

***NB :** Bien qu'il soit possible de placer username et password (à la place d'un token) dans l'URL d'un référentiel GIT ceci est une très mauvaise pratique d'un point de vue sécurité .*



## 2.4. Url git en mode ssh

Alors que github propose les "access\_tokens" en mode https: au sein de son offre gratuite , le concurrent gitlab propose cette option seulement dans les modes évolués ("Premium" ou "ultimate" pas gratuits).

Par contre , gitlab propose gratuitement des urls en mode ssh dans son offre gratuite . Ce qui fait que beaucoup d'utilisateur de gitlab utilisent des URL en mode ssh .

Exemple :

Soit didier-lab un compte "gitlab" existant (suite à une inscription gratuite) et soit git-classroom.git un référentiel git créé à ce niveau.

Alors l'URL complète menant à ce référentiel sera :

URL HTTPS: `https://gitlab.com/didier-lab/git-classroom.git`

URL SSH: `git@gitlab.com:didier-lab/git-classroom.git`

si besoin de générer et gérer des clefs ssh :

→ documentation au sein de <https://docs.gitlab.com/ee/user/ssh.html>

NB : ed25519 est un peu plus sécurisé que RSA (2048 bits minimum)

```
ssh-keygen -t ed25519 -C "<comment>"
```

```
==> $HOME/.ssh/id_ed25519
      $HOME/.ssh/id_ed25519.pub
```

ou bien

```
ssh-keygen -t rsa -b 2048 -C "<comment>"
```

```
==> $HOME/.ssh/id_rsa
      $HOME/.ssh/id_rsa.pub
```

Exemple:

```
ssh-keygen -t ed25519 -C "my_gitlab_key"
```

Generating public/private ed25519 key pair.

Enter file in which to save the key (C:\Users\d2fde/.ssh/id\_ed25519): `.ssh/my_gitlab_key`

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in `.ssh/my_gitlab_key`

Your public key has been saved in `.ssh/my_gitlab_key.pub`

**NB : la clef privée est censée être gardée de manière confidentielle du coté client**

**la clef publique (.pub) sera placée du coté serveur** (éventuellement via un script et souvent *via la console d'administration*)

Pour placer les clefs publiques ssh au sein d'un compte/profil gitlab il faut :

- 1) se connecter sur <https://gitlab.com> avec son compte (username,password)
- 2) cliquer sur son icône/avatar
- 3) sélectionner le menu "configurer le profil" ou "edit profile"
- 4) sélectionner "SSH Keys"



puis se laisser guider par l'interface graphique (add new key , ...)  
et copier/coller tout le contenu du fichier ....pub

**NB:** on peut si besoin enregistrer et utiliser plusieurs jeux de clef complémentaires (par exemple si accès depuis plusieurs postes ou autres).

dans le répertoire local du projet:

```
git remote set-url origin git@gitlab.com:compteGitlab/projetXyz.git
```

exemple :

```
git remote set-url origin git@gitlab.com:didier-lab/git-classroom.git
```

pour avoir ensuite le droit de faire git push , ... sans mot de passe demandé (car authentification avec clef privé du sous répertoire .ssh de l'utilisateur courant)

Au sein de github, qui accepte également les clefs ssh , le menu est quasiment identique : connexion au profil, **settings** , **ssh and gpg keys** , ....

GitHub semble mieux accepter les clefs avec des noms classiques (ex : *id\_rsa.pub* , ...)

Pour tester une connexion ssh vers gitlab ou github :

ssh -vT [git@github.com](mailto:git@github.com) ou ssh -vT [git@gitlab.com](mailto:git@gitlab.com)

URL github en mode ssh :

```
git@github.com:compteGithub/projetXyz.git
```

exemple: [git@github.com](mailto:git@github.com):didier-tp/spring\_2025.git (à cloner et utiliser)

## 2.5. Configurations de git coté client

### 3 niveaux de configuration :

Niveau	Portée	caractéristiques
<b>--system</b>	Toute une installation (quelque soit l'utilisateur) [gitPath]/etc/gitconfig	Nécessite des droits administrateurs
<b>--global</b>	Pour tous ce qui est utilisé par un même utilisateur/développeur ~/.gitconfig or ~/.config/git/config	
<b>--local</b>	Pour un référentiel particulier [gitrepo]/.git/config	

**git config --show-scope --list**

L'option **--show-scope** permet d'afficher le scope (global,system,local,...)

**NB:** github semble d'abord tenir compte de la valeur du paramètre local (ou global) **user.email** pour déterminer le développeur contributeur ayant effectué un `commit_and_push` .

Si cette valeur de user.email n'est pas associée à un compte github connu alors github va afficher (prendre en compte) la valeur de **user.name** en tant que contributeur.

Exemple :

```
git config --local user.email "d2f.defrance@gmail.com"
```

```
git config --local user.name "DidierTp_dev_z"
```

## 2.6. Credential manager (sur poste de dev. , en local)

Attention: le "git-credential-manager" peut être utile pour mémoriser des "access\_token" en mode https mais ce n'est pas utilisé pour les clefs ssh (ni les éventuelles "ssh key passphrase" ).

Un "credential manager" est un gestionnaire d'éléments d'authentification.

Ceci permet de mémoriser des informations telles que (username + password) ou bien (username + token) de manière à utiliser ultérieurement git sans besoin de se ré-authentifier à chaque fois.

Attention: l'utilisation concrète d'un credential-manager peut grandement varier en fonction des éléments contextuels suivants :

- version du système d'exploitation (windows , linux , ...)
- version de git
- éventuel autre environnement

Sur un PC windows récent, on peut utiliser deux types de "credential manager" :

<b>git-credential-manager (manager)</b>	Multi-plateforme (windows, linux, mac)
<b>wincred (windows credentials)</b>	Spécifique windows (existe depuis longtemps)

Sur une machine windows d'entreprise, on peut également choisir "manager" comme type de "credential.helper" via la commande :

```
git config --global credential.helper manager
```

ou bien

```
git config --system credential.helper manager
```

ou bien

```
git config --global credential.helper wincred
```

ou bien ...

```
git config --local credential.username didier-tp
```

ou bien

```
git config --global credential.username didier-tp
```

puis en lançant la commande suivante pour vérifier :

```
git config --show-scope --list
```

```
git config --global --list
```

NB : comme d'habitude le scope peut valoir **--system** ou **--global** ou **--local** (pour paramétrages spécifiques à un projet/repository)

Options fines :

On peut par exemple configurer GIT pour qu'il retienne le mot de passe saisi durant une certaine période (exemple : 3600 secondes = 1 heure) :

```
git config [ --global ] credential.helper 'cache --timeout=3600'
```

Et si nécessaire (pour ancienne version de git) :

```
git credential-cache exit
```

pour que GIT oublie l'ancien mot de passe et que l'on puisse de ré-authentifier .

ou pour version récente de git :

```
git config --global --unset credential.helper
```

et/ou

```
git config --system --unset credential.helper (avec droits administrateurs)
```

pour désactiver (temporairement ou pas) le "credential.helper" mémorisant les username/password .

***NB: les exemples de configuration ci dessus doit être adaptés au cas par cas (selon version de git , selon OS linux ou windows , selon github ou gitlab, selon l'année , ....)***

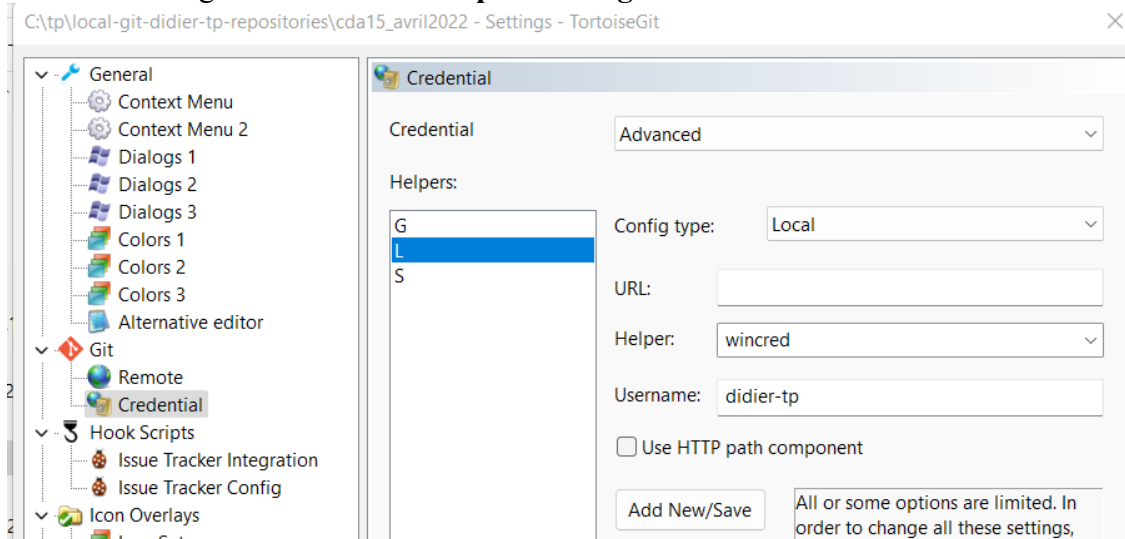
## 2.7. Gestion simplifiée des "credentials" via l'interface graphique de tortoise git

Un "token développeur" pourra être enregistré auprès d'un "**credential.helper**" du poste du développeur (ex : linux ou windows) et sera utilisé en tant qu'élément d'authentification .

Cette opération un peu technique peut éventuellement être effectuée graphiquement via une interface graphique telle que tortoise git ou autre .

Avec tortoise git (git settings , partie git/credentials ,

**add local** config with **credential-helper=manager or wincred** and **username=didier-tp**)



et spécifier le token via un copier/coller lorsque ce sera demandé (lors d'un git push par exemple). Ce token (lié à votre username) sera ensuite automatiquement conservé lors des futures requêtes .

Si le projet est en mode "url ssh" (avec gitlab par exemple),

alors du côté tortoise git , le paramétrage à vérifier ajuster est le chemin menant à ssh.exe

Via le menu **tortoisegit/settings.../Network**

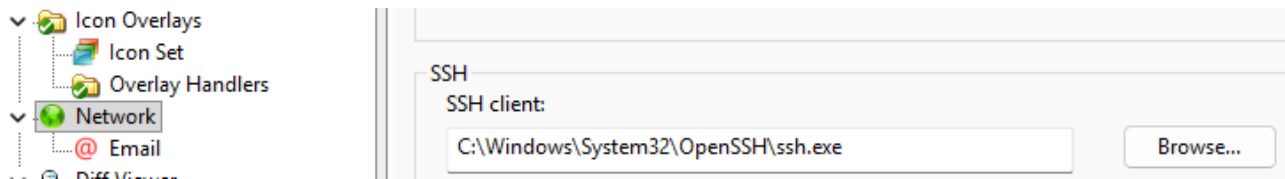
pas `C:\Program Files\TortoiseGit\bin\TortoiseGitLink.exe`

mais

`C:\Windows\System32\OpenSSH\ssh.exe` pour ordinateur windows 11

ou encore

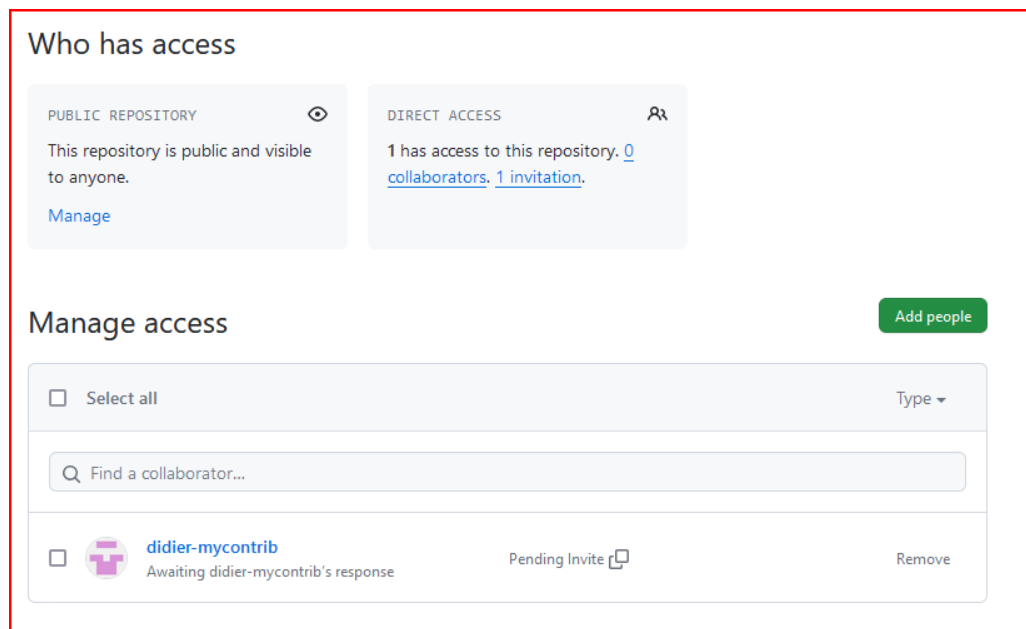
`C:\Users{user}\AppData\Local\Programs\Git\usr\bin\ssh.exe` ou autre



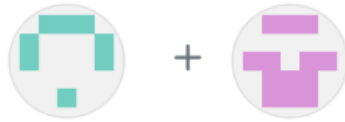
## 2.8. Invitation d'un nouveau collaborateur sur un projet github

1) sur la page d'un référentiel , settings

2) General / Collaborators




Le collaborateur (ayant un autre compte existant) , reçoit un mail et refuse ou accepte l'invitation



[didier-tp](#) invited you to collaborate

Accept invitation

Decline

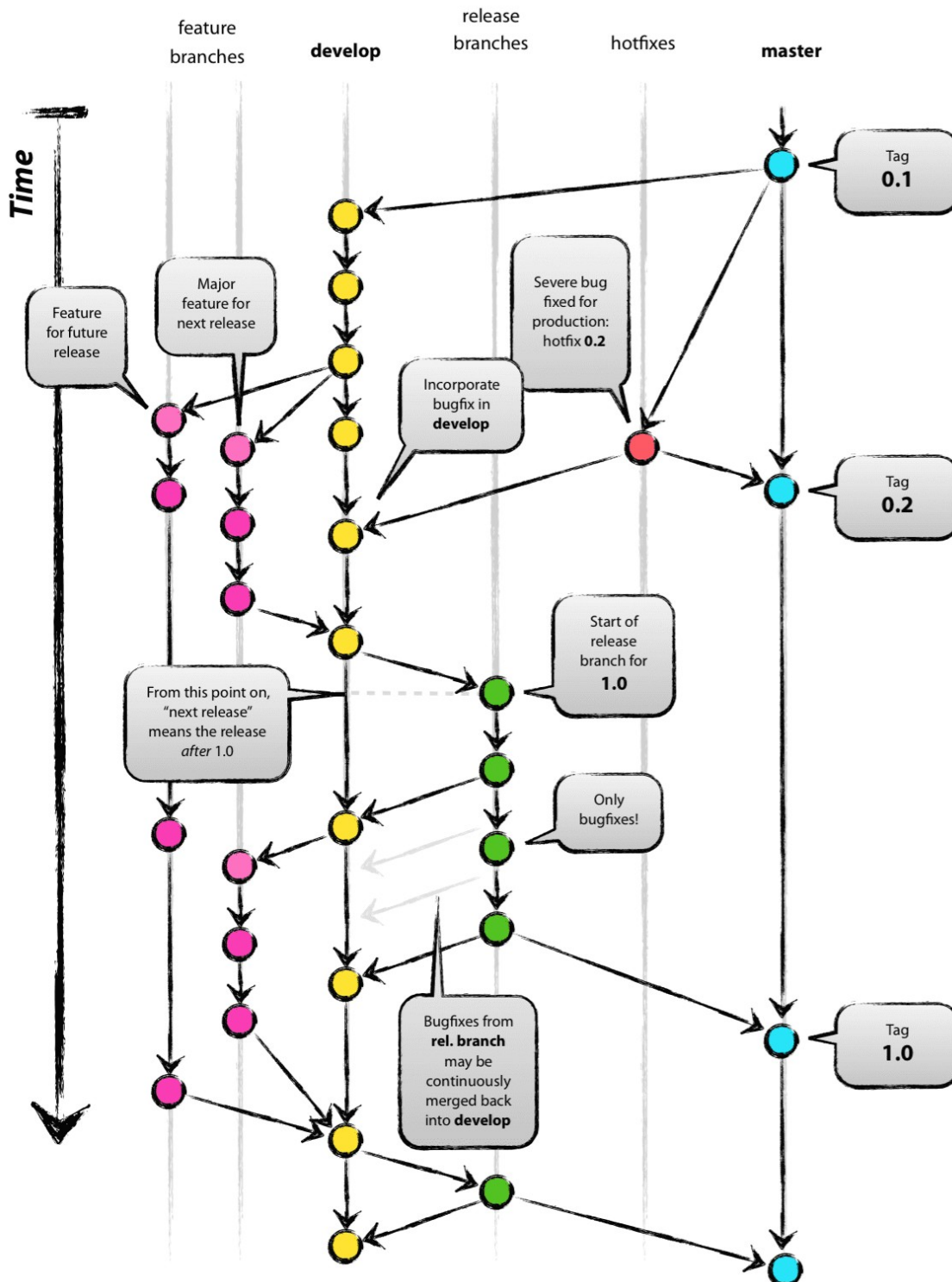
 Owners of git-classroom will be able to see:

- Your public profile information
- [Certain activity](#) within this repository
- Country of request origin
- Your access level for this repository
- Your IP address

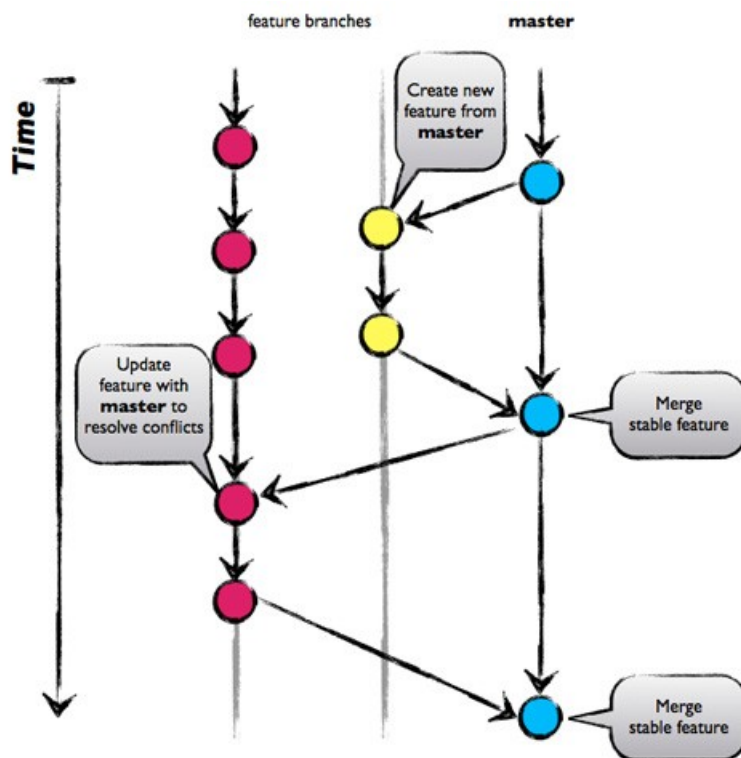
## VII - Git-flow et aspects divers

### 1. Quelques workflows pour GIT

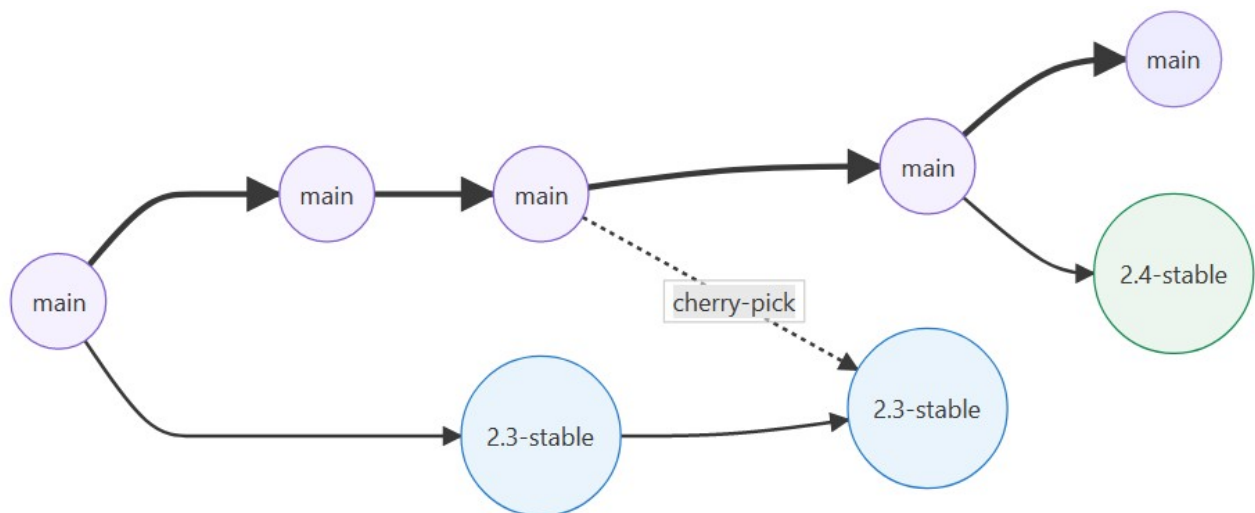
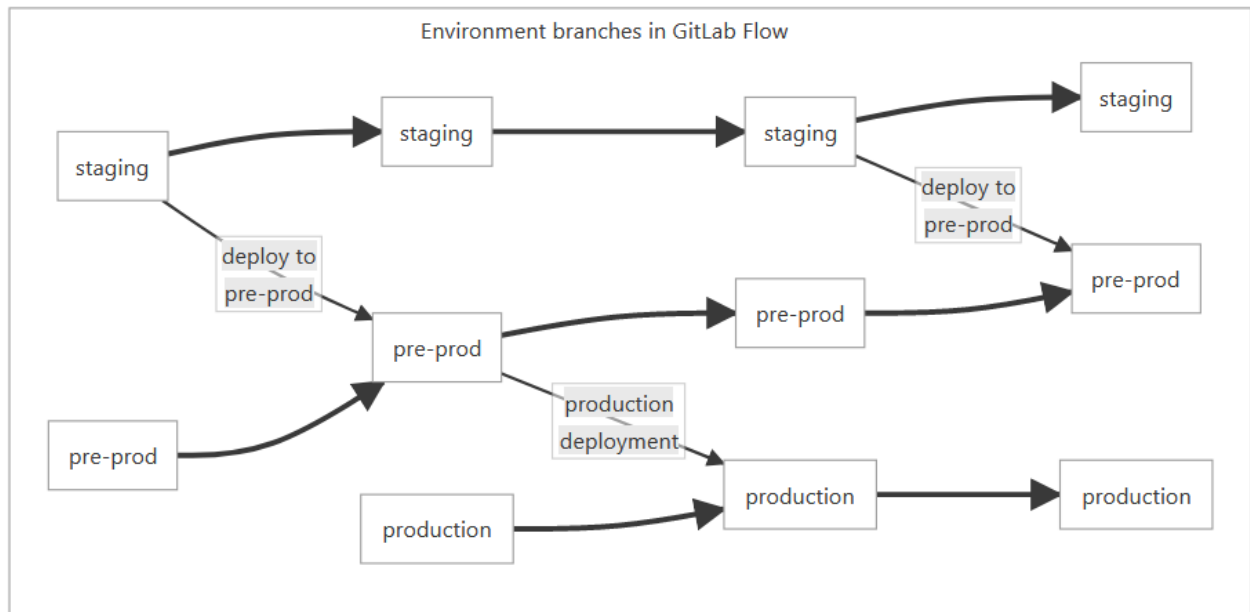
#### 1.1. git-flow (évolué mais trop complexe)



## 1.2. github-flow ( simple)



### 1.3. Gitlab flow (moins rapide , fiabilité privilégiée)





## 2. GitHub "pull request" et GitLab "merge request"

Les termes "pull request" et "merge request" désignent à peu près la même chose .

GitHub utilise le vocabulaire "pull request" et GitLab utilise plutôt "merge request" .

Une "pull request" est une **demande de fusion** d'une série de modifications d'une branche "features\_xyz" à une branche commune à une équipe de développeur (ex : "main" , "master" , "main\_dev" , ... ) .

Une "pull request" est **avant tout le début d'une discussion entre développeurs** pour **savoir si à terme les modifications proposées seront acceptées , refusées ou bien ajustées** avant d'être intégrées dans une branche commune.

### 2.1. Pull request (github)

Exemple :

```
git branch feature-a
git checkout feature-a
```

... des Ajouts (ex : tp.MyUtil.java , ...) et un premier commit

... d'autres ajouts/modifications et un second commit

#diffusion de cette branche sur github :

```
git push --set-upstream origin feature-a
```

Sur le site du référentiel "GitHub" , on peut créer une nouvelle "pull request" .

Pour commencer, le principal paramétrage à effectuer consiste à sélectionner :

- la branche réceptrice du futur merge (ex : m-dev)
- la branche des nouvelles fonctionnalités à ajouter/ajuster/intégrer (ex : feature-a)

```
destination_branche <--- feature_branch
m-dev                <--- feature-a
```

**Comparing changes**  
Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about diff comparisons](#).

base: m-dev ← compare: feature-a ✓ **Able to merge.** These branches can be automatically merged.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#) Create pull request

→ 2 commits      2 files changed      1 contributor

Commits on Feb 27, 2024

- MyUtil.display  
DidierTp\_dev\_x committed 39 minutes ago 2f6a357
- MyUtil.displayDlg  
DidierTp\_dev\_x committed 37 minutes ago b943b79

Showing 2 changed files with 16 additions and 0 deletions. Split Unified

```

basicJava/src/main/java/tp/Main.java
@@ -5,6 +5,9 @@ public class Main {
5      public static void main(String[] args) {
6          System.out.println("hello world from basicJava <<15>>");
7          System.out.println("ma suite v6 ..");
8      +
9      +         MyUtil.display("coucou");
10     +         //MyUtil.displayDlg("coucou2");
11     }

```

Une fois créé et initialisé (avec titre et petite description) :

**Feature a (MyUtil class with .display() and displayDlg()) #1**

Open didier-tp wants to merge 2 commits into m-dev from feature-a

Conversation 0      Commits 2      Checks 0      Files changed 2

didier-tp commented 1 minute ago Owner ...

MyUtil class is useful for display message

DidierTp\_dev\_x added 2 commits 43 minutes ago

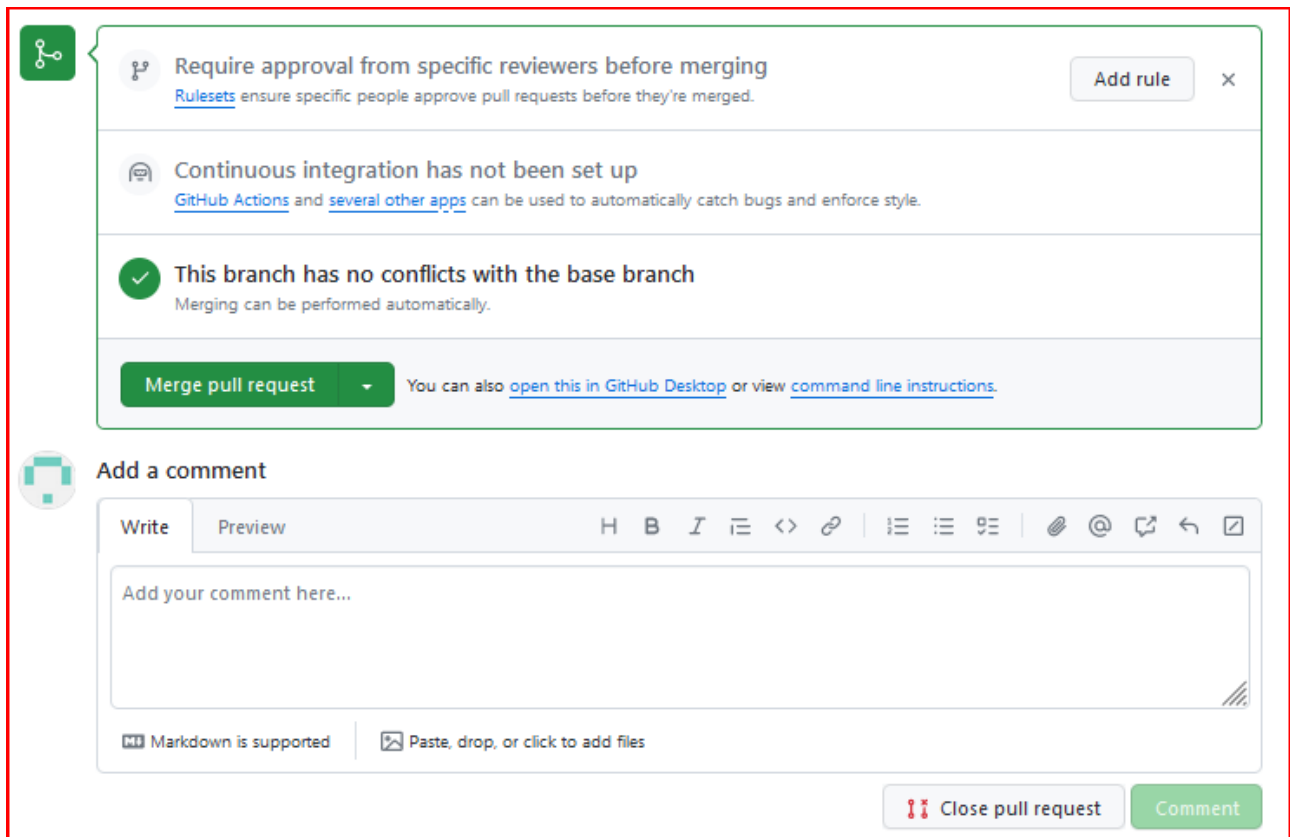
- MyUtil.display 2f6a357
- MyUtil.displayDlg b943b79

Add more commits by pushing to the [feature-a](#) branch on [didier-tp/git-classroom](#).

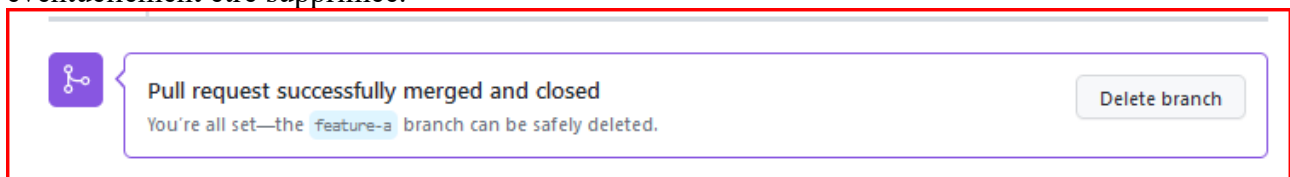
la "pull request" peut être vue comme une **nouvelle discussion** (sur plateforme "github" et avec lien quelquefois envoyé par email).

Cette discussion peut être à l'initiative de nouveaux ajustements (non définitifs, eux-même sujets à discussion) postés comme des commits additionnels sur la branche feature-a .

**NB :** la plateforme "gitHub" réactualise automatiquement l'état de la pull-request dès qu'un nouveau commit est "push"é sur la branche feature-a (plus envoi d'un mail à la personne à l'origine de la "pull-request") .



Une la fusion fois effectuée (merge associé à la pull-request) , la branche feature-a peut éventuellement être supprimée.



Pour une suppression complète (par ligne de commande) , on pourra enchaîner :

*#suppression de la branche locale*  
**git branch -d feature-a**

*#suppression de la branche distante*  
**git push origin --delete feature-a**  
 → affiche :

To https://github.com/didier-tp/git-classroom.git  
 - [deleted] feature-a

*#suppression à lancer sur les autres clones :*

**git fetch --all --prune**

et (éventuellement si besoin) :

**git branch -d feature-a**

## 2.2. Merge request (requête de fusion) avec gitlab :

✓ Vous avez poussé sur f1 📄 Il y a 7 minutes

[Créer une requête de fusion](#)

### Nouvelle requête de fusion

#### Branche source

didier-lab/git-classroom f1



f2Bis  
Didier Defrance a rédigé sept. 29, 2025

4ec88bcb



[Comparer les branches et continuer](#)

#### Branche cible

didier-lab/git-classroom main



f3EnPlus  
Didier Defrance a rédigé sept. 29, 2025

d1a5ae58



#### Merge can start

N'importe quand

Nécessite que les vérifications de fusion soient réussies.

#### Options de fusion

- ☒ Supprimez la branche source lorsque la requête de fusion est acceptée.
- ☐ Écrasez les validations lorsque la requête de fusion est acceptée. ?

[Créer une requête de fusion](#)

[Annuler](#)

### Nouvelle requête de fusion

De f1 vers main [Modifier les branches](#)

Titre (obligatoire)

fusion souhaitée de la branche F1 vers la branche main



Prêt pour la fusion !



- ☒ Supprimer la branche source ☐ Combiner (squash) les commits ? ☐ Modifier le message de validation

- La branche source est [1 validation de retard](#) sur la branche cible. [Rebaser la branche source](#)
- 2 validations et 1 validation de fusion seront ajoutées à main.

[Fusionner](#)

### 3. Fork de dépôt git

L'interface graphique de github (ou de ....) permet si besoin de créer une copie d'un dépôt/référentiel git .

Cette nouvelle copie pourra évoluer de manière complètement indépendante de l'original.

Lors d'un fork de dépôt git , l'historique du projet est conservée.

Si l'on souhaite dupliquer un dépôt git sans son historique , il vaut mieux de pas déclencher un fork mais :

- 1) cloner le référentiel original avec l'option `--depth nbRevisions`
- 2) changer l'url remote pour stocker le projet dans un nouveau dépôt

### 4. Git patch

GIT patch est utilisé pour partager des propositions de changements sans les diffuser (via git push) sur une branche importante. De cette façon les autres personnes (ayant reçu un patch par exemple par email) pourront appliquer et tester ces changements temporaires pour les évaluer.

Git patch peut éventuellement être vu comme un moyen particulier de diffuser certains correctifs.

NB : git patch permet également la diffusion de parties additionnelles d'un projet à un autre.

#### 4.1. Création d'un patch

1) Quelques ajouts/modifications  
et quelques "git add ..." et "git commit" SANS PUSH (par exemple sur une branche locale) .

2) **git format-patch -1**

*ou bien*

**git format-patch -2**

*selon le nombre de derniers commits à placer dans le patch*

→ cette commande génère dans le répertoire local du projet un fichier par commit patché :

```
0001-with_f7.patch
0002-with_f7_inprovement.patch
```

Ces fichiers **.patch** peuvent être diffusé de manière quelconque (par email , par clef usb , ....)

#### 4.2. Application d'un patch

(exemple: sur autre clone ou bien autre projet)

```
git apply -v /chemin/vers/votre/fichier.patch
```

Exemple :

```
git apply -v 0001-with_f7.patch
git apply -v 0002-with_f7_inprovement.patch
```

NB :

- Une fois que le patch a été appliqué, l'effet doit normalement se voir sur un ou plusieurs fichiers du répertoire de travail
- il vaut mieux déplacer ou supprimer les fichiers .patch une fois l'application effectuée
- on peut éventuellement utiliser git diff si le patch a eu pour effet de modifier certains fichiers

Si les ajouts/modifications conviennent, on peut alors les enregistrer classiquement (git add, git commit) .

Exemple :

```
git add f7.txt
git commit -m "avec patchs f7 appliqués"
```

NB :

Dans certains cas, le patch peut ne pas s'appliquer proprement (à cause de conflits)

Si cela se produit, l'option **--reject** avec **git apply** permet créer des fichiers **.rej** pour chaque conflit et l'on peut résoudre alors manuellement ces conflits :

```
git apply --reject /chemin/vers/votre/fichier.patch
```

## 5. utilisation d'un patch pour réorganiser commits

Soit main avec quelques modifs/commits m1,m2 (ex : sur f1a.txt)

**Soit bf1 une branche de fonctionnalité** avec quelques modifs/commits mm1,mm2,mm3 (ex : sur fb1.txt, fb2.txt) ou bien avec **de très nombreux commits successifs mal organisés**

En se plaçant sur bf1, on peut commencer par lancer *git rebase main* (ou *git merge main*) pour réintégrer dans bf1 les modifications effectuées sur main.

```
git diff main bf1 > ~/bf1.patch ou ../bf1.patch
```

Se replacer sur la branche main :

```
git checkout main
```

détruire l'ancienne version de la branche bf1 via l'option -D :

```
git branch -D bf1
```

*recréer une nouvelle version de la branche bf1 et se placer dessus :*

```
git checkout -b bf1
```

appliquer le patch sur la nouvelle version de la branche bf1 :

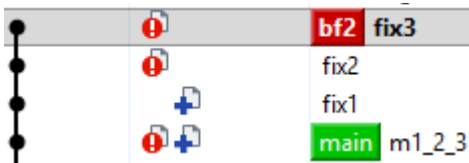
```
git apply ~/bf1.patch ou ../bf1.patch
```

enregistrer toutes les modifs rapatriées sur la nouvelle branche bf1 via **git add** et **git commit**

...

## 6. Réorganisation des commits lors d'un merge:

Soit une branche bf2 (issue de main) et avec 3 commits successifs fix1,fix2,fix3



git checkout main

```
git merge bf2 --squash
```

Updating b5af67c..337f39a

Fast-forward

**Squash commit -- not updating HEAD**

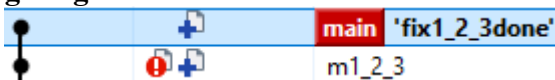
f3a.txt | 3 +++

1 file changed, 3 insertions(+)

create mode 100644 f3a.txt

```
git commit -m 'fix1_2_3done'
```

git log



git checkout bf2

git log (inchangé : fix1 et fix2 et fix3)

git checkout main

git branch -D bf2

Quand utiliser l'option <b>--squash</b> lors d'un merge	Pour <b>cacher une succession de petits messages de commits pas du tout importants/intéressants</b> ( <i>essai 1</i> , <i>retour arrière</i> , <i>correctif_a</i> , <i>correctif_b</i> , <i>meilleur_solution</i> , ...) et remplacer tout cela par un bon message synthétique et un seul gros commit
Quand <b>ne pas</b> utiliser l'option <b>--squash</b> lors d'un merge	Si l'on souhaite conserver un historique détaillé ou si plusieurs développeurs ont travaillé sur la branche et que l'on ne souhaite pas masquer/écraser les contributions individuelles .

## 7. RAZ d'un référentiel git (RAZ de l'historique)

Attention, cette opération radicale doit être vue comme exceptionnelle !!!

*#construit une nouvelle branche (sans parent) sur laquelle on sera automatiquement placée:*  
**git checkout --orphan temp\_raz**

**git add \***  
**git status**  
**git commit -m "raz"**

*#delete old main branch (ATTENTION , DEFINITIF en local)*  
**git branch -D main**

*#renommer la branche courante (ici "temp\_raz") en "main":*  
**git branch -m main**

*#force update do remote/upstream:*  
**git push -f origin main**

**git count-objects -v**

*#Attention : il faut également penser à détruire (et si besoin re-crée) toutes les éventuelles branches secondaires (localement et à distance, sur tous le clones , ...)  
pour que l'ensemble reste cohérent !!!!*



# ANNEXES

# VIII - Ancienne architecture centralisée (CVS, SVN)

## 1. SVN

### 1.1. Terminologie & fonctionnement de CVS et SVN

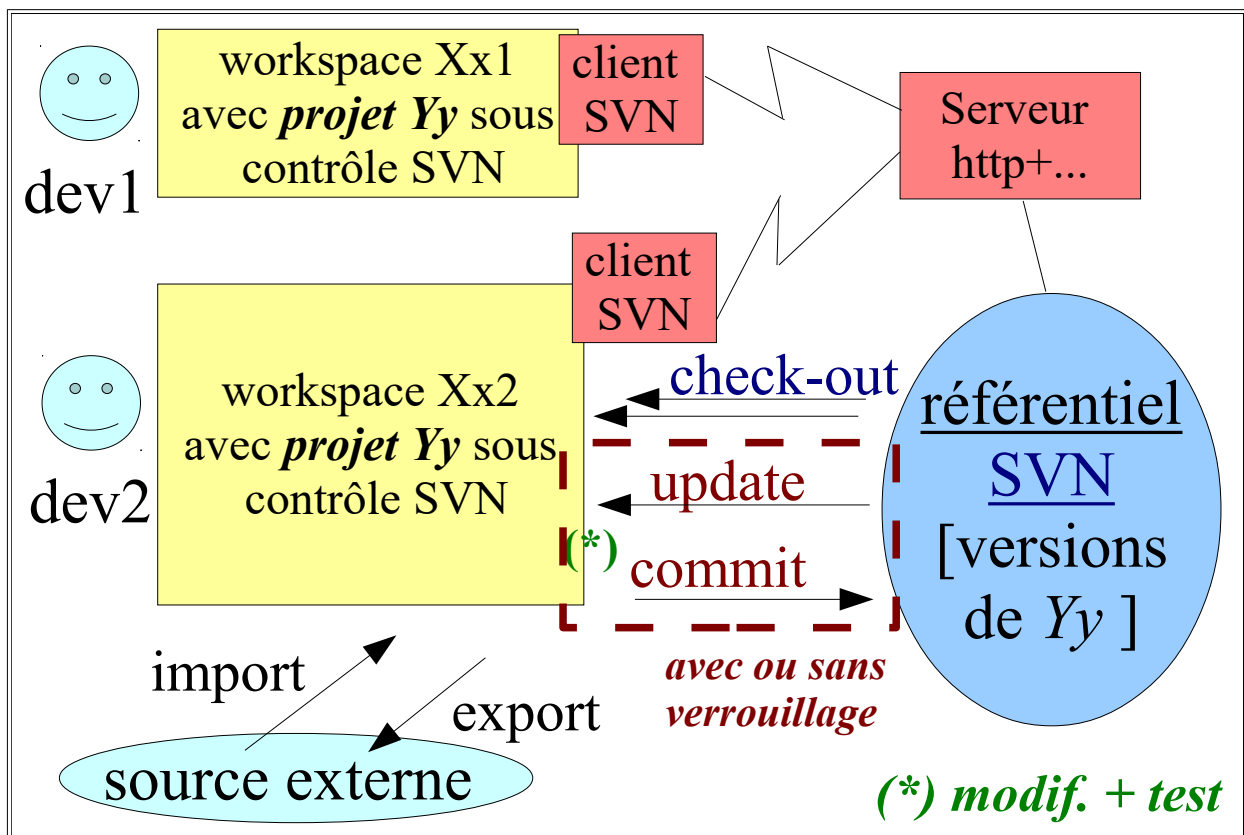
CVS = Concurrent Version System , SVN = SubVersioN

CVS est un produit "Open Source" qui permet de **gérer différentes versions d'un ensemble de fichiers sources** lié au développement d'un certain module logiciel.

**Différents programmeurs peuvent travailler en équipe sur des fichiers partagés au niveau d'un référentiel commun.**

SVN se veut avant tout être une "*version améliorée de CVS*" qui

- ne remet pas en cause les principes fondamentaux de CVS (référentiel commun et commit,update,... )
- a refondu l'implémentation du serveur et des référentiels (meilleure gestion des transactions, protocole d'accès plus simples, ...)



Vocabulaire utilisé par CVS (puis SVN):

<b>import</b>	Créer un nouveau module en rapatriant dans le référentiel le code d'un répertoire
---------------	---

	existant [qui devient alors "l'ancienne source"].
<b>check-out</b>	Récupérer une copie à jour (vue locale à un programmeur) d'un module logiciel géré par SVN. Cette vue locale correspondra à un répertoire de travail qui sera sous le contrôle de SVN [NB: Ce répertoire contiendra un sous répertoire caché .svn ]
<b>update</b>	Récupérer la dernière révision d'un fichier (pour lecture et/ou mise à jour)
<b>commit</b>	Enregistrer une nouvelle version d'un fichier (après modification et test unitaire). SVN incrémente alors automatiquement le numéro de révision.
<b>export</b>	Extraire depuis le référentiel une copie d'un module. Le répertoire externe alors créé ne sera plus sous le contrôle de SVN [Là est la principale différence avec un check-out classique]
...	

### Vocabulaire (suite) utilisé par CVS et SVN:

<b>module</b>	module logiciel (Application, Librairie, ...) à placer sous le contrôle de SVN. Un module correspond à une arborescence de fichiers (sources, make).
<b>révision</b>	Version auto-incrémentée précise d'un module SVN (ou d'un fichier avec CVS) (ex: <b>1.1</b> , 1.2 , ... 1.9 , 1.10 , <b>1.11</b> , ...) <u>NB</u> : incrémentation automatique lors d'un "commit"
<b>Release / Tag</b>	version contrôlée / taguée d'un module logiciel complet (ex: " <b>rel-1-1</b> " , " <b>rel-1-2</b> " , ...) <u>NB</u> : un nom de release (ou <b>Tag</b> ) doit avec CVS commencer par une lettre et ne doit pas contenir de point (.) ni de blanc .
...	

### Principales différences entre SVN et CVS:

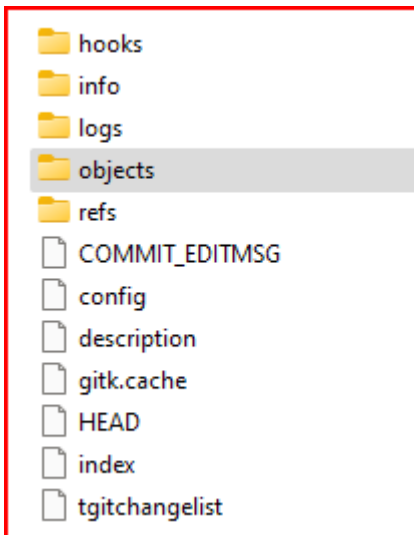
- Les numéros de révisions SVN sont liés à l'ensemble d'un module (projet) et non plus à un seul fichier (comme pour CVS).
- ....

## IX - Annexe – Structure interne de GIT (objets)

### 1. Structure interne d'un référentiel GIT

#### 1.1. Contenu du répertoire caché .git

Un projet pris en charge par git se reconnaît par le fait qu'il comporte un sous répertoire ".git" dont la structure est la suivante :



Le sous répertoire "objects" comporte des blobs référencés par d'autres éléments.

#### 1.2. Blobs , Tree et références

Au sein de git, le contenu des fichiers est stocké (en interne) dans des objets appelés **blobs**, gros objets binaires.

Contrairement aux fichiers ordinaires , un **blob** ne comporte que du contenu et aucune métadonnée (pas de date de création, pas de nom de fichier)

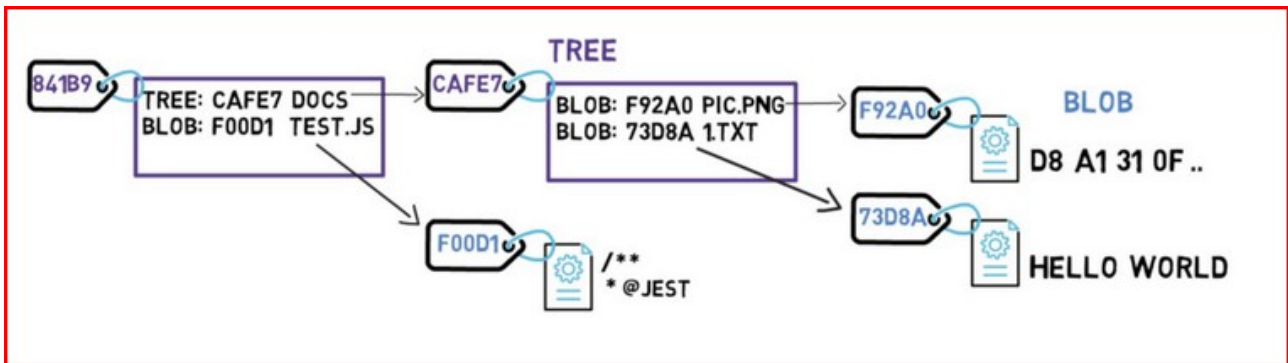
Certains blobs peuvent avoir une taille de plusieurs centaines de ko .

Chaque **blob** est identifié par son [hachage SHA-1](#). Les hachages SHA-1 sont constitués de 20 octets, généralement représentés par 40 caractères sous forme hexadécimale (avec des représentations souvent simplifiées/tronquées dans la plupart des exemples/schémas).

Au sein de git, l'équivalent d'un annuaire est un **arbre (tree)**. Un **arbre** est essentiellement une liste d'annuaires, faisant référence à des **blobs** ainsi qu'à d'autres **arbres** .

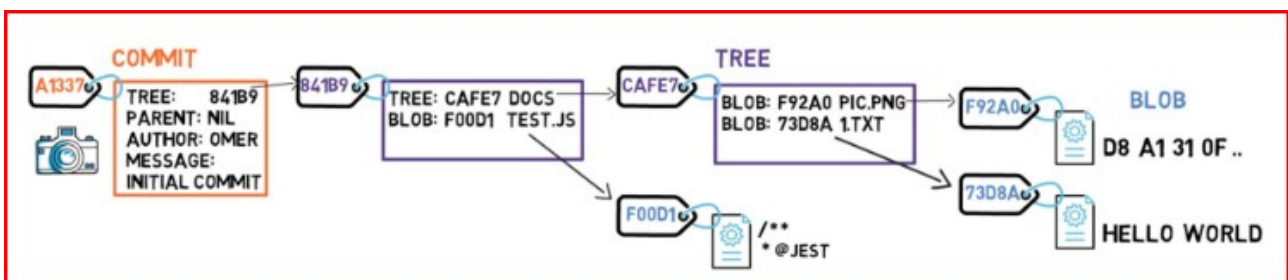
Les arbres sont également identifiés par des hachages SHA-1 .

Exemple :



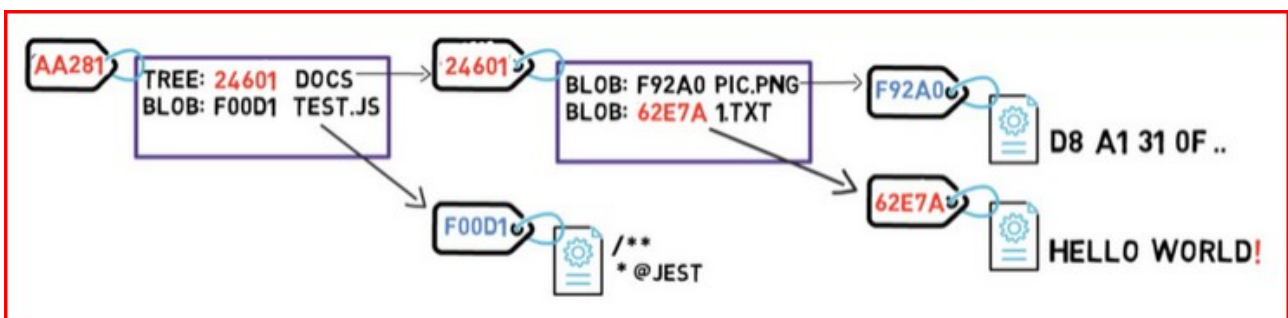
Le diagramme ci-dessus est équivalent à un (sous-)système de fichiers avec un répertoire racine qui a un fichier à */test.js*, et un répertoire nommé */docs* comportant deux fichiers: */docs/pic.png* et */docs/1.txt*

Au sein de git, un **instantané (snapshot)** est un **commit**. Un objet **commit** comprend un pointeur vers l'**arbre** principal (le répertoire racine), ainsi que d'autres métadonnées telles que le message de **commit** et l'**auteur** et l'**instant** du commit. Un commit est lui aussi identifié par un SHA-1 .

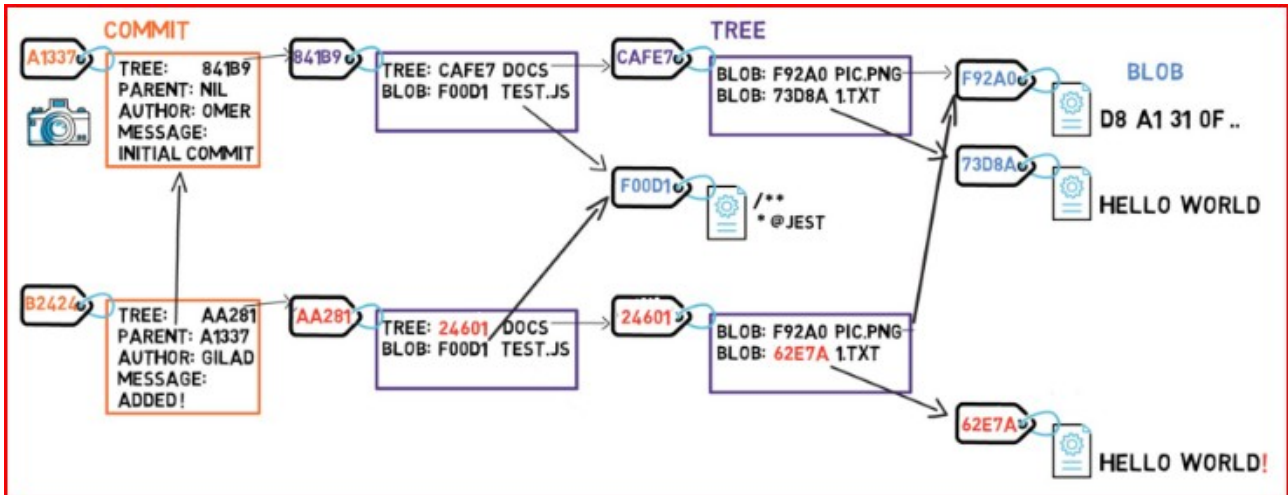


**NB :** chaque commit comporte le contenu complet d'un instantané (et pas seulement la différence entre un ancien commit).

Lors d'un nouveau commit , toute une partie de l'arborescence (ci dessous en rouge) est alors automatiquement modifiée (en cascade) et certaines parties non modifiées (ci dessous en bleu) sont réutilisées telles quelles au sein de l'arbre des référencements .

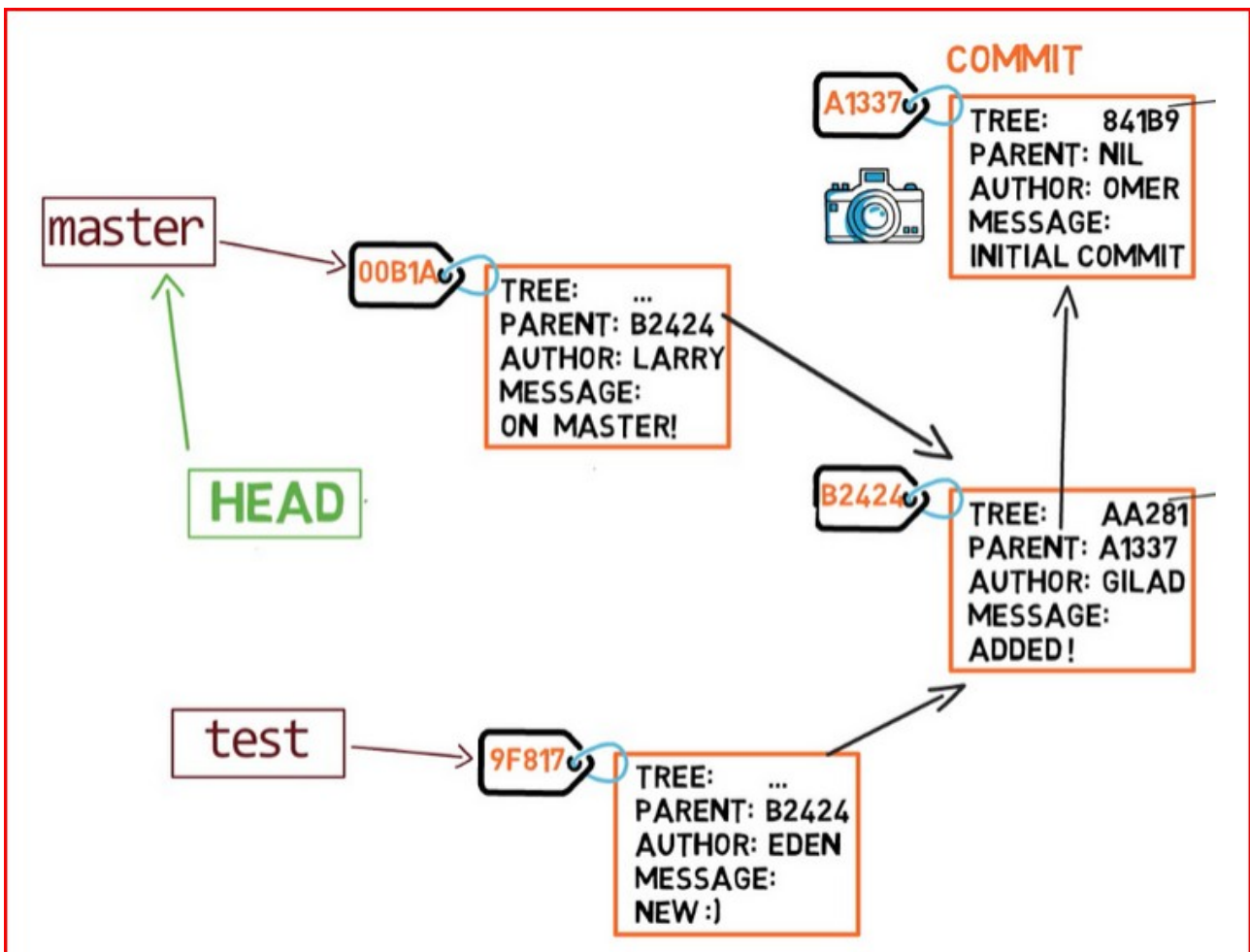


Vue d'ensemble sur les référencements principaux liés à deux commits successifs :



Au sein de git , les branches sont codées que comme des références sur certains commits .  
La référence spéciales "HEAD" référence (la tête de) la branche courante .

Exemple de situation (branche "test" et branche courante "master" ) :



# X - Annexe – GIT avec IDE eclipse

## 1. Plugin eclipse pour GIT (EGIT)

Le plugin eclipse pour GIT s'appelle EGIT .

### 1.1. Actions basiques (commit , checkout , pull , push)

→ Se laisser guider par la perspective "GIT" et via le menu "Team"

### 1.2. Résolution de conflits

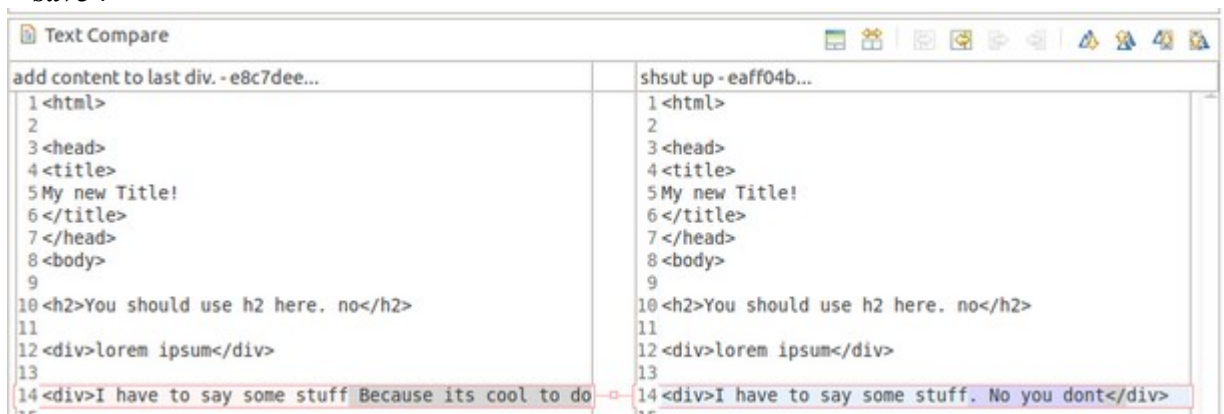
- 1) déclencher "**Team / pull**" pour récupérer (en tâche de fond) la dernière version (partagée / de l'équipe). Le plugin EGIT va alors tenter un "**auto-merge**" ("git fetch FETCH\_HEAD" suivi par "git merge").
- 2) En cas de conflit (non résoluble automatiquement) , les fichiers en conflit seront marqués d'un point rouge.

```
<<<<<< HEAD
<div>I have to say some stuff Because its cool to do so.</div>
=====
<div>I have to say some stuff. No you dont</div>
>>>>>> branch 'master' of /var/data/merge-issue.git
```

Sur chacun des fichiers en conflit , on pourra déclencher le *menu contextuel*

"**Team / Merge tool**" . (laisser par défaut la configuration de "Merge Tool" : use HEAD).

- 3) *Saisir , changer ou supprimer alors au moins un caractère dans la zone locale (à gauche) + save :*



... au cas par cas ....

- 4) Déclencher le menu contextuel "**Team / add to index**" pour ajouter le fichier modifié dans la liste de ceux à gérer (staging).
- 5) Effectuer un "**Team / commit**" local .
- 6) Effectuer un "**Team / push to upstream ...**" pour mettre à jour le référentiel distant/partagé .

### 1.3. Sécurité GIT

Attention : la gestion au sens username/password (basic http authentication) des mots de passe par le plugin EGIT ne fonctionne pas en mode token.

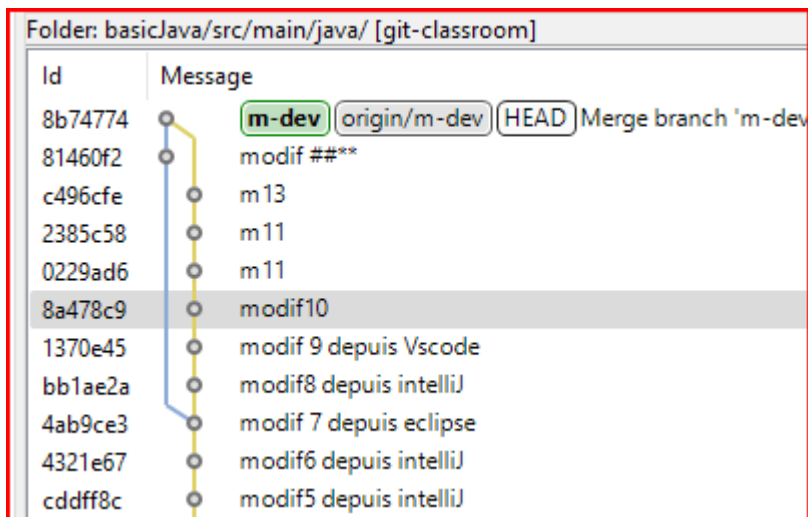
En mode "token" , pas de password mais la valeur du token doit être précisée en tant que partie de l'URL du référentiel GIT à utiliser

Rappel du format :

[https://username:token\\_value\\_string@github.com/repoOwner/repoXyz.git](https://username:token_value_string@github.com/repoOwner/repoXyz.git)

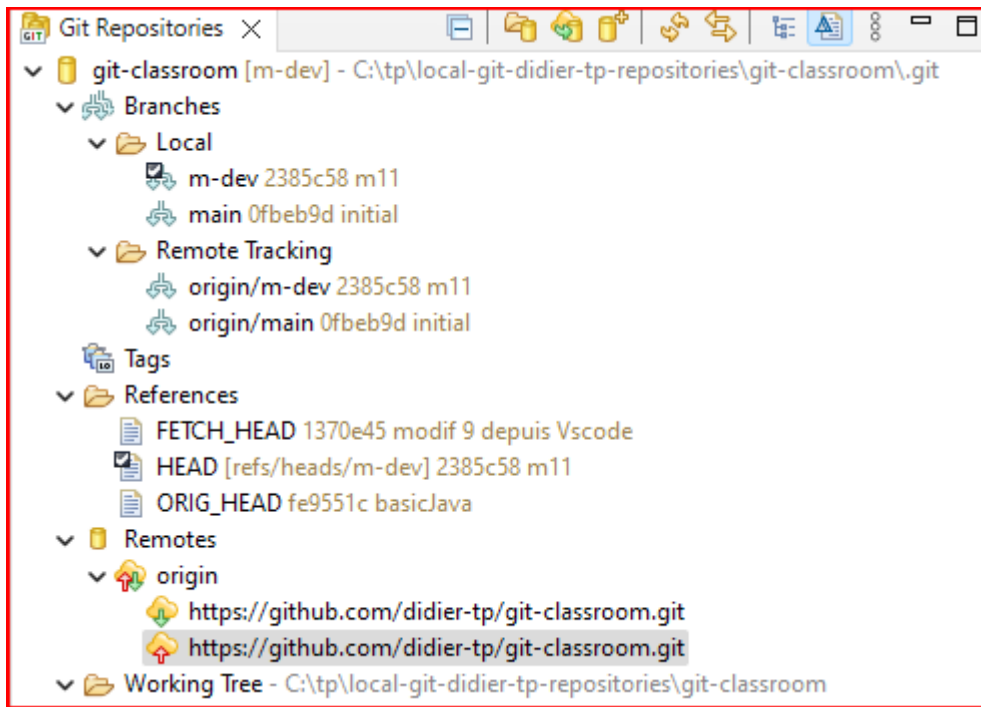
### 1.4. Autres fonctionnalités eclipse/git

Menu contextuel "Teams ... / show in history" pour ensuite gérer simplement les tags , ....



### 1.5. Perspective "Git" de eclipse (vue d'ensemble sur les branches)





# XI - Annexe – GIT avec IDE intelliJ

## 1. Bien préparer l'url du référentiel distant

Tous les IDE modernes vont automatiquement reprendre tous les paramétrages du référentiel local (souvent issu d'un clonage d'un référentiel distant).

Sur le référentiel local, on aura intérêt à bien préparer l'url distante (valeur de l'alias *origin*) avec si besoin un token de sécurité dans cette URL.

Exemple :

```
git remote set-url origin https://didier-tp:valeur_du_jeton@github.com/didier-tp/git-classroom.git
```

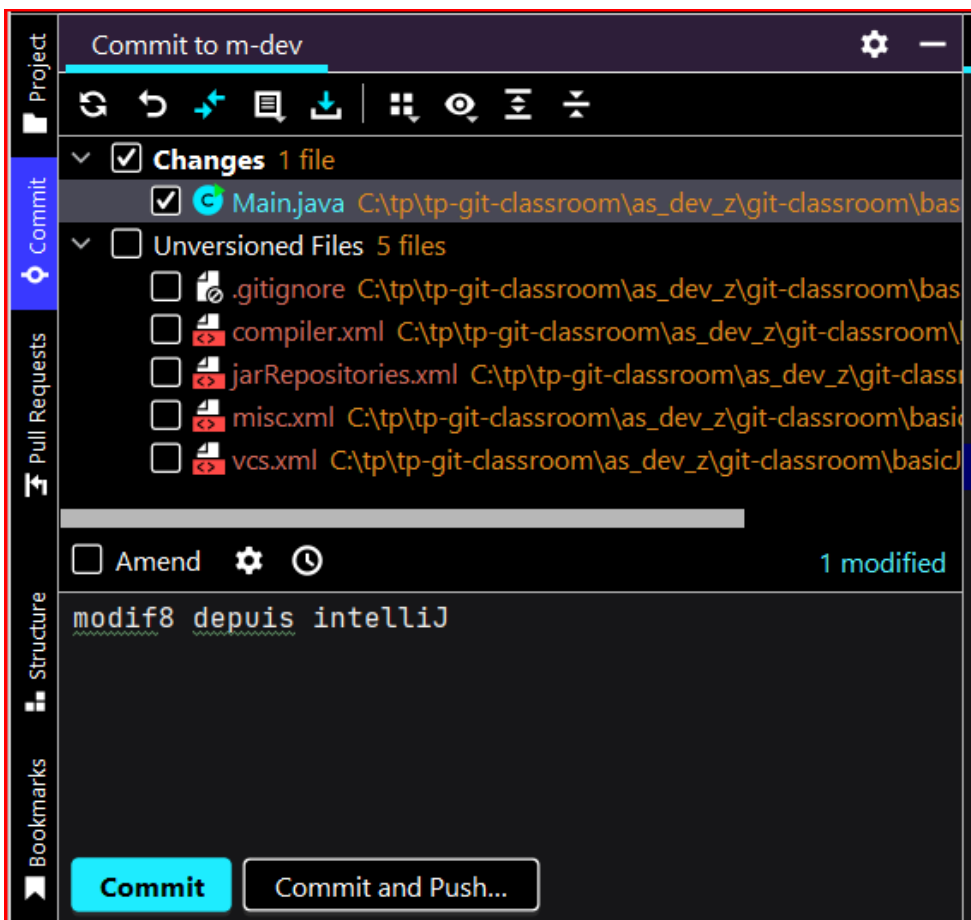
et si besoin :

```
git config --system --unset credential.helper
```

```
git config --global --unset credential.helper
```

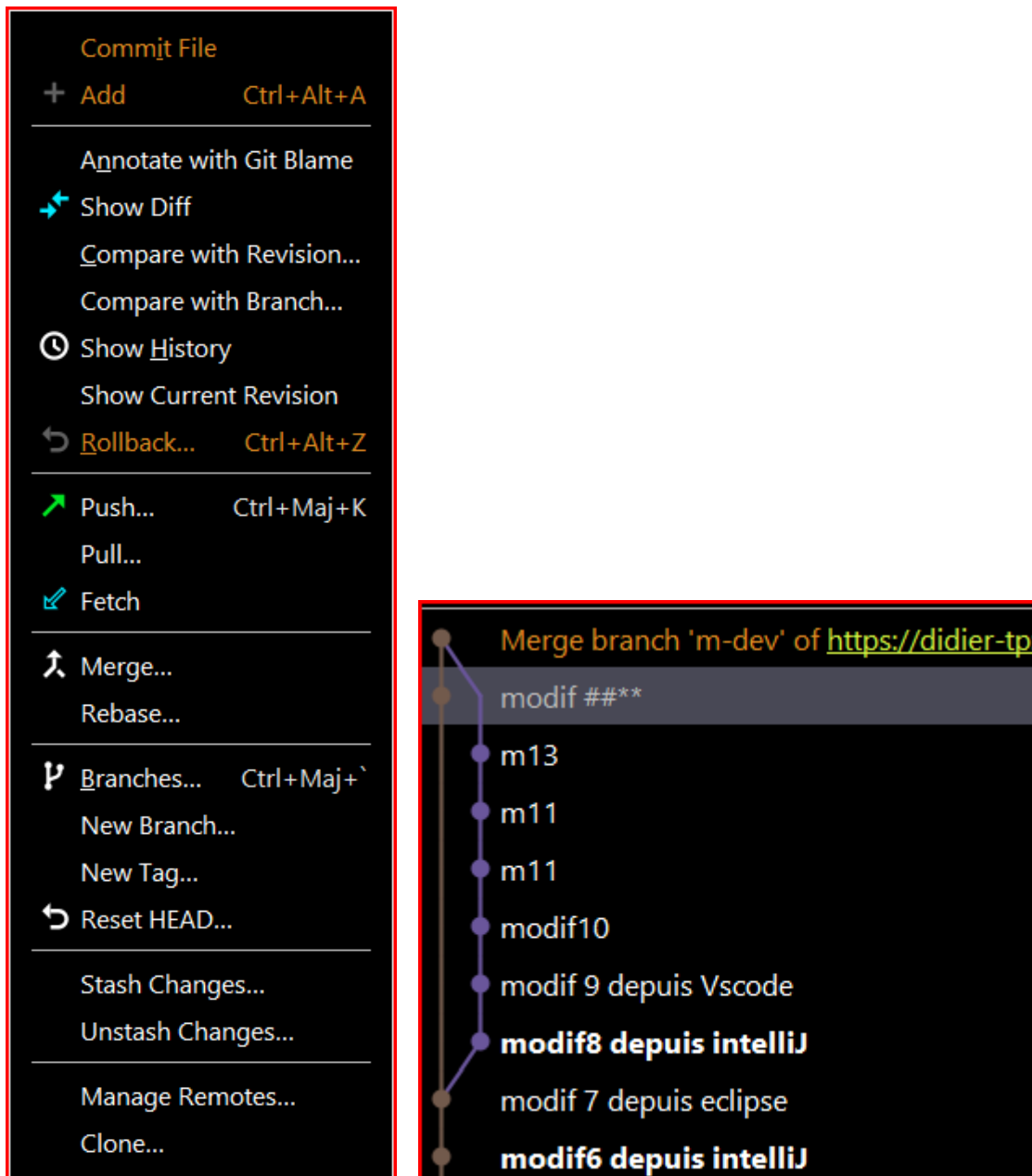
## 2. GIT avec IntelliJ

### 2.1. Commit au sein de IntelliJ



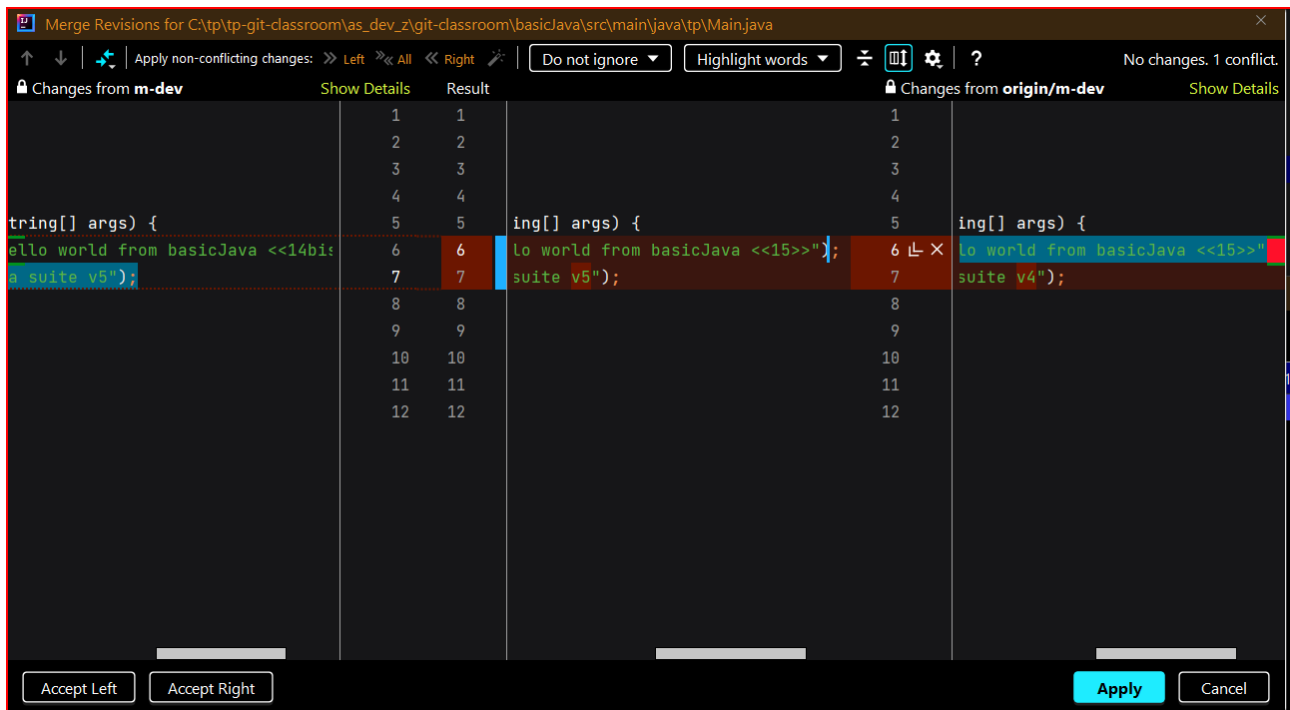
- 1) onglet gauche (en bleu) "commit" à côté de "project"
- 2) sélectionner fichier(s) à commiter
- 3) ajuster le message de commit
- 4) "commit" ou bien "commit and push"

## 2.2. Menu contextuel général "GIT" de intelliJ



Menu intuitif pour git ... pull , show\_history etc

## 2.3. Merge Tool d'intelliJ



Dans la logique d'intelliJ, la version à construire et converser sera celle du milieu.  
 Cette version est à construire en s'inspirant de notre version (à droite)  
 et de la version "remote" (à gauche)

## XII - Annexe – GIT avec IDE VisualStudioCode

### 1. Bien préparer l'url du référentiel distant

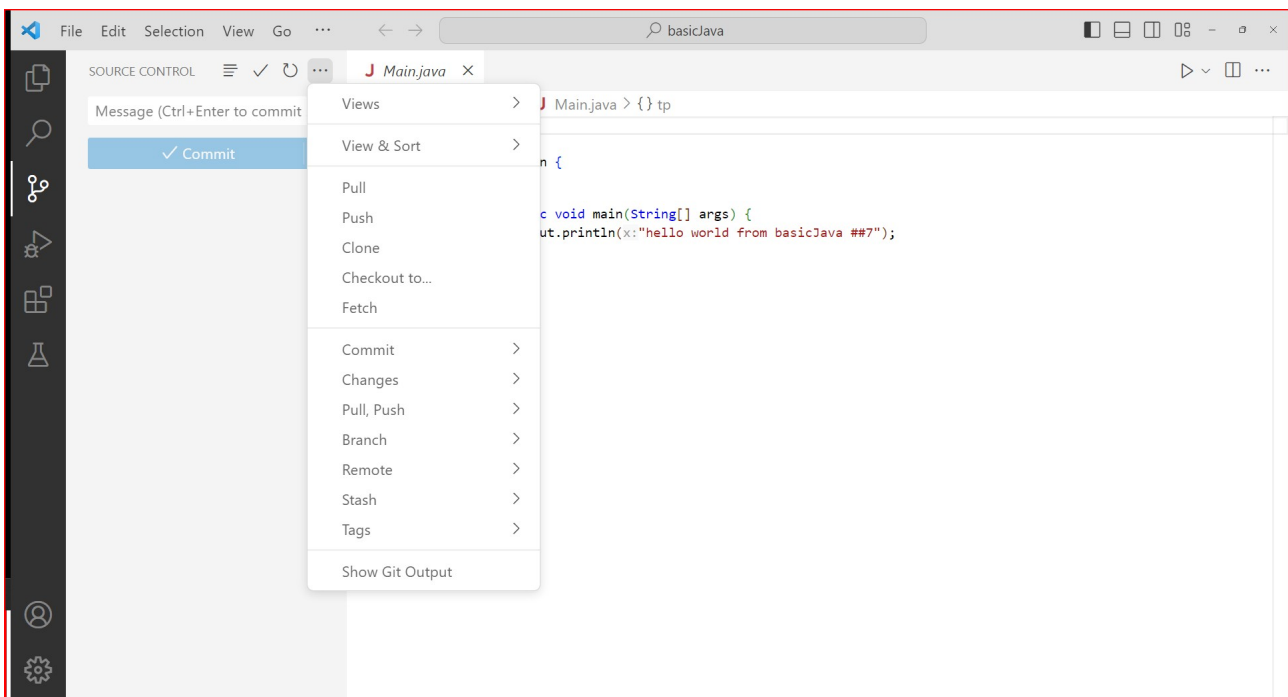
Tous les IDE modernes vont automatiquement reprendre tous les paramétrages du référentiel local (souvent issu d'un clonage d'un référentiel distant).

Sur le référentiel local, on aura intérêt à bien préparer l'url distante (valeur de l'alias *origin*) avec si besoin un token de sécurité dans cette URL.

Exemple :

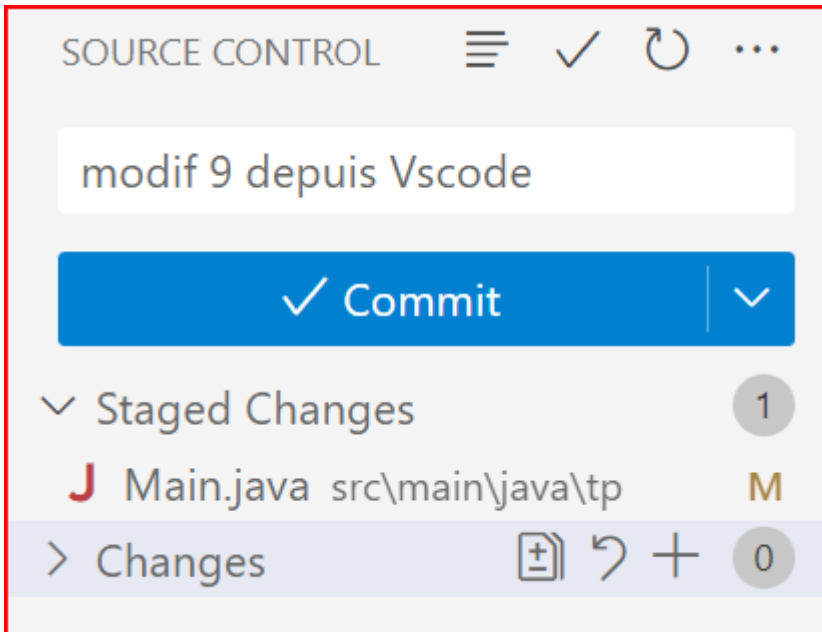
```
git remote set-url origin https://didier-tp:valeur_du_jeton@github.com/didier-tp/git-classroom.git
```

### 2. Git au sein de l'IDE "Visual Studio Code"

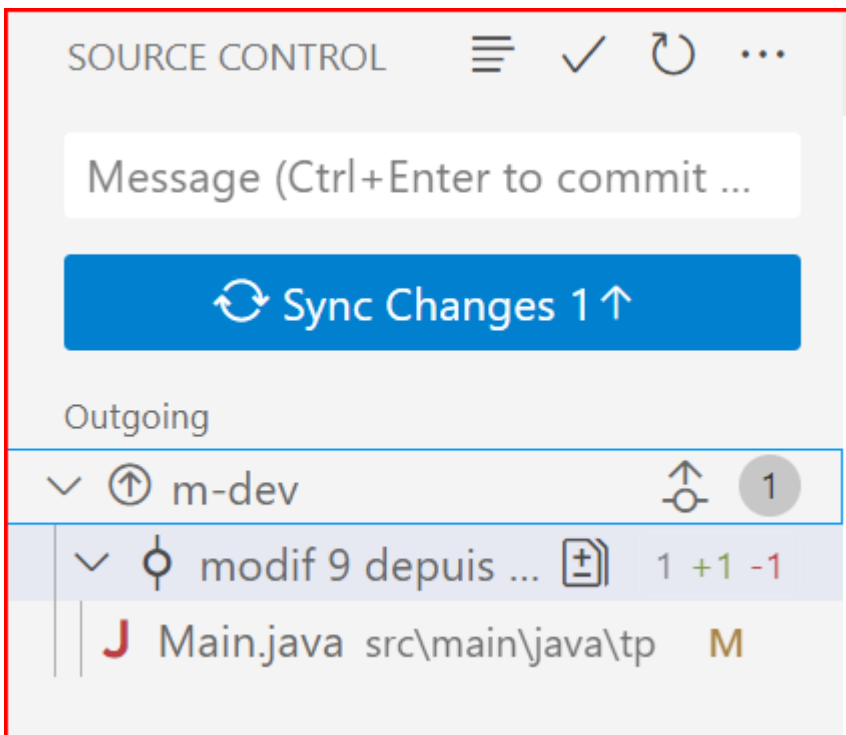


- 1) sur le coté gauche (**Source Control**) ou bien Ctrl+shift+G
- 2) icône ... (en haut) pour faire apparaître le menu contextuel global (pull , ....)

## 2.1. commit avec VsCode



## 2.2. push après un commit



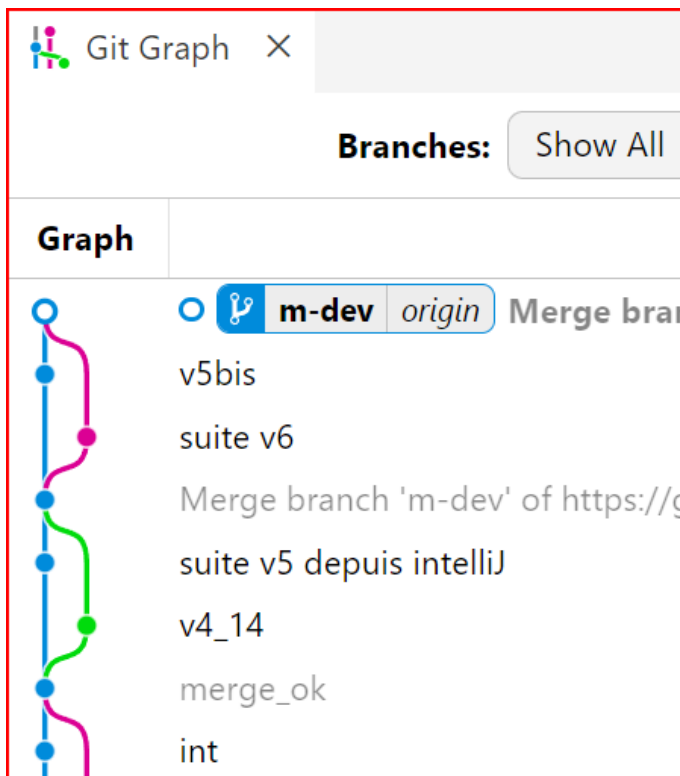
## 2.3. Merge tool (when conflict)



La version à construire et converser en bas pourra s'inspirer de la version distante (à gauche) et la version courante/locale (à droite).

Et comme d'habitude "complete merge" , "push" , ...

## 2.4. Show git history via "git graph extension"



## XIII - Annexe – Serveur GIT "on premise"

### 1. GitLab "On Premise"

Si l'on souhaite mettre en place un serveur de référentiel git privé en mode "OnPremise" sur un des ordinateurs de notre entreprise/organisation, on peut (entre autres) s'appuyer sur GitLab-CE ou bien GitLab-ee.

#### 1.1. Via docker

**docker pull gitlab/gitlab-ce**

*ou bien*

**docker pull gitlab/gitlab-ce:16.7.6-ce.0**

#### **docker-compose.yml (à adapter)**

```
version: '3.6'
services:
  gitlab:
    image: gitlab/gitlab-ce:16.7.6-ce.0
    container_name: gitlab
    restart: always
    hostname: 'gitlab.example.com'
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        # Add any other gitlab.rb configuration here, each on its own line
        external_url 'https://gitlab.example.com'
    ports:
      - '80:80'
      - '443:443'
      - '22:22'
    volumes:
      - '$GITLAB_HOME/config:/etc/gitlab'
      - '$GITLAB_HOME/logs:/var/log/gitlab'
      - '$GITLAB_HOME/data:/var/opt/gitlab'
    shm_size: '256m'
```



## 2. Très ancienne configuration d'accès distant à un référentiel Git

Si Répertoire "réseau" partagé (via NFS ou autre) , URL possible en [file:///](#)  
(ex: `file:///var/git/project.git`)

### 2.1. Accès distant (non sécurisé) via git

URL de type **git://nomMachineOuDomaine/xxxx/projetYy.git**  
où xxx est le chemin menant au référentiel git sur la machine "serveur" (ex: `/var/git/`)  
(Alias possible dans `.gitconfig`)

### 2.2. Accès distant sécurisé via git+ssh

URL de type **ssh://user@hostXx/var/git/projectYy.git**

git+ssh est un tunneling sécurisé ssh pour le protocole git

Les clefs (publiques et privées) ssh sont placés dans le répertoire `$HOME/.ssh`

Celles ci se génèrent via la commande **ssh-keygen** .

Lors de la génération des clefs , un mot de passe (*passphrase*) à retenir est demandé .

Ce mot de passe peut éventuellement être vide (sécurité alors que via la clef publique) .

La **clef publique** (à envoyer par email ou ....) correspond au fichier **id\_dsa.pub** (ou bien **id\_rsa.pub**) .

Si la partie "serveur" de ssh n'est pas encore installée , on peut alors lancer "**sudo apt-get install openssh-server**" puis éventuellement "**sudo service ssh start**" .

NB : sur certaines versions de Linux Ubuntu , la commande apt-get install ne fonctionne pas bien avec openssh-server et l'on peut dans ce cas installer alternativement openssh-server de la façon suivante :

- 1) télécharger le fichier **openssh-server\_5.9p1-5ubuntu1\_i386.deb** (via une recherche google)
- 2) lancer **sudo dpkg --install ./openssh-server\_5.9p1-5ubuntu1\_i386.deb**
- 3) redémarrer linux (ou ...) pour que le service "ssh" soit activé

### 2.3. Accès distant en lecture seule via http (sans webdav)

Configurer un accès de apache2 vers un répertoire correspondant à un référentiel git et activer le hook "post-update" en renommant `post-update.sample` en `post-update` .

```
cd xy.git
mv hooks/post-update.sample hooks/post-update
```

## 2.4. Accès distant en lecture/browsing via gitweb

En configurant sur le poste serveur , l'extension "gitweb"(pour apache2 et git) , on peut alors parcourir toute l'arborescence d'un projet GIT via un simple navigateur.

<http://localhost/gitweb/?p=p0.git>



Activation et configuration du site web "gitweb":

```
cd /var/www;
sudo mkdir gitweb;
cd gitweb;
sudo cp /usr/share/gitweb/* . ;
sudo cp /usr/share/gitweb/static/* .
```

Il faut également fixer la variable **\$projectroot = "/var/scm/git/"** dans le fichier **/etc/gitweb.conf**

**/etc/gitweb.conf**

```
$projectroot="/home/formation/scm/git";
# directory to use for temp files
$git_temp="/tmp";
# html text to include at home page
$home_text="indextext.html";
# file with project list; by default, simply scan the projectroot dir.
$projects_list=$projectroot;
# stylesheet to use
# I took off the prefix / of the following path to put these files inside gitweb directory directly
$stylesheet="gitweb.css";
# logo to use
$logo="git-logo.png";
# the 'favicon'
$favicon="git-favicon.png";
```

D'autre part, le module " RewriteEngine" d'apache2 doit être activé.  
Si ce n'est pas encore le cas, on l'active via la commande "**sudo a2enmod rewrite**"

Créer et configurer un nouveau fichier pour configurer gitweb sous apache2 :

**/etc/apache2/conf.d/git.conf**

```
...
<Directory /var/www/gitweb >
SetEnv GITWEB_CONFIG /etc/gitweb.conf
DirectoryIndex gitweb.cgi
Allow from all
AllowOverride all
Order allow,deny
Options +ExecCGI
AddHandler cgi-script .cgi
<Files gitweb.cgi>
    SetHandler cgi-script
</Files>
RewriteEngine on
RewriteRule ^[a-zA-Z0-9_-]+.git/(?\.?)?$ /gitweb.cgi%{REQUESTURI} [L,PT]
</Directory>
....
```

Redémarrage du service apache2:  
**service apache2 restart**

## 2.5. Accès distant "rw" via http/https (webdav)

Activer les *modules apache2* "dav" , "dav\_fs" et "dav\_lock"

```
sudo a2enmod dav
sudo a2enmod dav_fs
sudo a2enmod dav_lock
```

Créer et configurer un nouveau fichier pour configurer git sous apache2 :

**/etc/apache2/conf.d/git.conf**

```
...
Alias /git /var/scm/git/

<Location /git>
    DAV on
    #AuthType Basic
    #AuthName "Git"
    #AuthUserFile /etc/apache2/dav_git.passwd
    #Require valid-user
</Location>
...
```

Redémarrage du service apache2:

**service apache2 restart**

+ si besoin paramétrage d'autres détails (sécurité , ...) :

```
<Directory "/var/scm/git/">
Options Indexes FollowSymLinks MultiViews ExecCGI
#DirectoryIndex index
AllowOverride None
Order allow,deny
allow from all
</Directory>
```

Eventuel paramétrage (facultatif et très délicat) pour optimiser les transferts (par paquets) via HTTP:

```
# Git-Http-Backend (for smart http push , useful with egit )
SetEnv GIT_PROJECT_ROOT /var/scm/git/
SetEnv GIT_HTTP_EXPORT_ALL
#ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
ScriptAliasMatch \
    "(?x)^(git/(.*/(HEAD | \
        info/refs | \
        objects/(info/[^/]+ | \
            [0-9a-f]{2}/[0-9a-f]{38} | \
            pack/pack-[0-9a-f]{40}\\. (pack|idx)) | \
            git-(upload|receive)-pack))$" \
    "/usr/lib/git-core/git-http-backend/$1"

<LocationMatch "^/git/.*/git-receive-pack$">
    #AuthType Basic
    #AuthName "Git Access"
    #Require group committers
</LocationMatch>
```

# XIV - Annexe – Bibliographie, Liens WEB + TP

## 1. Bibliographie et liens vers sites "internet"

<a href="https://git-scm.com/">https://git-scm.com/</a>	site officiel de la technologie git
<a href="https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git">https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git</a>	tutoriel sur GIT

## 2. TP

### 2.1. Installation de git en mode texte

**Installer git en fonction du type d'ordinateur**

**Exemple pour linux debian ou ubuntu :**

```
$ sudo apt update
$ sudo apt upgrade -y
$ sudo apt install git
$ git --version
```

**Exemple pour windows :**

<https://git-scm.com/download> site officiel pour télécharger l'installateur de git

<https://github.com/git-for-windows/git/releases/download/v2.43.0.windows.1/Git-2.43.0-64-bit.exe>

Lancer l'installateur avec choix par défauts

Vérification:

```
git --version
git version 2.43.0.windows.1
```

### 2.2. Configuration locale fondamentale de git

```
git config --global user.name "PrenomNom"
git config --global user.email "prenom.nom@ici_ou_la.fr"
git config --global init.defaultbranch main
git config --list
```

## 2.3. Eventuelle installation de compléments graphiques :

Liste des clients "git" graphiques: <https://git-scm.com/downloads/guis>

**TortoiseGit** (*gratuit , sous windows*)

<https://tortoisegit.org/download/>

<https://download.tortoisegit.org/tgit/2.14.0.0/TortoiseGit-2.14.0.1-64bit.msi>

**GitKraken**

<https://www.gitkraken.com/download/windows64> (*produit payant , version gratuite limitée à 7jours à priori*)

**SmartGit**

-----

+ *plugins "git" des principaux IDE (eclipse, intelliJ , Visual studio code , ...)*

## 2.4. Git élémentaire en mode local

Régler si besoin l'explorateur de fichier windows pour qu'il affiche les extensions des noms de fichiers (ex : *.txt* ) et qu'il affiche les fichiers et répertoires cachés.

*Se créer un répertoire de tp (ex: c:\tp\tp-git )*

*Créer les sous répertoires c:\tp\tp-git\local-git-repositories-devA*

*et c:\tp\tp-git\local-git-repositories-devA\p1*

Au sein du répertoire p1 , créer (par exemple via notepad++) les fichiers simples suivants :

**p1\fl.txt**

```
ligne1
ligne2
```

**p1\l2.txt**

```
abc
def
```

**p1\inutile.bad**

```
fichier pas toujours utile et potentiellement de grande taille
```

Création d'un référentiel local git au sein du projet p1 :

se placer dans *local-git-repositories-devA\p1*

```
git init
```

→ Visualiser via l'explorateur de fichiers , le nouveau sous répertoire caché ".git" .

Générer (dans p1) le souvent indispensable fichier **.gitignore**

```
echo *.bad > .gitignore
```

Visualiser le contenu de **.gitignore** via notepad++ et ajouter y les lignes suivantes :

**.gitignore**

```
*.bad
*.jar
/node_modules
/bin
/target
```

Lancer les commandes suivantes et visualiser bien les effets de chacune d'elles

```
git status
git add *.txt
git add .gitignore
git status
git commit -m "initial"
git status
git log
```

En exercice :

- **ajouter** le fichier *f3.txt* contenant "*f3 c'est 3 fois rien*"
- **modifier** le contenu de *f1.txt* avec quelques majuscules
- lancer **git status** et **git diff** pour afficher les modifications pas encore enregistrées
- lancer **git status** , **git add ...** , **git commit -a -m "..."** de façon à enregistrer tous les changements
- réafficher l'historique via **git log**
- **ajouter** le fichier *f4.txt* contenant "*f4 est un nouveau fichier*"
- **modifier** le contenu de *f1.txt* avec une ligne en plus
- lancer **git status** , **git add ...** , **git commit -a -m "..."** de façon à enregistrer tous les changements
- **réafficher l'historique via git log**  
et copier dans le presse papier l'**id de l'avant dernier commit** associé à "*f3 et modif f1*"  
(exemple : *775c58b7a5f39c2d2d61d1e0bcc7bcbe3f412905* )

Retour en arrière (vers ancienne version sauvegardée) :

```
git checkout 775c58b7a5f39c2d2d61d1e0bcc7bcbe3f412905
```

→ visualiser le nouvel état de p1 (*f4.txt* a disparu et *f1.txt* ne comporte plus le 3ème ligne ajoutée)

Revenir en tête de branche principale :

```
git checkout main (ou bien git checkout master)
git log
```

→ visualiser le nouvel état de p1 (*f4.txt* est revenu et *f1.txt* comporte la 3ème ligne ajoutée)

Associer le tag *v1* à l'avant dernier commit :

```
git tag v1 775c58b7a5f39c2d2d61d1e0bcc7bcbe3f412905
git tag -l
git log
gitk
```

**git checkout tags/v1**

→ visualiser l'absence de *f4.txt*

**git checkout main**

→ visualiser le retour de *f4.txt*



TP facultatif :

- Expérimenter **git show** sur **f1.txt** pour afficher une ancienne version du fichier f1.txt sans effectuer de checkout
- Enregistrer une première fois un fichier notes.txt (via git add et git commit)
- Ajouter de nouvelles lignes au début de notes.txt  
Ajouter également des nouvelles lignes à la fin de notes.txt
- Effectuer un git add avec l'option -p sur notes.txt de manière à ne sélectionner que les lignes ajoutées au début (ce qui revient à désélectionner les lignes ajoutées à la fin)
- effectuer un commit et vérifier l'état de l'enregistrement effectué via git show .

## 2.5. Gestion élémentaire des branches de git

**git branch**

→ par défaut une seule branche master ou **main**

création d'une nouvelle branche b1 :

```
git branch b1
```

**git branch**

Se positionner sur branche b1

```
git checkout b1
```

**git branch**

*main (ou master)*

*\* b1 (avec \* devant branche courante/sélectionnée).*

En exercice :

- ajout de f5.txt et nouveau commit sur branche b1
- ajout de f6.txt et encore nouveau commit sur branche b1
- git log
- se positionner sur branche main ou master (via git checkout)  
→ visualiser absence de f5.txt et f6.txt
- se positionner sur branche b1 (via git checkout)  
→ visualiser retour de f5.txt et f6.txt

Fusion élémentaire de branches (fast forward) :

*Modifications effectuées que sur b1 , aucune modif effectuées sur main → fast forward*

```
git checkout main (ou bien git checkout master)
```

**git merge b1**

git log

**git branch -d b1**      ← suppression ancienne branche b1

git log

## 2.6. Merge de fusion sans conflit

*En exercice :*

- partir du sommet de la branche main
- créer une nouvelle branche temporaire b2
- se placer sur la branche b2
- ajouter le fichier fa.txt
- **ajouter \*\*\* en fin de la première ligne de f1.txt**
- commiter tous ces changements
- se placer sur la branche main
- ajouter fb.txt
- ajouter *dernièreNouvelleligne####* en **fin de f1.txt** (sans changer les premières lignes)
- commiter tous ces changements
- rester sur la branche main
- déclencher une fusion avec la branche b2
- visualiser tous les effets (git log et nouveau commit de fusion, état des fichiers fa.txt , fb.txt , f1.txt)
- supprimer l'ancienne branche temporaire b2

## 2.7. Merge git avec résolution de conflit

- partir du sommet de la branche main
- créer une nouvelle branche temporaire b3
- se placer sur la branche b3
- modifier la première ligne de f2.txt (abc transformé en ABC)
- commiter tous ces changements
- se placer sur la branche main
- modifier différemment la première ligne de f2.txt (abc transformé en a\_b\_c)
- commiter tous ces changements
- rester sur la branche main
- déclencher une fusion avec la branche b3
- visualiser tous le conflit (message de la commande, état du fichier f2.txt )

*Auto-merging f2.txt*

*CONFLICT (content): Merge conflict in f2.txt*

*Automatic merge failed; fix conflicts and then commit the result.*

- **git status**

- On pourrait éventuellement annuler la fusion via `git merge --abort` (si trop de conflits à résoudre). On va préférer résoudre le conflit pour finaliser la fusion.
- visualiser l'état temporaire de ***f2.txt***

```
<<<<<< HEAD
a_b_c
=====
ABC
>>>>>> b3
def
```

- ne garder dans ***f2.txt*** qu'une des versions proposées `a_b_c` ou `ABC` ou un mixte des 2.

```
A_B_C
def
```

- indiquer la fin de la résolution du conflit sur `f2.txt` via la commande **`git add f2.txt`**
- finaliser la fusion via la commande **`git commit -m "merge branch b3"`**
- `git log`
- supprimer l'ancienne branche temporaire `b3`

NB : un futur TP permettra d'expérimenter un merge avec un IDE tel que eclipse/java .

## 2.8. Git élémentaire en mode remote (clone, push, pull)

Se placer dans répertoire de tp (ex: `c:\tp\tp-git`)

Créer les sous répertoires `c:\tp\tp-git\local-git-repositories-devB`

et `c:\tp\tp-git\git-shared-repositories`

NB :

- si le début du chemin "`c:\tp\tp-git`" est différent sur votre PC , il faudra alors adapter ce chemin dans toute la suite de l'énoncé de ce TP .
- `git-shared-repositories` regroupera des référentiels git partagés (éventuellement accessibles à distance selon accès réseaux ...)
- `local-git-repositories-devA` regroupera des référentiels locaux pour le développeur `devA`
- `local-git-repositories-devB` regroupera des référentiels locaux pour le développeur `devB`

Se placer dans le répertoire `c:\tp\tp-git\git-shared-repositories`

et créer le sous répertoire `c:\tp\tp-git\git-shared-repositories\p1.git`

Se placer dans le répertoire `c:\tp\tp-git\git-shared-repositories\p1.git`

et initialiser le référentiel distant/partagé en lançant la commande

```
git init --bare
```

Visualiser le contenu de `p1.git`

Référencement de `p1.git` en tant que version distante/partagée de `local-git-repositories-devA/p1`

Se placer dans `local-git-repositories-devA/p1` et lancer les commandes suivantes :

```
git remote -v
```

(`git remote remove originTpP1` pour supprimer ancienne config en cas d'erreur d'URL)

```
git remote add originTpP1 file:///C:\tp\tp-git\git-shared-repositories\p1.git
```

```
git remote -v
```

NB: `originTpP1` est à voir comme un *alias* pour l'url `file:///C:\tp\tp-git\git-shared-repositories\p1.git`

Première publication partagée depuis `local-git-repositories-devA/p1`:

```
git push --set-upstream originTpP1 main
```

Clonage du référentiel partagé depuis `local-git-repositories-devB` (développeur B) :

Se placer dans `c:\tp\tp-git\local-git-repositories-devB`

```
git clone file:///C:\tp\tp-git\git-shared-repositories\p1.git
```

→ Visualiser le sous répertoire *p1* construit (par clonage)

**cd p1** (depuis ...-devB)

**git remote -v**

origin file:///C:/tp/tp-git/git-shared-repositories/p1.git (fetch)

origin file:///C:/tp/tp-git/git-shared-repositories/p1.git (push)

**git checkout main**

→ visualiser les copies locales des fichiers *f1.txt* , ...

Modif et publication depuis un développeur de l'équipe dev-B ou dev-A

- modifier un des fichiers (ex : nouvelle ligne dans *f1.txt* )
- intégration d'éventuelles modifications distantes via **git pull**
- **commit local**
- intégration d'éventuelles modifications distantes via **git pull**
- publication via **git push**

Rapatriement des modifications enregistrées/partagées depuis autre développeur dev-B ou dev-A

- **git pull**
- visualiser les changements

**Variantes d'URL pour ce TP :**

**file:///C:/tp/tp-git/git-shared-repositories/p1.git** (sur un seul ordinateur)

**file:///Y:/git-shared-repositories/p1.git** où Y: est un lecteur réseau associé à un répertoire partagé par un autre ordinateur via le réseau

**https://github.com/xyz/p1.git** où *p1.git* est un référentiel git partagé géré par l'utilisateur **xyz** ayant un compte sur **github** .

## 2.9. Merge git avec résolution de conflit et branches distantes

Expérimenter des commits/push/pull avec conflits à résoudre entre deux utilisateurs travaillant sur un même fichier d'un même projet (ex : *p1*).

On pourra effectuer se Tp en simulant deux utilisateurs sur le même ordinateur (configuration du TP précédent) ou bien en configurant plusieurs comptes **github** ou **gitlab** ou **gitbucket** .

Bonnes pratiques (en général) :

- d'abord un commit sur branche locale
- pull avant merge ou push

## 2.10. github (ou autre) en mode distant et réel avec un accès restreint sur un référentiel préparé (compte "formateur")

<https://github.com/didier-tp/git-classroom.git> est l'URL d'un référentiel distant préparé pour quelques Tps .

Ce référentiel (de type "sandbox" sans rien d'important à l'intérieur) appartient à l'utilisateur didier-tp et l'on peut tout de même y accéder en lecture écriture via un jeton d'accès personnel .

Token (révocable) permettant d'accéder qu'à ce référentiel de Tp :

```
github_pat_11AHMRVJQ0WZrJ1X5kYcdm_GohIS6u6uNisLW4ESk399AI5Q6ZNVqa7WwRIC2JnQxFCDNVHCQ6HejdgfV
```

URL adaptée avec le token (*origin* à changer après clonage):

[https://didier-tp:the\\_token\\_value@github.com/didier-tp/git-classroom.git](https://didier-tp:the_token_value@github.com/didier-tp/git-classroom.git)

[https://didier-tp:github\\_pat\\_11AHMRVJQ0WZrJ1X5kYcdm\\_GohIS6u6uNisLW4ESk399AI5Q6ZNVqa7WwRIC2JnQxFCDNVHCQ6HejdgfV@github.com/didier-tp/git-classroom.git](https://didier-tp:github_pat_11AHMRVJQ0WZrJ1X5kYcdm_GohIS6u6uNisLW4ESk399AI5Q6ZNVqa7WwRIC2JnQxFCDNVHCQ6HejdgfV@github.com/didier-tp/git-classroom.git)

1. Se préparer un répertoire **C:\tp\tp-git-classroom** (ou un équivalent)
2. Selon les consignes du formateur, dans ce répertoire , créer un fichier **url.txt** pour y mettre au point (par concaténation adaptée/ajustée) l'url complète avec le jeton d'accès valide à la date du TP.
3. Créer un sous répertoire "**as\_dev\_x**" (dans *C:\tp\tp-git-classroom*) , ce placer dedans et effectuer un clone du référentiel <https://github.com/didier-tp/git-classroom.git> avec idéalement l'url complétée avec le jeton d'accès.
4. Seulement si l'URL comportant le jeton est erronée et quelle a besoin d'être mise à jour , on pourra alors lancer la commande suivante (à adapter avec un jeton valide) :  
**git remote set-url origin** [https://didier-tp:bonne\\_valeur\\_jeton@github.com/didier-tp/git-classroom.git](https://didier-tp:bonne_valeur_jeton@github.com/didier-tp/git-classroom.git)
5. Se placer dans le répertoire **C:\tp\tp-git-classroom\as\_dev\_x\git-classroom**  
 Lancer ensuite un enchaînement de modifications et de commandes selon cette logique :
  - **git branch -a** (pour voir toutes les branches "remotes")
  - **git checkout -b m-dev origin/m-dev** (créer branche locale "m-dev" pistant la branche distante origin/m-dev)
  - **git branch -vv** (affichage avec détails sur pistages)
  - **git config --local user.email "....."**  
**git config --local user.name "DidierTp\_dev\_x"**

- Depuis un éditeur quelconque (notepad++ , ...) ou bien un IDE quelconque (eclipse ou intelliJ ou VisualStudioCode ou ...), modifier le contenu du fichier *notes.txt* (à peut être créer s'il n'existe pas encore)  
`essais_en_solo/s1_ou_..._ou_s15/notes.txt`
- répercuter la mise à jour au niveau du référentiel partagé (add,commit,push, ...)
- visualiser le contenu partagé de `essais_en_solo/s1_ou_..._ou_s15/notes.txt` sur la branche *m-dev* du référentiel sur le site de github via un navigateur internet (<https://github.com/didier-tp/git-classroom>)

#### Suite du tp (selon le contexte de l'environnement de TP) :

- Si "suite du Tp réalisée seul" , alors refaire toutes les manipulations précédentes au sein d'un sous répertoire "*as\_dev\_y*" de *C:\tp\tp-git-classroom* .
- Si "suite du Tp réalisée en groupe" , alors "chaque stagiaire" dispose d'un clone sur son poste dans un répertoire *C:\tp\tp-git-classroom\as\_dev\_x\git-classroom* (ou un équivalent).

#### Suite du Tp (travail en pseudo-équipe réelle ou simulée) :

- effectuer des "*séquences modif/add/commit/push*" de tous les cotés (dev\_x, dev\_y , ...) de différentes manières :
  - avec concertation et pull préalable (chacun son tour en attendant l'autre)
  - sans pull préalable
  - sans concertation au début et concertation lors des "merges\_avec\_conflits"
  - ...

NB : on pourra par exemple effectuées des modifications sur le fichier

`essais_en_groupes/g1_ou_..._ou_g10/notes.txt` (à créer si pas encore existant)

## 2.11. Git avec résolution de conflit depuis un IDE (Tp facultatif)

1. Créer un référentiel git comportant un petit projet java/maven très simple ou bien utiliser le référentiel du Tp précédent ( <https://github.com/didier-tp/git-classroom.git> et un clone tel que *C:\tp\tp-git-classroom\as\_dev\_x\git-classroom*)
2. Charger ce projet depuis un IDE (ex : eclipse , intelliJ , visual studio code, ...)
3. Expérimenter les opérations git basiques (commit, push, pull)
4. Expérimenter les résolutions de conflits depuis l'IDE

## 2.12. Git en mode http (via github ou autre , tp facultatif)

1. Se créer un compte gratuit sur github ou gitlab (une adresse email devra être vérifiée) et mémoriser les infos d'authentification (username/password ou token ou ...)
2. Créer un nouveau référentiel distant (vide au départ)



3. Publier un projet local vers le référentiel distant (git remote ... , git push ...)
4. Cloner le référentiel distant ailleurs (git clone , ....)
5. effectuer une série de changements (commit , push , pull, ...) sans conflit puis avec conflit soit en ligne de commande , soit via tortoiseGit , soit via un IDE (ex : eclipse).
6. On pourra éventuellement autoriser d'autres développeurs amis à effectuer des publications sur notre référentiel partagé github ou gitlab .

## 2.13. Expérimentation de "rebase" sur branches locales

On pourra par exemple réutiliser le référentiel p1.git  
et l'un de ses clones (ex : *C:\tp\tp-git\local-git-repositories-devA\p1*)

- partir de la branche "main"
- créer une nouvelle branche b-temp
- effectuer deux ajouts/commits sur la branche b-temp
- revenir sur la branche "main"
- effectuer deux ajouts/commits différents (sans conflits) sur la branche "main"
- se placer sur la branche **b-temp**
- afficher l'historique de b-temp via la commande **git log --oneline**
- *rebase la branche courante b-temp sur la tête de la branche main :*  
**git rebase main**
- réafficher le nouvel historique de b-temp (changé) via la commande **git log --oneline**
- se replacer sur la branche main
- afficher l'historique de main via la commande **git log --oneline**
- faire avancer la branche main vers la fin de la branche b-temp rebasée de manière à absorber les changements via un merge de type fast-forward :  
**git merge b-temp**
- réafficher l'historique de main via la commande **git log --oneline**
- **supprimer** la branche **b-temp**

## 2.14. Expérimentation de "cherry-pick" sur branches locales

On pourra par exemple réutiliser le référentiel p1.git  
et l'un de ses clones (ex : *C:\tp\tp-git\local-git-repositories-devA\p1*)

- partir de la branche "main"
- créer une nouvelle branche b-dev
- effectuer un ajout/commit sur la branche b-dev et afficher l'historique
- revenir sur la branche "main"
- effectuer une modification de fichier existant (ex: f1.txt avec ligne supplémentaire HOTFIX) et un commit ayant pour message "**hotfix\_f1**" sur la branche "main"
- copier dans le presse-papier l'identifiant au format "sha1" du commit "hotfix\_f1" (exemple *dc3ae11030331cb69d7463d7a0a2c0a6153a9447*) que l'on peut trouver via une commande comme **git log -n 3**

- se replacer sur la branche **b-dev**
- **git cherry-pick** <shaDuCommitHotfixF1>  
(exemple: **git cherry-pick** *dc3ae11030331cb69d7463d7a0a2c0a6153a9447*)
- réafficher le nouvel historique de b-dev (changé) via la commande **git log --oneline**
- maintenant que le hotfix (correction de bug d'urgence sur "main") a été rapatrié sur la branche b-dev, on peut tranquillement continuer à travailler sur la branche b-dev avec une bonne prise en compte de ce correctif .

## 2.15. Expérimentation de "stash"

On pourra par exemple réutiliser le référentiel p1.git et l'un de ses clones (ex : *C:\tp\tp-git\local-git-repositories-devA\p1*)

- partir de la branche "**main**"
- effectuer un **git push**
- créer une nouvelle branche "**bb**" et rester sur la branche "main".
- Ajouter un fichier **ficzz.txt** (avec contenu **mm\_zz**) et modifier un fichier existant (ex : **f2.txt**) sur la branche main et commiter cela
- se placer sur la branche "**bb**" créée précédemment.
- ajouter un nouveau fichier de nom **ficzz.txt** (avec contenu différent : **bb\_zz**) sans commiter ce changement
- tenter de revenir sur la branche "main" (message d'erreur et on est toujours sur branche bb , à visualiser via "git branch")
- **git stash -u**
- **visualiser la disparition du fichier ficzz.txt (sur branche bb) dans le répertoire de travail**
- revenir sur la branche "**main**" et ré-effectuer un petit changement sur f2.txt (à commiter)
- effectuer un "**git push**" sur la branche "main"
- revenir sur la branche "**bb**" (où le fichier **ficzz.txt** n'apparaît toujours pas)
- **git stash list**
- **git stash pop**
- visualiser la **réapparition** du fichier **ficzz.txt** dans le répertoire de travail
- commiter l'ajout du fichier **ficzz.txt**
- revenir sur la branche "main" , fusionner "bb" et détruire "bb".
- **git push** et **git log**

## 2.16. Autres expérimentations libres

- git squash
- git patch
- ...