



# GIT ET GITHUB

CONCEPTS ET COMMANDES FONDAMENTALES ET  
AVANCÉES

LUCA ITALO MANSUTTI  
[luca.mansutti7@gmail.com](mailto:luca.mansutti7@gmail.com)

# TABLE DES MATIERES

<b>PARTIE 1 : CONCEPTS FONDAMENTAUX.....</b>	<b>1</b>
<b>1 CONFIGURATION DE GIT .....</b>	<b>1</b>
<b>1.1 NIVEAUX DE CONFIGURATIONS .....</b>	<b>1</b>
<b>1.2 AJOUTER DES CONFIGURATIONS.....</b>	<b>1</b>
<b>1.3 SUPPRIMER DES CONFIGURATIONS .....</b>	<b>1</b>
1.3.1 Une clé spécifique .....	1
1.3.2 Toutes les occurrences d'une clé .....	1
1.3.3 Une section entière .....	1
<b>1.4 CONSULTER LES CONFIGURATIONS.....</b>	<b>2</b>
1.4.1 Globales.....	2
1.4.2 Locales .....	2
1.4.3 Valeur unique.....	2
<b>1.5 CONFIGURATIONS SPÉCIFIQUES .....</b>	<b>2</b>
1.5.1 S'identifier à git globalement .....	2
1.5.2 Changer le nom de la branche par défaut .....	2
1.5.3 Choisir l'éditeur par défaut .....	2
1.5.4 Choisir entre merge/rebase lors d'un pull .....	2
1.5.5 Activer ou non le RERERE (voir chapitre 17) .....	2
<b>2 REPOSITORY.....</b>	<b>3</b>
<b>2.1 INITIALISATION DU REPOSITORY.....</b>	<b>3</b>
2.1.1 Créer un dossier et des fichiers .....	3
2.1.2 Initialiser le dossier comme dépôt Git.....	3
<b>2.2 TYPES D'ÉTATS DE FICHIERS .....</b>	<b>3</b>
<b>2.3 STATUS .....</b>	<b>3</b>
<b>2.4 STAGES.....</b>	<b>3</b>
2.4.1 Ajouter un fichier au staging .....	3
2.4.2 Retirer du staging et arrêter de suivre le fichier .....	3
2.4.3 Annuler un ajout au staging.....	3
<b>2.5 COMMIT.....</b>	<b>4</b>
2.5.1 Faire un commit avec un message.....	4
2.5.2 Modifier le message du dernier commit en cas d'erreur .....	4
<b>3 GIT LOG .....</b>	<b>4</b>
<b>3.1 INFORMATIONS FOURNIES PAR GIT LOG.....</b>	<b>4</b>
<b>3.2 LOG COMMAND.....</b>	<b>4</b>
<b>3.3 OPTIONS (FLAGS) DE LA COMMANDE GIT LOG.....</b>	<b>4</b>
3.3.1 Limiter l'affichage à un certain nombre de commits .....	4
3.3.2 Affichage simplifié en une ligne par commit .....	4
3.3.3 Afficher l'historique de toutes les branches (pour voir les merges) .....	5
3.3.4 Affichage graphique des commits .....	5

3.3.5 Afficher les commits parents .....	5
3.3.6 Afficher les commits d'une branche distante spécifique .....	5
3.3.7 Afficher les informations des fichiers modifiés des commits .....	5
3.3.8 Afficher les différences entre tous les commits.....	5
3.3.9 Afficher toutes les références (pointeurs vers les commits).....	5
3.3.10 Limiter les logs à certaines dates.....	5
3.3.11 Afficher les commits d'un auteur en particulier .....	5
<b>4 FONCTIONNEMENT INTERNE .....</b>	<b>6</b>
<b>4.1 BASES DU FONCTIONNEMENT INTERNE (PLUMBING) .....</b>	<b>6</b>
<b>4.2 EXPLORATION DES OBJETS.....</b>	<b>6</b>
4.2.1 Trouver le hash d'un commit .....	6
4.2.2 Lister le contenu du dossier .git/objects .....	6
<b>4.3 COMMANDE CAT-FILE .....</b>	<b>6</b>
4.3.1 Afficher le contenu d'un commit.....	6
<b>4.4 TREES &amp; BLOBS .....</b>	<b>7</b>
4.4.1 Exploration d'un objet commit .....	7
4.4.2 Examiner un objet Tree .....	7
4.4.3 Examiner un objet Blob.....	7
<b>4.5 COMMITS MULTIPLES (PARENTS) .....</b>	<b>8</b>
<b>4.6 STOCKAGE DES DONNÉES .....</b>	<b>8</b>
4.6.1 Comment git stocke les données .....	8
4.6.2 Optimisation du stockage .....	8
<b>5 BRANCHING.....</b>	<b>9</b>
<b>5.1 UTILITÉ DES BRANCHES .....</b>	<b>9</b>
<b>5.2 FONCTIONNEMENT INTERNE .....</b>	<b>9</b>
<b>5.3 OPÉRATIONS SUR LES BRANCHES .....</b>	<b>10</b>
5.3.1 Afficher toutes les branches du repository .....	10
5.3.2 Créer une branche sans y basculer directement .....	10
5.3.3 Créer et basculer directement sur la nouvelle branche.....	10
5.3.4 Basculer entre les branches .....	10
5.3.5 Renommer une branche.....	10
5.3.6 Supprimer une branche.....	10
<b>5.4 AJOUT D'UN COMMIT DANS UNE NOUVELLE BRANCHE .....</b>	<b>10</b>
<b>5.5 LE FICHIER GIT ET LES BRANCHES.....</b>	<b>11</b>
5.5.1 Branches et leurs références .....	11
5.5.2 Commandes pour examiner les références.....	11
5.5.2.1     Afficher le hash du commit d'une branche .....	11
5.5.2.2     Afficher toutes les branches .....	11
5.5.2.3     Afficher toutes les branches et leur commits .....	11
<b>6 MERGE.....</b>	<b>12</b>
<b>6.1 POURQUOI AVOIR PLUSIEURS BRANCHES ? .....</b>	<b>12</b>
<b>6.2 MERGE COMMAND .....</b>	<b>12</b>
<b>6.3 MERGE COMMITS .....</b>	<b>12</b>

6.3.1 Fonctionnement des merge commits .....	12
<b>6.4 MERGE LOG .....</b>	<b>13</b>
6.4.1 Commande de vérification du merge .....	13
6.4.2 Explications des hash avec des parents .....	13
6.4.2.1 Commit de fusion (Merge Commit) .....	13
6.4.2.2 Commits de la branche add_classics.....	13
6.4.2.3 Commits de la branche main : .....	13
<b>6.5 FAST FORWARD MERGE / COMMIT .....</b>	<b>14</b>
<b>7 REBASE .....</b>	<b>15</b>
<b>7.1 VISUALISATION D'UN REBASE .....</b>	<b>15</b>
<b>7.2 EXÉCUTER UN REBASE.....</b>	<b>15</b>
7.2.1 Se positionner dans la branche dans laquelle on veut ramener les commits.....	15
7.2.2 Commande Rebase.....	15
<b>7.3 MERGE VS REBASE .....</b>	<b>16</b>
7.3.1 Merge.....	16
7.3.2 Rebase : .....	16
<b>7.4 QUAND UTILISER REBASE ? .....</b>	<b>16</b>
<b>7.5 QUAND ÉVITER REBASE ? .....</b>	<b>16</b>
<b>8 ANNULATION ET RESTAURATION .....</b>	<b>17</b>
<b>8.1 RESET COMMAND .....</b>	<b>17</b>
8.1.1 Soft reset.....	17
8.1.2 Hard reset .....	17
.....	17
8.1.3 Dangers du hard reset.....	17
<b>8.2 RESTORE COMMAND .....</b>	<b>18</b>
8.2.1 Restaurer des fichiers du dernier commit dans le répertoire local .....	18
8.2.2 Annuler un ajout au stage (unstage) : .....	18
<b>8.3 REVERT COMMAND.....</b>	<b>18</b>
<b>8.4 RÉCAPITULATIF .....</b>	<b>18</b>
<b>9 REMOTE.....</b>	<b>19</b>
<b>9.1 GÉRER DES DÉPÔTS DISTANTS.....</b>	<b>19</b>
<b>9.2 AJOUTER UN REMOTE.....</b>	<b>19</b>
<b>9.3 FETCH .....</b>	<b>19</b>
9.3.1 Fetch command .....	19
9.3.2 Fetch n'intègre pas les commits du dépôt distant.....	19
9.3.3 Conclusion.....	19
<b>9.4 LOG REMOTE .....</b>	<b>20</b>
9.4.1 Afficher l'Historique d'une Branche Distante avec git log .....	20
<b>9.5 MERGE REMOTE .....</b>	<b>20</b>
9.5.1 Fusionner une Branche Distante dans une Branche Locale .....	20
9.5.2 Merge remote command.....	20
<b>10 GITHUB .....</b>	<b>21</b>

<b>10.1</b>	<b>GITHUB REPOSITORY .....</b>	<b>21</b>
10.1.1	GitHub .....	21
10.1.2	GitHub CLI .....	21
<b>10.2</b>	<b>GESTION DU DÉPÔT SUR GITHUB.....</b>	<b>22</b>
10.2.1	Créer un dépôt sur GitHub .....	22
10.2.2	S'authentifier avec GitHub CLI (gh) .....	22
<b>10.3</b>	<b>REMOTE ADD COMMAND .....</b>	<b>23</b>
10.3.1	Lier le dépôt local au dépôt distant (remote) sur GitHub.....	23
10.3.2	Vérifier la liaison.....	23
<b>10.4</b>	<b>GIT PUSH .....</b>	<b>23</b>
10.4.1	Options alternatives .....	23
<b>10.5</b>	<b>GIT PULL .....</b>	<b>23</b>
<b>10.6</b>	<b>PULL REQUEST .....</b>	<b>24</b>
10.6.1	Fonctionnement d'un pull request .....	24
10.6.2	Exemple de pull request .....	24
10.6.2.1	Partie "Envoyer les modifications" .....	24
10.6.2.2	Partie "Création d'une Pull Request" .....	25
<b>11</b>	<b>GITIGNORE .....</b>	<b>26</b>
<b>11.1</b>	<b>FICHIER .GITIGNORE .....</b>	<b>26</b>
11.1.1	Exemple d'utilisation .....	26
<b>11.2</b>	<b>NESTED .GITIGNORE .....</b>	<b>26</b>
11.2.1	Fonctionnement .....	26
11.2.2	Exemple concret .....	27
<b>11.3</b>	<b>PATTERNS.....</b>	<b>27</b>
11.3.1	Wildcard (*). ....	27
11.3.2	Double wildcard (**) .....	27
11.3.3	Rooted patterns (/) .....	27
11.3.4	Négation .....	28
11.3.5	Commentaires.....	28
<b>11.4</b>	<b>QUE METTRE DANS LE .GITIGNORE ? .....</b>	<b>28</b>
11.4.1	Fichiers générés .....	28
11.4.2	Dépendances .....	28
11.4.3	Fichiers personnels ou spécifiques à l'éditeur .....	29
11.4.4	Informations sensibles ou dangereuses .....	29
<b>12</b>	<b>README.MD .....</b>	<b>30</b>
<b>12.1</b>	<b>QU'EST-CE QU'UN FICHIER README ? .....</b>	<b>30</b>
<b>12.2</b>	<b>SYNTAXE D'UN FICHIER .MD.....</b>	<b>30</b>
12.2.1	Titres .....	30
12.2.2	Gras et italique .....	30
12.2.3	Listes .....	30
12.2.4	Liens et images .....	31
12.2.5	Bloc de code avec coloration syntaxique .....	31
12.2.6	Tableaux .....	31
12.2.7	Saut de ligne .....	31

12.2.8 Citations .....	31
12.2.9 Séquences d'échappement.....	31
<b>12.3 EXEMPLE COMPLET DE FICHIER .MD.....</b>	<b>32</b>
12.3.1 Code markdown .....	32
12.3.2 Affichage sur GitHub .....	33
<b>12.4 FICHIER README DE PROFIL GITHUB .....</b>	<b>33</b>
12.4.1 Marche à suivre .....	33
12.4.2 Exemple de présentation de profil GitHub.....	34
<b>PARTIE 2 : CONCEPTS AVANCÉS .....</b>	<b>1</b>
<b>13 FORK .....</b>	<b>35</b>
<b>13.1 QU'EST-CE QU'UN FORK ?.....</b>	<b>35</b>
<b>13.2 COMMENT FORK UN REPOSITORY .....</b>	<b>35</b>
<b>13.3 EXEMPLE RÉEL DE TRAVAIL COLLABORATIF .....</b>	<b>35</b>
13.3.1 Cloner le dépôt "forké".....	35
13.3.2 Créer une nouvelle branche pour les nouvelles fonctionnalités .....	35
13.3.3 Ajouter et commit les modifications .....	35
13.3.4 Pousser les changements vers GitHub .....	35
13.3.5 Créer un pull request .....	35
<b>14 REFLOG .....</b>	<b>36</b>
<b>14.1 HEAD.....</b>	<b>36</b>
14.1.1 Voir où pointe HEAD .....	36
<b>14.2 REFLOG COMMAND .....</b>	<b>36</b>
14.2.1 Explications de reflog.....	36
14.2.2 Fonctionnement de reflog.....	36
14.2.3 Log VS Reflog.....	36
<b>14.3 RÉCUPÉRATION DU CONTENU D'UN FICHIER SUPPRIMÉ .....</b>	<b>37</b>
14.3.1 Étapes à suivre pour récupérer le contenu d'un fichier supprimé .....	37
14.3.1.1 Vérifier l'historique avec reflog .....	37
14.3.1.2 Explorer le commit avec git cat-file -p.....	37
14.3.1.3 Explorer le tree .....	37
14.3.1.4 Examiner le blob pour retrouver le contenu du fichier .....	37
<b>14.4 RÉCUPÉRATION AMÉLIORÉE AVEC MERGE .....</b>	<b>38</b>
14.4.1 Merge avec un commitish .....	38
<b>15 MERGE CONFLICTS.....</b>	<b>39</b>
<b>15.1 INTRODUCTION AUX CONFLITS .....</b>	<b>39</b>
15.1.1 Exemple de conflit .....	39
<b>15.2 RÉSOUDRE UN CONFLIT .....</b>	<b>40</b>
15.2.1 Localiser le conflit.....	40
15.2.2 Éditer les fichier et résoudre les conflits .....	40
15.2.3 Ajouter et finaliser la fusion .....	40
15.2.4 Vérifier le merge.....	40
<b>15.3 ARRÊTER UN MERGE LORS D'UN CONFLIT .....</b>	<b>41</b>
<b>15.4 OUTILS DE RÉSOLUTION DE CONFLITS INTÉGRÉE À GIT .....</b>	<b>41</b>

15.4.1 "Ours" vs "Theirs" (Notre vs Leurs) .....	41
15.4.2 Checkout merge command .....	41
<b>16 REBASE CONFLICTS .....</b>	<b>42</b>
<b>16.1 EXEMPLE DE REBASE CONFLICTS .....</b>	<b>42</b>
16.1.1 Scénario : .....	42
16.1.2 L'état "detached HEAD" .....	42
16.1.3 Checkout rebase command.....	43
16.1.4 Terminer le rebase .....	43
<b>17 RERERE (REUSE RECORDED RESOLUTION).....</b>	<b>44</b>
<b>17.1 QU'EST-CE QUE GIT RERERE.....</b>	<b>44</b>
17.1.1 Activer / désactiver rerere.....	44
17.1.2 Supprimer le dossier rerere .....	44
<b>17.2 EXEMPLE D'UTILISATION DE RERERE.....</b>	<b>44</b>
17.2.1 Premier rebase de main sur favs (Entrée dans la zone de conflit).....	44
17.2.2 Résolution des conflits et fermeture du rebase.....	45
17.2.3 Rebase de main sur favs2 (Résolution automatique) .....	45
17.2.4 Fermeture du rebase .....	45
<b>17.3 ANNULER UN COMMIT (ALORS QU'IL FALLAIT --CONTINUE).....</b>	<b>45</b>
17.3.1 Problème courant.....	45
17.3.2 Annuler un commit accidentel.....	45
<b>18 SQUASH.....</b>	<b>46</b>
<b>18.1 QU'EST-CE QUE LE SQUASHING ?.....</b>	<b>46</b>
<b>18.2 COMMENT FAIRE UN SQUASH ? .....</b>	<b>46</b>
18.2.1 Démarrer un rebase interactif .....	46
18.2.2 Choisir les commits à squash .....	46
<b>18.3 OVERWRITE MAIN .....</b>	<b>47</b>
18.3.1 Supprimer la branche main .....	47
18.3.2 Renommer la branche temporaire par main .....	47
<b>18.4 DANGERS DU SQUASHING .....</b>	<b>47</b>
<b>18.5 SQUASHING PULL REQUESTS .....</b>	<b>48</b>
18.5.1 Résumé du flux de travail.....	48
<b>18.6 FORCE PUSH.....</b>	<b>48</b>
18.6.1 Qu'est-ce qu'un force push .....	48
18.6.2 Force push command .....	48
<b>19 STASH.....</b>	<b>49</b>
<b>19.1 A QUOI SERT LE STASHING ?.....</b>	<b>49</b>
<b>19.2 STASH COMMAND .....</b>	<b>49</b>
19.2.1 Créer un stash .....	49
19.2.2 Lister les stashes.....	49
19.2.3 Stash Pop .....	49
19.2.4 Stash Apply .....	49
19.2.5 Stash drop .....	49

<b>20 DIFF.....</b>	<b>50</b>
<b>20.1 DIFF COMMAND .....</b>	<b>50</b>
20.1.1 Comparer l'état actuel et le dernier commit.....	50
20.1.2 Comparer un commit précédent et l'état actuel .....	50
20.1.3 Comparer deux commits spécifiques .....	50
20.1.4 Comparer l'index (zone de staging) avec la dernière validation .....	50
20.1.5 Comparer l'arbre de travail avec l'index .....	50
<b>EXEMPLE D'UTILISATION DE DIFF .....</b>	<b>50</b>
<b>20.2 BLAME COMMAND .....</b>	<b>51</b>
20.2.1 Exemple de sortie de git blame .....	51
<b>21 CHERRY PICK .....</b>	<b>52</b>
<b>21.1 QUE FAIT LA COMMANDE CHERRY-PICK ? .....</b>	<b>52</b>
<b>21.2 CHERRY-PICK COMMAND.....</b>	<b>52</b>
<b>22 BISECT .....</b>	<b>53</b>
<b>22.1 A QUOI SERT LE BISECT ? .....</b>	<b>53</b>
<b>22.2 MARCHE À SUIVRE DU BISECT.....</b>	<b>53</b>
22.2.1 Commencer le bisect .....	53
22.2.2 Marquer un commit fonctionnel (bon commit).....	53
22.2.3 Marquer un commit problématique (mauvais commit).....	53
22.2.4 Tester le commit intermédiaire.....	53
22.2.5 Déclarer le résultat du test .....	53
22.2.6 Répéter les 2 étapes précédentes .....	53
22.2.7 Terminer la session de bisect.....	53
<b>23 WORKTREE .....</b>	<b>54</b>
<b>23.1 QU'EST-CE QU'UN WORKTREE ? .....</b>	<b>54</b>
<b>23.2 TYPES DE WORKTREES .....</b>	<b>54</b>
23.2.1 Main Worktree .....	54
23.2.2 Linked Worktree .....	54
<b>23.3 WORKTREE COMMANDS .....</b>	<b>54</b>
23.3.1 Lister tous les worktrees associés à un repository.....	54
23.3.2 Créer un linked worktree .....	54
23.3.3 Afficher le contenu du fichier .git d'un linked worktree .....	54
<b>23.4 PAS DE BRANCHES PARTAGÉES ENTRE WORKTREES ! .....</b>	<b>55</b>
<b>23.5 TRACKING DES LINKED WORKTREES.....</b>	<b>55</b>
<b>23.6 MAIN ET LINKED WORKTREES : SYSTÈME UNIQUE .....</b>	<b>55</b>
<b>23.7 SUPPRIMER UN WORKTREE .....</b>	<b>56</b>
23.7.1 Supprimer un worktree ne supprime pas la branche .....	56
23.7.2 Remove command.....	56
23.7.3 Prune command .....	56
<b>24 TAGS .....</b>	<b>57</b>
<b>24.1 QU'EST-CE QU'UN TAG ? .....</b>	<b>57</b>
<b>24.2 TAG COMMANDS .....</b>	<b>57</b>

24.2.1 Lister les tags existants.....	57
24.2.1.1 Localement.....	57
24.2.1.2 Sur un dépôt distant .....	57
Afficher les détails d'un tag .....	57
24.2.2 Créer un tag annoté.....	57
24.2.3 Supprimer un tag .....	57
24.2.3.1 Localement.....	57
24.2.3.2 Sur le dépôt distant .....	57
24.2.4 Pousser les tags vers le dépôt distant .....	57
<b>FORMULAIRE DES COMMANDES .....</b>	<b>30</b>
<b>25   COMMANDES FONDAMENTALES .....</b>	<b>57</b>
<b>26   COMMANDES AVANCÉES .....</b>	<b>60</b>
<b>SOURCES .....</b>	<b>61</b>
<b>27 REFERENCES .....</b>	<b>62</b>



---

## PARTIE 1 : CONCEPTS FONDAMENTAUX

---

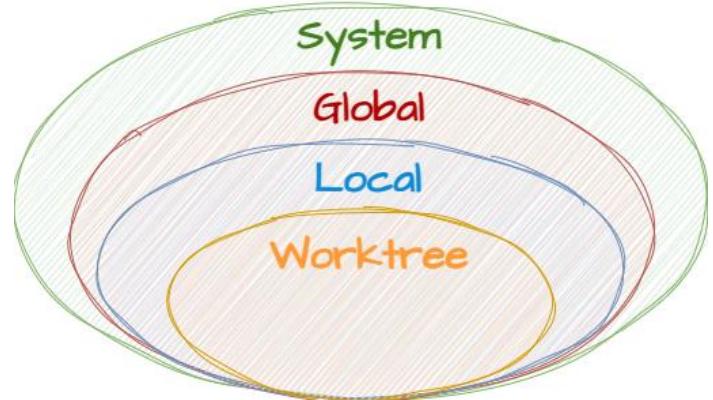


# 1 CONFIGURATION DE GIT

## 1.1 NIVEAUX DE CONFIGURATIONS

Les configurations sont hiérarchisées, du plus général au plus spécifique :

1. **Système** : `/etc/gitconfig`  
Impacte tous les utilisateurs du système.
2. **Global** : `~/.gitconfig`  
Impacte tous les dépôts de l'utilisateur.
3. **Local** : `.git/config`  
Impacte uniquement le dépôt courant.
4. **Worktree** : `.git/config.worktree`  
Spécifique à une branche ou partie d'un dépôt.



**Remarque** : Les configurations **locales** écrasent les configurations **globales**

```
worktree > local > global > system
```

## 1.2 AJOUTER DES CONFIGURATIONS

```
git config [--add] [--global | --local] <SECTION>.<KEYNAME> "VALEUR"
```

Exemple : `git config --add --global netflix.ceo "Luca Mansutti"`

## 1.3 SUPPRIMER DES CONFIGURATIONS

### 1.3.1 UNE CLÉ SPÉCIFIQUE

```
git config --unset [--global] <SECTION>.<KEYNAME>
```

### 1.3.2 TOUTES LES OCCURRENCES D'UNE CLÉ

```
git config --unset-all [--global] <SECTION>.<KEYNAME>
```

### 1.3.3 UNE SECTION ENTIÈRE

```
git config --remove-section [--global] <SECTION>
```

## 1.4 CONSULTER LES CONFIGURATIONS

### 1.4.1 GLOBALES

```
git config --list          # Toutes les configurations globales
```

### 1.4.2 LOCALES

```
git config --list --local  # Configurations locales uniquement  
cat .git/config           # Affiche les configurations locales
```

### 1.4.3 VALEUR UNIQUE

```
git config --get <SECTION>.<KEYNAME>      # Valeur associée à la clé
```

## 1.5 CONFIGURATIONS SPÉCIFIQUES

### 1.5.1 S'IDENTIFIER À GIT GLOBALEMENT

```
git config [--add] --global user.name "USERNAME"  
git config [--add] --global user.email "EMAIL"
```

### 1.5.2 CHANGER LE NOM DE LA BRANCHE PAR DÉFAUT

```
git config [--add] --global init.defaultBranch <BRANCH_NAME>
```

### 1.5.3 CHOISIR L'ÉDITEUR PAR DÉFAUT

```
git config --global core.editor "EDITOR_NAME"
```

### 1.5.4 CHOISIR SI GIT PREND EN COMPTE LA CASSE

```
git config --global core.ignorecase [TRUE | FALSE]
```

**Note :** De base Git ne prend pas en compte la casse dans les fichiers, c'est embêtant car les changements de nom de fichiers (min/maj) ne sont pas pris en compte, on peut donc lui dire d'ignorer la casse :

### 1.5.5 CHOISIR ENTRE MERGE/REBASE LORS D'UN PULL

Si vous préférez que `git pull` utilise le rebase au lieu de merge :

```
git config --global pull.rebase [TRUE | FALSE]
```

### 1.5.6 ACTIVER OU NON LE RERERE (VOIR CHAPITRE 17)

```
git config --global rerere.enabled true
```

## 2 REPOSITORY

### 2.1 INITIALISATION DU REPOSITORY

#### 2.1.1 CRÉER UN DOSSIER ET DES FICHIERS

```
mkdir <NOM_DU_DOSSIER>
cd <NOM_DU_DOSSIER/>
touch <NOM_DU_FICHIER>
```

#### 2.1.2 INITIALISER LE DOSSIER COMME DÉPÔT GIT

```
git init
```

**Note :** Le dossier .git est créé pour contenir toutes les informations de suivi. Supprimer ce dossier arrête la liaison avec Git

### 2.2 TYPES D'ÉTATS DE FICHIERS

- **untracked** : non suivis par git (ignoré)
- **staged** : prêt à être commit (en attente)
- **committed** : déjà commit (sauvé dans l'historique des commit du repo)

### 2.3 STATUS

```
git status
```

**Note :** Cette commande permet d'afficher l'état des fichiers d'un repository

### 2.4 STAGES

#### 2.4.1 AJOUTER UN FICHIER AU STAGING

```
git add <PATH | PATTERN | . | FILE>
```

**Note :** Cette commande permet de choisir les fichiers à inclure pour le prochain commit

#### 2.4.2 RETIRER DU STAGING ET ARRÊTER DE SUIVRE LE FICHIER

```
git rm --cached <FILE>
```

**Note :** Le fichier ne sera plus suivi par Git

#### 2.4.3 ANNULER UN AJOUT AU STAGING

```
git restore --staged <FILE>
```

**Note :** Le fichier reste suivi par Git

## 2.5 COMMIT

### 2.5.1 FAIRE UN COMMIT AVEC UN MESSAGE

```
git commit -m "MESSAGE"
```

**Note :** Le commit crée une snapshot du repository à un moment donné (historique). Cela crée une sauvegarde des fichiers à ce moment-là.

Si les fichiers sont déjà suivis, on peut faire un commit et un add en une seule ligne :

```
git commit -am "MESSAGE"
```

### 2.5.2 MODIFIER LE MESSAGE DU DERNIER COMMIT EN CAS D'ERREUR

```
git commit --amend -m "MESSAGE"
```

## 3 GIT LOG

### 3.1 INFORMATIONS FOURNIES PAR GIT LOG

La commande `git log` affiche l'historique des commits du projet

- **Auteur** : Qui a réalisé le commit.
- **Date** : Quand le commit a été effectué.
- **Modifications** : Les fichiers ou parties modifiées.
- **Commit hash** : Un identifiant unique (longue chaîne de caractères) qui identifie le commit, par exemple : 5ba786fcc93e8092831c01e71444b9baa2228a4f

**Note :** Pour simplifier, il suffit d'utiliser les 7 premiers caractères du hash (ex. : 5ba786f).

### 3.2 LOG COMMAND

```
git log [OPTIONS]
```

**Note :** La commande démarre dans un **pager interactif** qui permet de naviguer avec les flèches du clavier et de quitter avec q..

### 3.3 OPTIONS (FLAGS) DE LA COMMANDE GIT LOG

#### 3.3.1 LIMITER L'AFFICHAGE À UN CERTAIN NOMBRE DE COMMITS

```
-n <NOMBRE>
```

#### 3.3.2 AFFICHAGE SIMPLIFIÉ EN UNE LIGNE PAR COMMIT

```
--oneline
```

### 3.3.3 AFFICHER L'HISTORIQUE DE TOUTES LES BRANCHES (POUR VOIR LES MERGES)

```
--all
```

### 3.3.4 AFFICHAGE GRAPHIQUE DES COMMITS

```
--graph
```

### 3.3.5 AFFICHER LES COMMITS PARENTS

```
--parents
```

### 3.3.6 AFFICHER LES COMMITS D'UNE BRANCHE DISTANTE SPÉCIFIQUE

```
<REMOTE>/<BRANCH>
```

### 3.3.7 AFFICHER LES INFORMATIONS DES FICHIERS MODIFIÉS DES COMMITS

```
--stat
```

### 3.3.8 AFFICHER LES DIFFÉRENCES ENTRE TOUS LES COMMITS

```
-p
```

### 3.3.9 AFFICHER TOUTES LES RÉFÉRENCES (POINTEURS VERS LES COMMITS)

```
--decorate=<FULL | SHORT | NO>
```

- short (par défaut) : affiche les noms abrégés des références.
- full : affiche le nom complet des références.
- no : n'affiche pas les références.

### 3.3.10 LIMITER LES LOGS À CERTAINES DATES

```
git log --since="date" --until="date"
```

- Dates relatives : "1.week.ago", "yesterday", "2.days.ago"
- Dates absolues : "2024-11-23", "2024-11-23T15:30:00"
- Expressions naturelles : "last Friday", "1 month ago"

### 3.3.11 AFFICHER LES COMMITS D'UN AUTEUR EN PARTICULIER

```
git log --author="<nom_ou_email>"
```

## 4 FONCTIONNEMENT INTERNE

### 4.1 BASES DU FONCTIONNEMENT INTERNE (PLUMBING)

L'intégralité des données d'un dépôt Git (commits, branches, tags, objets, etc.) est stockée dans le dossier caché .git. Les objets Git sont enregistrés dans le répertoire **.git/objects/** où chaque commit n'est qu'un type d'objet parmi d'autres.

### 4.2 EXPLORATION DES OBJETS

#### 4.2.1 TROUVER LE HASH D'UN COMMIT

```
git log
```

```
lucam@lucam-VirtualBox:~/Documents/webflyx/.git/objects/53$ git log -n 10
commit 5324abd8a25cb558817dc21ed4f4fb962305e345 (HEAD -> master)
Author: MAN-Luca <luca.mansutti7@gmail.com> ↗
Date:   Sat Nov 16 17:15:55 2024 +0100
```

**Note :** Le but est de trouver les 2 premiers digits du hash et de retrouver l'objet correspondant

#### 4.2.2 LISTER LE CONTENU DU DOSSIER .GIT/OBJECTS

```
ls -al .git/objects
```

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ ls -al .git/objects/
total 28
drwxrwxr-x 7 lucam lucam 4096 Nov 16 17:15 .
drwxrwxr-x 8 lucam lucam 4096 Nov 16 17:15 ..
drwxrwxr-x 2 lucam lucam 4096 Nov 16 17:15 53 ↗
drwxrwxr-x 2 lucam lucam 4096 Nov 16 17:15 5b
drwxrwxr-x 2 lucam lucam 4096 Nov 16 17:11 ef
drwxrwxr-x 2 lucam lucam 4096 Nov 16 17:02 info
drwxrwxr-x 2 lucam lucam 4096 Nov 16 17:02 pack
```

### 4.3 COMMANDE CAT-FILE

#### 4.3.1 AFFICHER LE CONTENU D'UN COMMIT

**cat-file** est une commande de git qui permet de lire le contenu d'un commit

```
git cat-file -p <COMMIT_HASH>
```

**Note :** Mettre le hash complet ou les 7 premiers digits. Le -p (pretty print) affiche le contenu d'une manière lisible

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git cat-file -p 5324abd
tree 5b21d4f16a4b07a6cde5a3242187f6a5a68b060f
author MAN-Luca <luca.mansutti7@gmail.com> 1731773755 +0100
committer MAN-Luca <luca.mansutti7@gmail.com> 1731773755 +0100
```

```
A: add contents.md
```

## 4.4 TREES & BLOBS

Dans Git, les **arbres** (trees) et les **blobs** sont des objets qui servent à structurer et stocker les données du dépôt.

- **Tree** : Représente un **répertoire** (ou un dossier). Il contient des références vers des blobs (fichiers) ou d'autres arbres (sous-répertoires).
- **Blob** : Représente le contenu d'un **fichier**. C'est un objet de données pur sans structure hiérarchique.

### 4.4.1 EXPLORATION D'UN OBJET COMMIT

```
git cat-file -p <COMMIT_HASH>
```

**Résultat** : Cela t'affiche les informations sur le commit (auteur, date, message du commit) ainsi que l'objet **tree** associé, qui représente la structure des fichiers et répertoires du commit.

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git cat-file -p 5324abd
tree 5b21d4f16a4b07a6cde5a3242187f6a5a68b060f
author MAN-Luca <luca.mansutti7@gmail.com> 1731773755 +0100
committer MAN-Luca <luca.mansutti7@gmail.com> 1731773755 +0100

A: add contents.md
```

### 4.4.2 EXAMINER UN OBJET TREE

```
git cat-file -p <TREE_HASH>
```

**Résultat** : Cela te montrera une liste de fichiers et sous-répertoires associés à ce commit. Chaque fichier est représenté par un objet **blob** avec un hash unique.

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git cat-file -p 5b21d4f
100644 blob ef7e93fc61a91deecaa551c4707e4c3049af42c9 contents.md
```

### 4.4.3 EXAMINER UN OBJET BLOB

```
git cat-file -p <blob_HASH>
```

**Résultat** : Cela affiche le **contenu du fichier** sous forme de texte (si le fichier est lisible).

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git cat-file -p ef7e93
# contents
```

## 4.5 COMMITS MULTIPLES (PARENTS)

Avec la commande `git log` nous pouvons voir le nouveau commit effectué

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git --no-pager log -n 10
commit 3dba88a6a89a1bba0f8d0eeae60de590f8e57333 (HEAD -> master)
Author: MAN-Luca <luca.mansutti7@gmail.com>
Date:   Sat Nov 16 19:26:13 2024 +0100
        ↙ 2eme
    B: add titles

commit 5324abd8a25cb558817dc21ed4f4fb962305e345
Author: MAN-Luca <luca.mansutti7@gmail.com>
Date:   Sat Nov 16 17:15:55 2024 +0100
        ↙ 1er
    A: add contents.md
```

Avec la commande `git cat-file -p <COMMIT_HASH>` nous voyons désormais le paramètre **parent** (le commit précédent dans l'historique du dépôt)

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git cat-file -p 3dba88a
tree 04819d36f56c9198bd060844298495418bc35743
parent 5324abd8a25cb558817dc21ed4f4fb962305e345
author MAN-Luca <luca.mansutti7@gmail.com> 1731781573 +0100
committer MAN-Luca <luca.mansutti7@gmail.com> 1731781573 +0100
B: add titles
```

## 4.6 STOCKAGE DES DONNÉES

### 4.6.1 COMMENT GIT STOCKE LES DONNÉES

Git stocke une **photo instantanée** (snapshot) de tous les fichiers à chaque commit, ce qui signifie que chaque commit contient une version complète des fichiers. Contrairement à ce que l'on pourrait penser, Git ne stocke pas seulement les changements effectués dans un commit, mais plutôt l'état complet des fichiers.

### 4.6.2 OPTIMISATION DU STOCKAGE

Bien que Git stocke des snapshots complets, il optimise le stockage afin de ne pas rendre le répertoire .git trop volumineux :

- Git **compresse** et **regroupe** les fichiers pour les stocker de manière plus efficace.
- Git **déduplique** les fichiers identiques entre les commits : si un fichier n'a pas changé entre deux commits, Git ne le stocke qu'une seule fois.

Les fichiers qui n'ont pas changé entre 2 commits possèdent le même hash\_blob

## **5 BRANCHING**

### **5.1 UTILITÉ DES BRANCHES**

En Git, une **branche** est un mécanisme qui permet de suivre des modifications séparément, sans affecter directement le projet principal.

Deux scénarios possibles :

- **Les modifications sont satisfaisantes** : Vous fusionnez (**merge**) la branche X avec la branche principale (master ou main).
- **Les modifications ne conviennent pas** : Vous supprimez la branche X sans que la branche principale soit affectée.

### **5.2 FONCTIONNEMENT INTERNE**

**Une branche = un pointeur nommé vers un commit spécifique.**

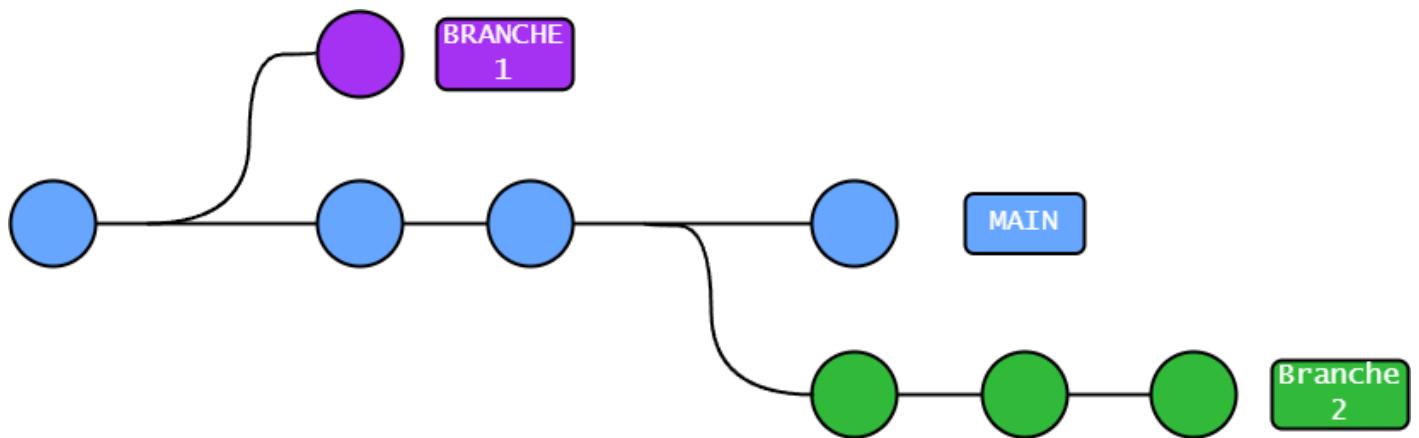
Lorsque vous créez une branche, Git crée simplement un "pointeur" vers un commit existant.

**La branche suit les nouveaux commits.**

À chaque nouveau commit, le pointeur de la branche se déplace automatiquement pour désigner le commit le plus récent (appelé "**tip** de la branche").

**Branches légères et efficaces :**

- Les branches ne dupliquent pas le projet sur le disque dur.
- Créer plusieurs branches est rapide et peu gourmand en ressources.



## 5.3 OPÉRATIONS SUR LES BRANCHES

### 5.3.1 AFFICHER TOUTES LES BRANCHES DU REPOSITORY

```
git branch [--list]
```

**Note** : la branche avec le \* est la branche dans laquelle on se trouve

### 5.3.2 CRÉER UNE BRANCHE SANS Y BASCULER DIRECTEMENT

```
git branch <BRANCH> [<COMMIT_HASH>]
```

**Note** : le flag COMMIT\_HASH permet de choisir à partir de quel commit on veut créer la branche (à utiliser si on ne veut pas créer la branche à partir du dernier commit)

### 5.3.3 CRÉER ET BASCULER DIRECTEMENT SUR LA NOUVELLE BRANCHE

```
git switch -c <BRANCH> [<COMMIT_HASH>]
```

### 5.3.4 BASCULER ENTRE LES BRANCHES

```
git switch <BRANCH>
```

### 5.3.5 RENOMMER UNE BRANCHE

```
git branch -m <OLD_NAME> <NEW_NAME>
```

**Note** : GitHub utilise main pour la branche par défaut

Ex : `git branch -m master main` cela change la branche "master" en "main"

### 5.3.6 SUPPRIMER UNE BRANCHE

```
git branch -d <BRANCH>
```

## 5.4 AJOUT D'UN COMMIT DANS UNE NOUVELLE BRANCHE

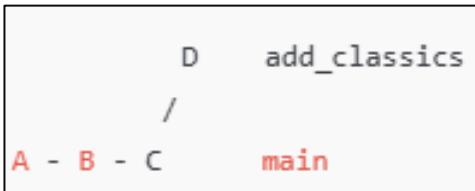
**Exemple** : Ajout de la branche "add\_classics", contenant un nouveau fichier "classics.csv"

```
git switch -c add_classics
touch classics.csv
git add .
git commit -m "Ajout de classics.csv"
```

Vérifions le résultat avec `git log` :

Résultat attendu :

- La branche `add_classics` contient un commit supplémentaire (D).
- La branche `main` reste intacte, pointant toujours au commit C.



```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git log  
commit 7fc1887d31b224acedc8a9d2f8d04e43607ca048 (HEAD -> add_classics)  
Author: MAN-Luca <luca.mansutti7@gmail.com>  
Date:  Sun Nov 17 14:33:17 2024 +0100  
  
    D: add classics  
  
commit 5526a0969904e002e71e0db9a58b14ebe71e0e9f (main)  
Author: MAN-Luca <luca.mansutti7@gmail.com>  
Date:  Sat Nov 16 19:46:14 2024 +0100  
  
    C: add quotes  
  
commit 3dba88a6a89a1bb0f8d0eeae60de590f8e57333  
Author: MAN-Luca <luca.mansutti7@gmail.com>  
Date:  Sat Nov 16 19:26:13 2024 +0100  
  
    B: add titles  
  
commit 5324abd8a25cb558817dc21ed4f4fb962305e345  
Author: MAN-Luca <luca.mansutti7@gmail.com>  
Date:  Sat Nov 16 17:15:55 2024 +0100  
  
    A: add contents.md
```

## 5.5 LE FICHIER GIT ET LES BRANCHES

Git stocke toutes les informations liées à votre projet dans le répertoire caché `.git` à la racine de votre projet. Ce répertoire contient des sous-dossiers et fichiers organisés pour suivre les commits, branches, objets et configurations.

### 5.5.1 BRANCHES ET LEURS RÉFÉRENCES

Les branches sont représentées comme des fichiers dans le répertoire `.git/refs/heads`. Chaque fichier porte le nom d'une branche et contient le hash du commit vers lequel pointe cette branche.

**Exemple :** Le fichier `.git/refs/heads/main` contient le hash du dernier commit

### 5.5.2 COMMANDES POUR EXAMINER LES RÉFÉRENCES

#### 5.5.2.1 AFFICHER LE HASH DU COMMIT D'UNE BRANCHE

```
cat .git/refs/heads/<BRANCH>
```

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ cat .git/refs/heads/main  
5526a0969904e002e71e0db9a58b14ebe71e0e9f
```

#### 5.5.2.2 AFFICHER TOUTES LES BRANCHES

```
find .git/refs/heads -type f
```

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ find .git/refs/heads/ -type f  
.git/refs/heads/add_classics  
.git/refs/heads/main
```

#### 5.5.2.3 AFFICHER TOUTES LES BRANCHES ET LEUR COMMITS

```
find .git/refs/heads -type f | while read branch; do echo "$branch: $(cat $branch)"; done
```

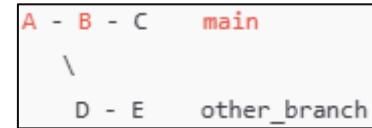
```
lucam@lucam-VirtualBox:~/Documents/webflyx$ find .git/refs/heads/ -type f | while  
read branch; do echo "$branch: $(cat $branch)"; done  
.git/refs/heads/add_classics: 7fc1887d31b224acedc8a9d2f8d04e43607ca048  
.git/refs/heads/main: 5526a0969904e002e71e0db9a58b14ebe71e0e9f
```

# 6 MERGE

## 6.1 POURQUOI AVOIR PLUSIEURS BRANCHES ?

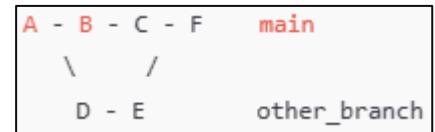
Les branches multiples permettent de faire des modifications sans affecter la branche principale. Une fois les modifications terminées, vous pouvez les fusionner avec la branche principale pour les intégrer au produit final.

**Exemple :** Supposons que vous ayez deux branches avec des commits uniques :



Si vous fusionnez **other\_branch** dans **main**, Git combine les deux branches en créant un nouveau commit de fusion (F) qui a à la fois C et E comme parents :

Dans cet exemple, F ramène les modifications de D et E dans la branche principale.



## 6.2 MERGE COMMAND

```
git merge <OTHER_BRANCH> -m "MESSAGE"
```

**Note :** Avant d'effectuer cette commande se redéplacer dans la branche main (ou autre)

## 6.3 MERGE COMMITS

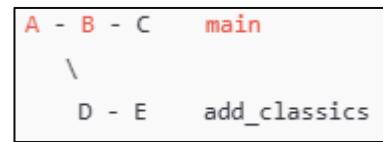
### 6.3.1 FONCTIONNEMENT DES MERGE COMMITS

Un **merge commit** est un commit spécial créé lorsqu'on fusionne deux branches.

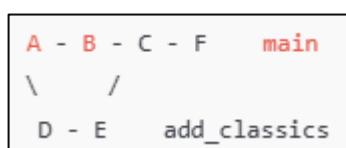
**Exemple :** Imaginons que nous avons deux branches, **main** et **add\_classics**, avec leurs commits respectifs

Fusionnons la branche **add\_classics** dans **main**

→ Git va procéder de la manière suivante :



- **Trouver le commit de base de fusion** : C'est le commit A, l'ancêtre commun des deux branches.
- **Rejouer les changements de main** (depuis le commit de base, donc depuis A).
- **Rejouer les changements de add\_classics** (depuis le commit de base, donc depuis A).
- **Créer un commit de fusion** (F), qui aura **deux parents** : C (de main) et E (de add\_classics).



## 6.4 MERGE LOG

### 6.4.1 COMMANDE DE VÉRIFICATION DU MERGE

```
git log --oneline --graph --parents [--all]
```

Note : Ajout du flag **--parents** pour voir les hash des commits parents (ou bien **--all**)

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git log --oneline --graph --parents
*   a6bf079 b864b2f 7fc1887 (HEAD -> main) F: Merge branch 'add_classics'
|\ \
| * 7fc1887 5526a09 (add_classics) D: add classics
* | b864b2f 5526a09 E: update contents.md
|/
* 5526a09 3dba88a C: add quotes
* 3dba88a 5324abd B: add titles
* 5324abd A: add contents.md
```

- **Les barre verticale (|)** montre la progression linéaire des commits dans une branche.
- **Les barres obliques (/)** indiquent la fusion de deux branches, ce qui est représenté par le commit de fusion (89629a9).
- **Les astérisques (\*)** indiquent des commits, avec les hash qui identifient chaque commit.

### 6.4.2 EXPLICATIONS DES HASH AVEC DES PARENTS

#### 6.4.2.1 COMMIT DE FUSION (*MERGE COMMIT*)

- **a6bf079 b864b2f 7fc1887** : Cette ligne représente un commit de fusion, identifié par l'ID de commit **a6bf079**. Les deux autres hash (**b864b2f** et **7fc1887**) sont les commits parents de cette fusion. Cela signifie que le commit de fusion a deux parents, l'un venant de la branche principale (main), et l'autre de la branche fusionnée (add\_classics).
- **(HEAD -> main)** indique que vous êtes actuellement sur la branche main et que ce commit est le plus récent sur cette branche.

#### 6.4.2.2 COMMITS DE LA BRANCHE ADD\_CLASSICS

- **7fc1887 5526a09** : Ce commit appartient à la branche add\_classics, et **7fc1887** est l'identifiant de ce commit. Il indique l'ajout de fonctionnalités liées à "add classics". **5526a09** est le parent.

#### 6.4.2.3 COMMITS DE LA BRANCHE MAIN :

- **b864b2f 5526a09** : Ce commit appartient à la branche main, et **b864b2f** est l'identifiant de ce commit. Il indique l'ajout de fonctionnalités liées à "main ". **5526a09** est le parent.

## 6.5 FAST FORWARD MERGE / COMMIT

Un **merge fast-forward** est la méthode la plus simple de fusion entre deux branches Git. Cela se produit lorsque la **branche de destination contient tous les commits de la branche source**. Git peut alors avancer le pointeur de la branche de destination directement à la pointe de la branche source, **sans créer de commit de fusion**.

Exemple : Etat initial :



```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git merge update_titles
Updating a6bf079..82d7f8f
Fast-forward
  titles.md | 1 +
  1 file changed, 1 insertion(+)
```

Apres merge :



```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git log --oneline
82d7f8f (HEAD -> main, update_titles) G: update titles.md
a6bf079 F: Merge branch 'add_classics'
b864b2f E: update contents.md
7fc1887 D: add classics
5526a09 C: add quotes
3dba88a B: add titles
5324abd A: add contents.md
```

**Note :** Le merge fast-forward est souvent utilisé pour fusionner des branches de courte durée ou des modifications mineures, car il simplifie l'historique Git en évitant les commits de fusion inutiles.

## 7 REBASE

### 7.1 VISUALISATION D'UN REBASE

Le rebase ne crée pas de merge commit

La branche qui est rebase ne change pas ! Les commits sont ramenés à l'autre branche

Situation initiale :

Nous avons deux branches avec l'historique suivant :

- **main** : contient les commits partagés par toute l'équipe.
- **feature\_branch** : contient des modifications spécifiques en cours de développement



**Objectif** : Mettre à jour **feature\_branch** avec les derniers changements de **main** tout en évitant la création d'un commit de merge.

Après un Rebase :

Le rebase **rejoue** les commits D et E de **feature\_branch** sur la base de la branche **main**. Cela donne un historique propre et linéaire, comme si D et E avaient été créés directement après C.



### 7.2 EXÉCUTER UN REBASE

#### 7.2.1 SE POSITIONNER DANS LA BRANCHE DANS LAQUELLE ON VEUT RAMENER LES COMMITS

`git switch <BRANCH>`

#### 7.2.2 COMMANDE REBASE

`git rebase <main | OTHER_BRANCH>`

Ici nous voyons qu'il manque les commits de main E, F et G dans la branche **update\_dune**

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git log --oneline
f53c429 (HEAD -> update_dune) I: update dune again
d272c43 H: update dune
7fc1887 D: add classics
5526a09 C: add quotes
3dba88a B: add titles
5324abd A: add contents.md
```

Après le rebase, les commits qui n'étaient pas présent sont ramenés dans la branche **update\_dune**

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git rebase main
Successfully rebased and updated refs/heads/update_dune.
lucam@lucam-VirtualBox:~/Documents/webflyx$ git log --oneline
702becc (HEAD -> update_dune) I: update dune again
868244f H: update dune
82d7f8f (main) G: update titles.md
a6bf079 F: Merge branch 'add_classics'
b864b2f E: update contents.md
7fc1887 D: add classics
5526a09 C: add quotes
3dba88a B: add titles
5324abd A: add contents.md
```

## 7.3 MERGE VS REBASE

### 7.3.1 MERGE

- Conserve **l'historique complet** du projet, montrant quand et où les branches ont été fusionnées.
- Utile pour garder une trace précise du travail collaboratif.
- **Inconvénient** : peut encombrer l'historique avec de nombreux commits de merge, le rendant plus difficile à suivre.

### 7.3.2 REBASE :

- Crée un **historique linéaire** en rejouant les commits d'une branche sur une autre.
- Plus facile à lire et à comprendre, surtout pour un travail individuel ou isolé.
- Idéal pour garder les branches de fonctionnalités (features) à jour sans ajouter de commits de merge inutiles.

## 7.4 QUAND UTILISER REBASE ?

### Branches publiques (par ex. main) :

- Ne jamais faire un rebase sur une branche publique. Réécrire l'historique d'une branche partagée peut causer des problèmes sérieux, comme casser l'historique pour les autres développeurs.
- Préférez **merge** pour intégrer les modifications sur les branches publiques.

## 7.5 QUAND ÉVITER REBASE ?

### • Branches privées (vos branches de fonctionnalités) :

- Faites régulièrement un rebase pour intégrer les changements de **main** ou d'autres branches sans polluer l'historique.
- Permet de garder votre branche à jour tout en maintenant un historique propre.

### • Nettoyage avant de partager :

- Utilisez rebase pour nettoyer ou combiner des commits en unités significatives avant de les partager avec votre équipe.

## 8 ANNULATION ET RESTAURATION

### 8.1 RESET COMMAND

#### 8.1.1 SOFT RESET

Cette option de reset revient à un commit précédent tout en gardant toutes les modifications intactes dans l'index (staged).

```
git reset --soft <COMMIT_HASH>
```

Effets :

- Les changements des commits annulés restent **staged**.
- Les changements non committés restent dans leur état initial (staged ou unstaged).



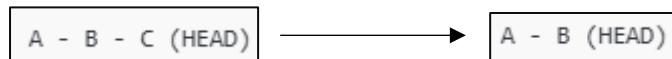
#### 8.1.2 HARD RESET

Cette option de reset permet de revenir à un commit précédent tout en supprimant toutes les modifications effectuées après ce commit. Contrairement à **--soft**, cette option **efface complètement** les changements dans l'index (staged) et dans le répertoire de travail (worktree).

```
git reset --hard <COMMIT_HASH>
```

Effets :

- **Le HEAD** est déplacé vers le commit spécifié.
- **L'index et le worktree** sont mis à jour pour correspondre exactement à l'état du commit.
- **⚠️ Toutes les modifications non committées ou staged sont perdues.**



#### 8.1.3 DANGERS DU HARD RESET

**⚠️ Attention :** La commande git reset --hard est extrêmement puissante, mais elle peut entraîner des pertes irrémédiables de données. Contrairement à la suppression simple d'un fichier suivi par Git (qui peut être facilement restauré car il est enregistré dans l'historique), **un fichier supprimé via git reset --hard ne pourra pas être récupéré directement**.

**Assurez-vous d'avoir sauvegardé vos changements ou utilisé git stash.**

## 8.2 RESTORE COMMAND

### 8.2.1 RESTAURER DES FICHIERS DU DERNIER COMMIT DANS LE RÉPERTOIRE LOCAL

```
git restore <FILE | .>
```

**Note :** Cela permet de revenir à l'état du fichier tel qu'il était dans le dernier commit, annulant ainsi les modifications locales.

### 8.2.2 ANNULER UN AJOUT AU STAGE (UNSTAGE) :

```
git restore --staged <FILE>
```

**Note :** Cette commande permet de retirer un fichier de l'index (staged), le rendant à nouveau modifié mais non encore prêt pour le commit.

## 8.3 REVERT COMMAND

Revert crée un **nouveau commit** qui **annule exactement les modifications apportées par un commit précédent**. Contrairement à reset, il **ne supprime aucun commit**, ce qui le rend idéal pour des situations où l'historique doit être conservé intact

```
git revert <COMMIT_HASH>
```

**Note :** Pour annuler un revert, il faut "revert le revert" en utilisant le nouveau commit hash associé au revert

## 8.4 RÉCAPITULATIF

Commande	Historique modifié ?	Modifications locales conservées ?	Utilisation typique
reset	Oui	Non (avec --hard) Oui (--soft)	Revenir à un état précédent sans garder l'historique.
restore	Non	Oui	Annuler des changements locaux ou unstager un fichier.
revert	non	Oui	Annuler un commit tout en conservant l'historique.

## **9 REMOTE**

### **9.1 GÉRER DES DÉPÔTS DISTANTS**

Les dépôts distants permettent de collaborer avec d'autres développeurs en partageant le code via un référentiel externe. Ces dépôts, appelés **remotes**, conservent généralement un historique Git similaire à votre dépôt local.

Par convention le dépôt distant est appelé **origin**

### **9.2 AJOUTER UN REMOTE**

**git remote add <REMOTE> <REMOTE\_URL>**

**Note :** L'URL peut aussi être le chemin relatif du dépôt local dans la machine

**Exemple :** `git remote add origin main` ajoute main comme dépôt distant "origin"

### **9.3 FETCH**

#### **9.3.1 FETCH COMMAND**

La commande `git fetch` permet de récupérer les modifications d'un dépôt distant (dossier `.git/`) sans les intégrer automatiquement dans vos branches locales.

**git fetch [<REMOTE>]**

**Exemple :** `git fetch origin`

#### **9.3.2 FETCH N'INTÈGRE PAS LES COMMITS DU DÉPÔT DISTANT**

Même après un `fetch`, les commits ne sont pas encore accessibles dans votre historique local tant que vous n'avez pas explicitement lié votre branche locale à une branche distante ou fusionné les modifications.

#### **9.3.3 CONCLUSION**

Tant que vous n'avez pas explicitement basculé sur une branche distante ou fusionné les commits récupérés, votre historique local restera vide. Cette étape démontre que **git fetch ne modifie pas automatiquement vos branches locales**, ce qui vous donne un contrôle total sur le processus d'intégration

## 9.4 LOG REMOTE

### 9.4.1 AFFICHER L'HISTORIQUE D'UNE BRANCHE DISTANTE AVEC GIT LOG

La commande git log peut également être utilisée pour examiner les commits d'une branche distante sans les avoir intégrés à votre branche locale. Cela permet d'avoir un aperçu des changements dans le dépôt distant.

```
git log remote/branch
```

## 9.5 MERGE REMOTE

### 9.5.1 FUSIONNER UNE BRANCHE DISTANTE DANS UNE BRANCHE LOCALE

Lorsque vous fusionnez une branche distante (origin/main) dans une branche locale (main), cela permet de **synchroniser le contenu du dépôt distant dans votre dossier local**. Cela inclut les nouveaux commits et fichiers.

### 9.5.2 MERGE REMOTE COMMAND

```
git merge remote/branch
```

**Note :** On fait cette commande dans le **dépôt local** au sein de la **branche main**

# **10 GITHUB**

## **10.1 GITHUB REPOSITORY**

### **10.1.1 GITHUB**

**GitHub** est la plateforme la plus populaire pour héberger des dépôts Git en ligne. Elle offre plusieurs avantages :

- **Sauvegarde de ton code** : Stocke tes projets dans le cloud, assurant une copie de sauvegarde en cas de problème avec ta machine locale.
- **Collaboration** : Partage ton code et collabore avec d'autres développeurs.
- **Portfolio public** : Utilise GitHub comme vitrine pour tes projets et démontre ton travail.

### **Git ≠ GitHub**

Il est essentiel de comprendre que **Git** et **GitHub** ne sont pas la même chose :

- **Git** : Un outil open-source de gestion de versions que tu utilises localement sur ton ordinateur pour suivre les changements apportés aux fichiers de code.
- **GitHub** : Un service web commercial qui héberge des dépôts gérés par Git. Il existe également d'autres plateformes similaires comme **GitLab** et **Bitbucket**.

### **10.1.2 GITHUB CLI**

**GitHub CLI** (Command Line Interface) est un outil qui permet d'interagir avec GitHub directement depuis le terminal. Il facilite les actions courantes comme la gestion des dépôts, l'authentification, l'ouverture de pull requests, la création de nouveaux issues ou la consultation des activités, sans avoir à passer par l'interface web. Cela permet aux développeurs de gagner du temps et d'automatiser certaines tâches liées à GitHub.

## 10.2 GESTION DU DÉPÔT SUR GITHUB

### 10.2.1 CRÉER UN DÉPÔT SUR GITHUB

The screenshot shows the GitHub web interface for creating a new repository. On the left, a sidebar lists options: New repository (selected), Import repository, New codespace, New gist, and New organization. The main area is titled 'Create a new repository' with a sub-instruction: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.' It asks for 'Owner \*' (set to MAN-Luca) and 'Repository name \*'. Below, it says 'Great repository names are short and memorable. Need inspiration? How about musical-garbanzo ?'. A 'Description (optional)' field is present. At the bottom, there are two radio button options: 'Public' (selected) with the note 'Anyone on the internet can see this repository. You choose who can commit.', and 'Private' with the note 'You choose who can see and commit to this repository.'

**Set up GitHub Copilot**  
Use GitHub's AI pair programmer to autocomplete suggestions as you code.  
[Get started with GitHub Copilot](#)

**Add collaborators to this repository**  
Search for people using their GitHub username or email address.  
[Invite collaborators](#)

**Quick setup — if you've done this kind of thing before**  
Set up in Desktop or HTTPS SSH <https://github.com/MAN-Luca/webflyx.git>  
Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

**...or create a new repository on the command line**  
echo "# webflyx" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin https://github.com/MAN-Luca/webflyx.git  
git push -u origin main

**...or push an existing repository from the command line**  
git remote add origin https://github.com/MAN-Luca/webflyx.git  
git branch -M main  
git push -u origin main

### 10.2.2 S'AUTHENTIFIER AVEC GITHUB CLI (GH)

```
gh auth login / logout
```

```
lucam@lucam-VirtualBox:~$ gh auth login
? What account do you want to log into? GitHub.com
? What is your preferred protocol for Git operations? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI? Login with a web browser

! First copy your one-time code: E5D3-D184
- Press Enter to open github.com in your browser... █
```

## 10.3 REMOTE ADD COMMAND

### 10.3.1 LIER LE DÉPÔT LOCAL AU DÉPOT DISTANT (REMOTE) SUR GITHUB

```
git remote add origin <REPOSITORY_URL>
```

### 10.3.2 VÉRIFIER LA LIAISON

```
git ls-remote
```

```
lucam@lucam-VirtualBox:~/Documents/webflyx$ git ls-remote  
From https://github.com/MAN-Luca/webflyx.git
```

## 10.4 GIT PUSH

La commande git push envoie les modifications locales vers un "dépôt distant", comme GitHub. Par exemple, pour pousser les commits de notre branche main locale vers la branche main du dépôt distant, on utiliserait :

```
git push [-u] [--force] origin main
```

**Note : Pour le premier push il faut mettre -u pour lier les dépôts locaux et distants.**

L'option **--force** permet de forcer git à push sur le repository en cas d'erreurs

Il est important d'être authentifié avec le dépôt distant pour pouvoir pousser des modifications

### 10.4.1 OPTIONS ALTERNATIVES

Vous pouvez également pousser une branche locale vers une autre branche que main

```
git push origin <BRANCH>
```

De plus, vous pouvez supprimer une branche distante en poussant une branche vide :

```
git push origin :<REMOTEBRANCH>
```

## 10.5 GIT PULL

Le git pull permet de récupérer les modifications réelles des fichiers depuis un dépôt distant, pas seulement les métadonnées comme avec git fetch

```
git pull [<REMOTE>/<BRANCH>]
```

**Note :** Si vous exécutez git pull sans rien spécifier, il récupérera la branche actuelle depuis le dépôt distant par défaut

## 10.6 PULL REQUEST

### 10.6.1 FONCTIONNEMENT D'UN PULL REQUEST

Le Push effectué pour faire un Pull request doit se faire sur une autre branche que main

Une Pull Request (PR) sur GitHub est un moyen de proposer des changements dans un projet

#### 1. Collaboration et révision :

- Elle permet à d'autres membres de l'équipe ou aux responsables du projet de voir les changements proposés avant qu'ils ne soient intégrés au projet principal..

#### 2. Flux de travail avec GitHub :

- Une fois qu'une PR est ouverte, elle peut être commentée, validée, et même modifiée avant d'être fusionnée dans la branche cible.

#### 3. Fusion (Merge) :

- Après les discussions et les ajustements nécessaires, une PR peut être fusionnée dans la branche cible (souvent `main`), ce qui intègre les changements dans le code de base.

### 10.6.2 EXEMPLE DE PULL REQUEST

#### 10.6.2.1 PARTIE "ENVOYER LES MODIFICATIONS"

- Création d'une nouvelle branche à partir de `main` :

```
git switch -c <NEW_BRANCH>
```

- Modifications des fichiers
- Commit les modifications

```
git add .  
git commit -m "MESSAGE"
```

- Push de la nouvelle branche sur GitHub :

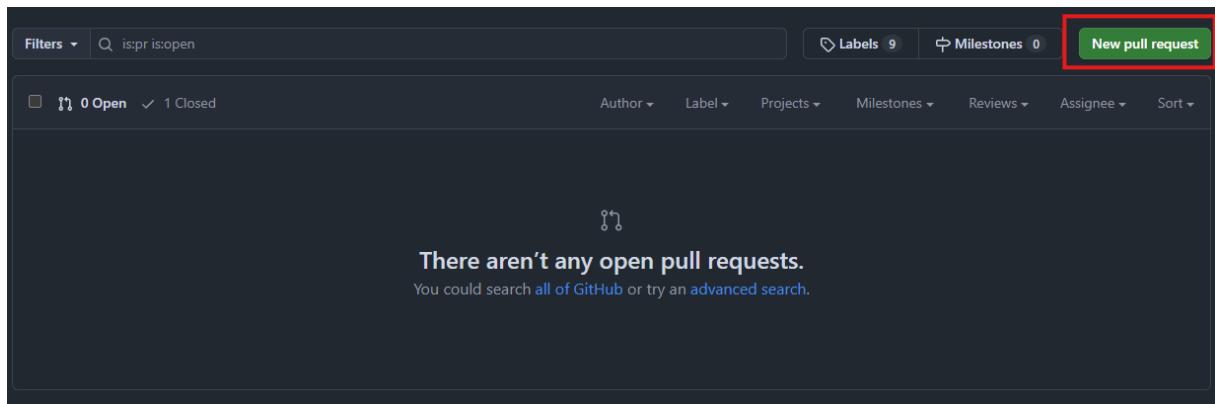
```
git push origin <NEW_BRANCH>
```

## 10.6.2.2 PARTIE "CRÉATION D'UNE PULL REQUEST"

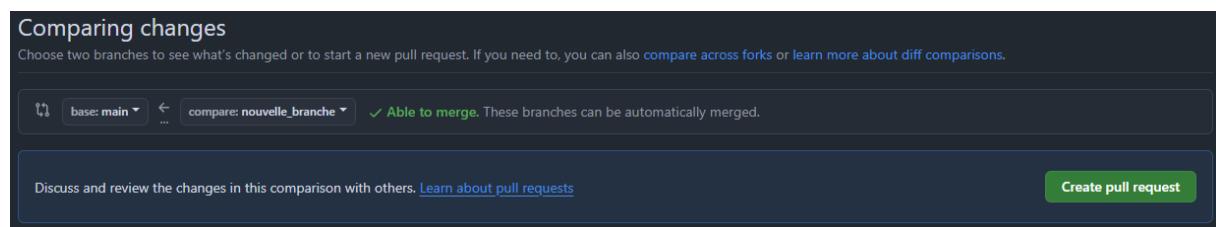
- Clique sur l'onglet "Pull requests" en haut de la page



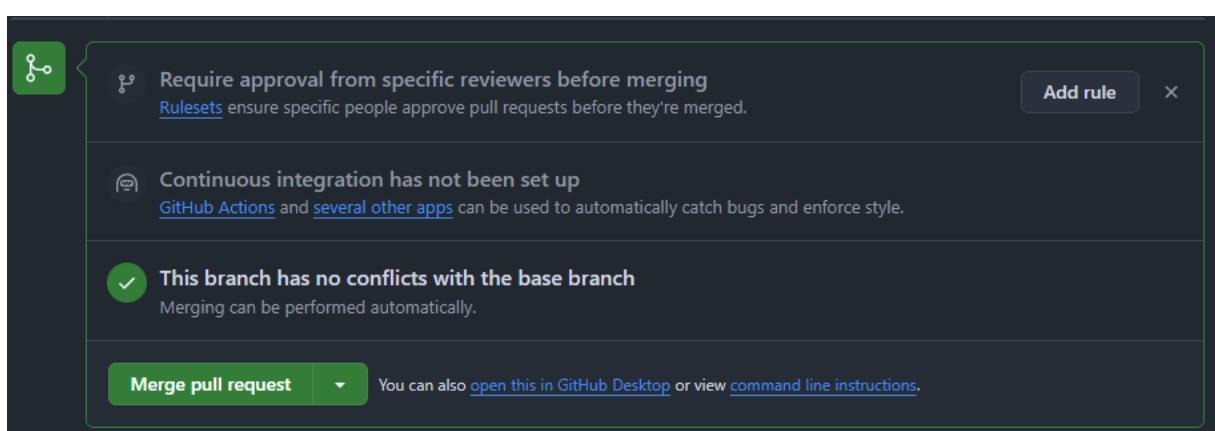
- Clique sur "New pull request"



- Dans l'interface de création de la PR, choisis **main** comme branche de base et la **nouvelle\_branche** comme branche de comparaison. Puis crée la pull request



- Pour finir l'administrateur peut décider de merge ou non les modifications



# 11 GITIGNORE

## 11.1 FICHIER .GITIGNORE

Le fichier `.gitignore` permet de spécifier quels fichiers ou répertoires Git doit ignorer lors des opérations de suivi de fichiers dans un dépôt.

### 11.1.1 EXEMPLE D'UTILISATION

Exemple, si tu écris dans un fichier `.gitignore` :

```
node_modules
```

Git ignorera **tous** les répertoires ou fichiers appelés `node_modules` à n'importe quel niveau du répertoire. Cela inclut :

- `node_modules/code.js`
- `src/node_modules/code.js`
- `src/node_modules/`

Cependant, cela **ne va pas** ignorer des chemins comme :

- `src/node_modules_2/code.js`
- `env/node_modules_3`

Cela montre que Git va uniquement ignorer les fichiers ou dossiers exactement nommés `node_modules` et leur contenu, sans toucher aux autres chemins similaires.

## 11.2 NESTED .GITIGNORE

Dans un projet, il est possible d'avoir plusieurs fichiers `.gitignore` placés dans différents répertoires. Ces fichiers **fonctionnent de manière hiérarchique et ne s'appliquent qu'au répertoire où ils se trouvent et à ses sous-répertoires.**

### 11.2.1 FONCTIONNEMENT

- **Fichier `.gitignore` dans le répertoire racine (unique):**
  - Les règles définies dans ce fichier s'appliquent à l'ensemble du projet, sauf si elles sont contredites par un fichier `.gitignore` dans un sous-répertoire.
- **Fichiers `.gitignore` dans des sous-répertoires (peuvent être multiples) :**
  - Ces fichiers remplacent ou ajoutent des règles spécifiques pour le répertoire où ils se trouvent et ses sous-répertoires.

## 11.2.2 EXEMPLE CONCRET

Arborescence :

```
/project
  .gitignore
  /src
    .gitignore
    /components
      file.js
      temp.log
    build/
  /docs
    temp.doc
```

Contenu du .gitignore à la racine :

```
*.log
build/
```

Contenu du .gitignore dans /src :

```
!build/
*.tmp
```

Résultat :

1. **À la racine :**
  - o Les fichiers .log sont ignorés partout dans le projet.
  - o Le dossier /src/build est suivi car le .gitignore imbriqué utilise !build/ pour annuler l'exclusion.
2. **Dans /src :**
  - o Les fichiers .tmp dans /src et ses sous-dossiers sont ignorés.
  - o Les fichiers .log restent ignorés à cause de la règle racine.
3. **Dans /docs :**
  - o Le fichier temp.doc n'est pas affecté par le .gitignore dans /src.

## 11.3 PATTERNS

### 11.3.1 WILDCARD (\*)

**Le caractère \* remplace n'importe quel caractère sauf /**

Exemple : **\*.txt** Ignore tous les fichiers avec l'extension .txt

### 11.3.2 DOUBLE WILDCARD (\*\*)

**\*\*/foo** : Correspond à "foo" partout dans l'arborescence.  
**/foo/\*\*** : Correspond à tout contenu dans "foo" avec une profondeur infinie.

### 11.3.3 ROOTED PATTERNS (/)

Les patterns commençant par / s'appliquent uniquement au répertoire contenant le fichier .gitignore

**Exemple :** `/main.` Ignore uniquement le fichier main.py situé dans le répertoire racine, mais pas dans les sous-dossiers.

#### 11.3.4 NÉGATION

**Le caractère ! annule une règle précédente pour inclure des fichiers ou dossiers spécifiques**

**Exemple :**

```
*.txt  
!important.txt
```

Ignore tous les fichiers .txt sauf important.txt

#### 11.3.5 COMMENTAIRES

**Toute ligne commençant par # est un commentaire, utile pour documenter les règles**

**Exemple :**

```
# Ignorer tous les fichiers logs  
*.log
```

### 11.4 QUE METTRE DANS LE .GITIGNORE ?

#### 11.4.1 FICHIERS GÉNÉRÉS

Il s'agit des fichiers qui sont créés automatiquement par des processus de compilation ou de construction. Ils peuvent être régénérés à partir du code source ou de la configuration, il n'est donc pas nécessaire de les suivre avec Git.

- **Code compilé** : Fichiers comme .class, .o, .exe, .dll, .pyc, .pyo, etc.
- **Fichiers minifiés** : Fichiers générés pour la production comme .min.js, .min.css.
- **Fichiers de logs et temporaires** : Fichiers comme .log, .bak, .swp ou ceux créés lors des tests.

#### 11.4.2 DÉPENDANCES

Les dépendances sont gérées par des gestionnaires de paquets, il est donc préférable de ne pas les suivre. Si vous les suivez, votre dépôt va grossir inutilement et les changements de dépendances devraient être gérés par le gestionnaire de paquets.

- **Node modules** (`node_modules`) pour les projets JavaScript.
- **Environnements virtuels** (`venv`, `env`) pour les projets Python.
- **Répertoires de paquets** comme `vendor/` pour PHP, `libs/` pour Java, ou `.bundle/` pour Ruby.
- **Fichiers spécifiques au système** pour différents environnements comme `*.dmg`, `.app`, ou `.vs` pour Visual Studio.

#### 11.4.3 FICHIERS PERSONNELS OU SPÉCIFIQUES À L'ÉDITEUR

De nombreux développeurs utilisent des configurations personnalisées pour leurs éditeurs, IDE ou autres outils. Ces fichiers sont spécifiques à l'environnement de travail local et ne sont pas nécessaires pour le projet ou l'équipe.

- **Paramètres d'éditeur :** Fichiers comme .vscode/, .idea/, .editorconfig, .sublime-project.
- **Fichiers spécifiques au système d'exploitation :** Fichiers comme .DS\_Store (macOS), Thumbs.db (Windows), ou desktop.ini.

#### 11.4.4 INFORMATIONS SENSIBLES OU DANGEREUSES

Il est crucial d'éviter de suivre des fichiers contenant des informations sensibles, car ils pourraient exposer des mots de passe, des clés API et d'autres secrets qui doivent rester privés. Ces fichiers ne doivent jamais être dans un dépôt public ou partagé.

- **Fichiers d'environnement** comme .env, .env.local, qui peuvent contenir des clés API, des secrets ou des informations de connexion à une base de données.
- **Clés privées ou certificats** comme .pem, .key, .crt.
- **Fichiers de base de données** comme \*.sqlite, \*.db qui peuvent contenir des données personnelles, financières ou autrement sensibles.

## **12 README.MD**

### **12.1 QU'EST-CE QU'UN FICHIER README ?**

Un fichier **README.md** est essentiel dans un projet / repository pour expliquer son objectif, son utilisation et ses détails techniques. Il sert de **première impression** pour les collaborateurs, utilisateurs ou recruteurs.

Ce fichier doit être **placé à la racine du dépôt local** et doit s'appeler **README.md** (md signifie **markdown**)

GitHub reconnaît automatiquement ce fichier et l'affiche sur la page d'accueil du repository

### **12.2 SYNTAXE D'UN FICHIER .MD**

#### **12.2.1 TITRES**

Utilisez des signes dièse (#) pour créer des titres. Plus il y a de dièses, plus le titre est de niveau bas.

**Exemple :**

```
# Titre de niveau 1  
## Titre de niveau 2  
### Titre de niveau 3
```

#### **12.2.2 GRAS ET ITALIQUE**

- **Gras** : Entourez le texte avec deux astérisques \*\* ou deux underscores \_\_.
- **Italique** : Entourez le texte avec un astérisque \* ou un underscore \_.

```
**Texte en gras**  
*Texte en italique*
```

#### **12.2.3 LISTES**

- **Listes à puces** : Utilisez des astérisques \*, des tirets -, ou des plus +.
- **Listes numérotées** : Utilisez des numéros suivis d'un point.

```
- Élément 1  
- Élément 2  
  - Sous-élément 1  
  - Sous-élément 2  
* Élément 3  
  
1. Première étape  
2. Deuxième étape
```

## 12.2.4 LIENS ET IMAGES

Pour ajouter un lien, utilisez cette syntaxe. Les images sont ajoutées de manière similaire aux liens, mais avec un point d'exclamation au début :

```
| <!>[Texte](URL)
```

## 12.2.5 BLOC DE CODE AVEC COLORATION SYNTAXIQUE

- **Code en ligne** : Entourez le texte de backticks (`).
- **Blocs de code** : Utilisez trois backticks (```) avant et après le code.

```
| ````<LANGUAGE>
| Lignes de codes
| ````
```

## 12.2.6 TABLEAUX

Les tableaux sont créés en utilisant des barres verticales | et des tirets - pour délimiter les colonnes et les lignes.

Titre 1	Titre 2
- - - - -	- - - - -
Ligne 1	Donnée 1
Ligne 2	Donnée 2

## 12.2.7 SAUT DE LIGNE

Pour forcer un saut de ligne, vous pouvez ajouter deux espaces à la fin de la ligne avant de presser "Entrée".

## 12.2.8 CITATIONS

```
| > Ceci est une citation.
```

## 12.2.9 SÉQUENCES D'ÉCHAPPEMENT

Si vous avez besoin d'afficher des caractères spéciaux (comme des astérisques, des dièses, etc.), utilisez un backslash \ devant ces caractères.

```
| \*Texte entre astérisques\*
```

## 12.3 EXEMPLE COMPLET DE FICHIER .MD

### 12.3.1 CODE MARKDOWN

```
# Mon Projet

## Description

**Mon projet** est un projet open source que j'ai créé pour apprendre la programmation en Java.

### Fonctionnalités

- Fonction 1
- Fonction 2
- Fonction 3

## Installation

1. Clonez le dépôt :
`git clone https://github.com/username/projet.git`

2. Allez dans le répertoire :
`cd projet`

## Contribution

1. Forkez le projet.
2. Créez une nouvelle branche pour votre fonctionnalité.
3. Faites une Pull Request.

## Liens utiles

- [GitHub](https://github.com)
- [Mon blog](https://monblog.com)

> "La simplicité est la sophistication suprême." - Léonard de Vinci

### Code Extrait

Voici un exemple de code simple en Java :

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Bonjour, Monde!");
    }
}
```

```

## 12.3.2 AFFICHAGE SUR GITHUB

The screenshot shows a dark-themed GitHub README page. At the top, there's a navigation bar with a 'README' tab and other icons. Below it, the title 'Mon Projet' is displayed. The page is structured with sections: 'Description', 'Fonctionnalités', 'Installation', 'Contribution', 'Liens utiles', and 'Code Extrait'. The 'Description' section contains the text: 'Mon projet est un projet open source que j'ai créé pour apprendre la programmation en Java.' The 'Fonctionnalités' section lists three bullet points: 'Fonction 1', 'Fonction 2', and 'Fonction 3'. The 'Installation' section provides two steps with terminal commands: 'git clone https://github.com/username/projet.git' and 'cd projet'. The 'Contribution' section outlines three steps: 'Forkez le projet.', 'Créez une nouvelle branche pour votre fonctionnalité.', and 'Faites une Pull Request.'. The 'Liens utiles' section links to 'GitHub' and 'Mon blog'. A quote from Leonardo da Vinci is present: '"La simplicité est la sophistication suprême." – Léonard de Vinci'. The 'Code Extrait' section shows a simple Java code snippet:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Bonjour, Monde!");  
    }  
}
```

## 12.4 FICHIER README DE PROFIL GITHUB

### 12.4.1 MARCHE À SUIVRE

Pour que le fichier **README** apparaisse sur votre profil GitHub, vous devez **créer un dépôt avec le même nom que votre nom d'utilisateur GitHub**. (Laissez le dépôt en **public**)  
Ensuite, mettre un fichier README.md dans le dépôt. Maintenant vous avez une présentation sur votre profil GitHub

The screenshot shows the GitHub interface for creating a new repository. It includes fields for 'Owner \*' (set to 'MAN-Luca') and 'Repository name \*' (set to 'MAN-Luca'). A note below says 'MAN-Luca is available.' A message at the bottom explains: 'MAN-Luca/MAN-Luca is a ⚡ special ⚡ repository that you can use to add a README.md to your GitHub profile. Make sure it's public and initialize it with a README to get started.'

## 12.4.2 EXEMPLE DE PRÉSENTATION DE PROFIL GITHUB

```
# Bonjour, je suis Luca Mansutti 🤙
```

Je suis un développeur passionné par la programmation Java, l'utilisation de Git/GitHub et la programmation en assembleur pour microcontrôleurs.

```
## 🛠 Compétences
```

- \*\*Langages\*\* : Java, Python, Assembleur (PIC16F84)
- \*\*Outils\*\* : Git, GitHub, Eclipse, MPLABX
- \*\*Autres\*\* : Documentation technique, Gestion de projet

```
## 🚀 Projets
```

- [Pong en Java](<https://github.com/username/pong>) : Jeu rétro avec des fonctionnalités avancées.
- [Carnet de contacts](<https://github.com/username/carnet>) : Application de gestion de contacts en Java.

```
## 💬 Me contacter
```

- \*\*Email\*\* : [luca.mansutti@example.com](mailto:luca.mansutti@example.com)
- \*\*LinkedIn\*\* : [Luca Mansutti](<https://linkedin.com/in/luca-mansutti>)

The screenshot shows a GitHub profile page for a user named 'MAN-Luca'. On the left, there's a circular profile picture of a lighthouse at night. Below it, the name 'Mansutti Luca' and the handle 'MAN-Luca' are displayed. There are buttons for 'Edit profile' and contact information ('Brussels, Belgium' and email 'luca.mansutti7@gmail.com'). A 'Achievements' section is also visible.

The main content area displays the README.md file:

```
MAN-Luca / README.md
```

**Bonjour, je suis Luca Mansutti 🤙**

Je suis un développeur passionné par la programmation Java, l'utilisation de Git/GitHub et la programmation en assembleur pour microcontrôleurs.

**🛠 Compétences**

- Langages : Java, Python, Assembleur (PIC16F84)
- Outils : Git, GitHub, Eclipse, MPLABX
- Autres : Documentation technique, Gestion de projet

**🚀 Projets**

- [Pong en Java](#) : Jeu rétro avec des fonctionnalités avancées.
- [Carnet de contacts](#) : Application de gestion de contacts en Java.

**💬 Me contacter**

- Email : [luca.mansutti@example.com](mailto:luca.mansutti@example.com)
- LinkedIn : [Luca Mansutti](https://linkedin.com/in/luca-mansutti)



---

## PARTIE 2 : CONCEPTS AVANCÉS

---

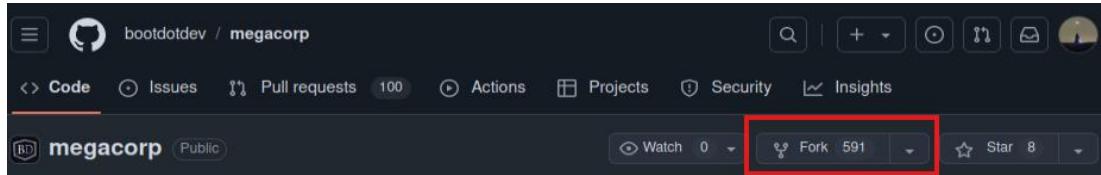


# 13 FORK

## 13.1 QU'EST-CE QU'UN FORK ?

Un fork est une **copie d'un dépôt Git**, permettant de modifier un projet sans impacter l'original et de proposer des changements via des pull requests. "Forker" un repository permet donc de ramener une copie de celui-ci au sein de **vos propres repositories**

## 13.2 COMMENT FORK UN REPOSITORY



Ensuite, le repository est copié dans vos repositories personnels

## 13.3 EXEMPLE RÉEL DE TRAVAIL COLLABORATIF

### 13.3.1 CLONER LE DÉPÔT "FORKÉ"

```
git clone <REPOSITORY_URL>
```

### 13.3.2 CRÉER UNE NOUVELLE BRANCHE POUR LES NOUVELLES FONCTIONNALITÉS

```
git switch -c <BRANCH>
```

### 13.3.3 AJOUTER ET COMMIT LES MODIFICATIONS

```
git add .
git commit -m "MESSAGE"
```

### 13.3.4 POUSSER LES CHANGEMENTS VERS GITHUB

```
git push -u origin <BRANCH>
```

**Note :** Il est conseillé de push vers une branche autre que main

### 13.3.5 CRÉER UN PULL REQUEST

Ce pull request se fait sur **votre repository forké (B)**, mais les changements devront être amenés sur **le repository de base (A)**



# 14 REFLOG

## 14.1 HEAD

**HEAD** représente l'**endroit où vous vous trouvez actuellement** dans votre dépôt. Plus précisément, HEAD est un pointeur qui indique :

- **La branche active** sur laquelle vous travaillez.
- **Le dernier commit** référencé par cette branche.

### 14.1.1 VOIR OÙ POINTE HEAD

```
cat .git/HEAD
```

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git branch
* main
lucam@lucam-VirtualBox:~/Documents/megacorp$ cat .git/HEAD
ref: refs/heads/main
```

## 14.2 REFLOG COMMAND

```
git reflog
```

### 14.2.1 EXPLICATIONS DE REFLOG

La commande **git reflog** (abréviation de **reference log**) est un outil qui enregistre l'historique des changements effectués sur une référence (comme HEAD ou une branche). Contrairement à **git log**, qui affiche les commits dans l'ordre chronologique, **git reflog** se concentre sur **les déplacements récents de HEAD** ou d'autres références.

### 14.2.2 FONCTIONNEMENT DE REFLOG

Chaque fois que HEAD (ou une branche) change d'état ou de position (par exemple, lors d'un commit, d'un merge, ou d'un reset), Git l'enregistre dans le reflog. Cela vous permet de retrouver des commits perdus ou de suivre les actions effectuées dans votre dépôt

|           |                                      |
|-----------|--------------------------------------|
| HEAD@ {0} | Position actuelle de HEAD            |
| HEAD@ {0} | Position de HEAD 1 action en arrière |
| HEAD@ {2} | Position de HEAD 2 action en arrière |
| HEAD@ {3} | Position de HEAD 3 action en arrière |

### 14.2.3 LOG VS REFLOG

| git log                                   | git reflog  |
|---|---|
| Historique des commits                    | Historique des mouvements de HEAD                               |
| Affiche les commits visibles              | Inclut même les commits "perdus" ou cachés                      |
| Format axé sur les hashes et les branches | Format basé sur <code>HEAD@ { }</code> pour retracer les étapes |

## 14.3 RÉCUPÉRATION DU CONTENU D'UN FICHIER SUPPRIMÉ

**git reflog** est un outil puissant pour retracer les actions dans un dépôt Git. Il est particulièrement utile pour **récupérer des commits perdus** ou **comprendre les modifications de HEAD**.

### 14.3.1 ÉTAPES À SUIVRE POUR RÉCUPÉRER LE CONTENU D'UN FICHIER SUPPRIMÉ

#### 14.3.1.1 VÉRIFIER L'HISTORIQUE AVEC REFLGOG

**git reflog**

Permet de retrouver le **COMMIT\_HASH**

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git reflog
0d16f95 (HEAD -> main, origin/main, origin/HEAD) HEAD@{0}: checkout: moving from
  slander to main
452f673 HEAD@{1}: commit: B: Slander
```

#### 14.3.1.2 EXPLORER LE COMMIT AVEC GIT CAT-FILE -P

**git cat-file -p <COMMIT\_HASH>**

Une fois que vous avez le hash du commit, inspectez son contenu pour retrouver le **tree\_hash**

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git cat-file -p 452f673
tree 15b6ed42d6cdd0b452eb64c9698c0a36699d654a
parent 0d16f954dfda53be7e875da6a23698c428cf68af
author MAN-Luca <luca.mansutti7@gmail.com> 1732099025 +0100
committer MAN-Luca <luca.mansutti7@gmail.com> 1732099025 +0100
```

#### 14.3.1.3 EXPLORER LE TREE

**git cat-file -p <TREE\_HASH>**

Une fois que vous avez le hash du tree inspectez son contenu pour retrouver le **blob\_hash**

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git cat-file -p 15b6ed42d6cdd0b
100644 blob 13b4fae17d1b50828697a36c4ab337039018a35e README.md
040000 tree 33b4479084b4e02df83af65191b3d1a87db008b0 contributors
040000 tree f408a5791812c598240e2e5e30759675a33957e8 customers
040000 tree 3f24c2dac3af70dc3af6713f686d4f57d9608a42 orgs
040000 tree 87e79e11ffc811016818fe57ab91fac55cb5a8b8 scripts
100644 blob 30fbe30f2546d07d072707dc4fc46079ee26ff09 slander.md
```

#### 14.3.1.4 EXAMINER LE BLOB POUR RETROUVER LE CONTENU DU FICHIER

**git cat-file -p <blob\_hash>**

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git cat-file -p 30fbe30f254
# Breaking News
```

## 14.4 RÉCUPÉRATION AMÉLIORÉE AVEC MERGE

La méthode manuelle avec cat-file est fastidieuse. Heureusement nous pouvons remplacer toutes ces étapes par une seule commande

### 14.4.1 MERGE AVEC UN COMMITISH

```
git merge <COMMITISH>
```

**Note** : commitish = HEAD@{n}

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git reflog
0d16f95 (HEAD -> main, origin/main, origin/HEAD) HEAD@{0}: checkout: moving from
slander_to_main
452f673 HEAD@{1}: commit: B: Slander
```

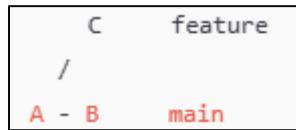
```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git merge HEAD@{1}
```

# 15 MERGE CONFLICTS

## 15.1 INTRODUCTION AUX CONFLITS

Les conflits se produisent lorsque plusieurs développeurs modifient les **mêmes lignes** dans un fichier ou lorsque des modifications contradictoires sont apportées sur des branches différentes **sans relation parent-enfant**. Cela arrive souvent lors des **fusions** ou des **rebases**.

### 15.1.1 EXEMPLE DE CONFLIT



- Fichier "number.js" dans la branche main (commit B) :

```
package main

func isNice(num int) bool {
    return num == 69 // commit B a changé cette ligne
}
```

- Fichier "number.js" dans la branche feature (commit C) :

```
package main

func isNice(num int) bool {
    return num == 420 // commit C a changé cette ligne
}
```

Si vous tentez de fusionner "feature" dans "main", Git détecte que la **même ligne** a été modifiée dans les deux branches **sans relation parent-enfant**. Cela crée un **conflit de fusion**.

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git merge main
Auto-merging number.js
CONFLICT (add/add): Merge conflict in number.js
Automatic merge failed; fix conflicts and then commit the result.
```

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git status
On branch feature
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 added:       number.js
```

## 15.2 RÉSoudre un conflit

### 15.2.1 LOCALISER LE CONFLIT

Lorsque Git détecte un conflit, il marque le fichier concerné avec des **marqueurs de conflit**

```
package main

func isNice(num int) bool {
<<<<< HEAD
    return num == 420 // commit C changed this line
=====
    return num == 69 // commit B changed this line
>>>>> main
}
```

La première section entre <<<<< HEAD et ====== représente **la branche dans laquelle vous êtes**.

La seconde section entre ====== et >>>>> main représente **la branche avec laquelle vous voulez faire le merge**

### 15.2.2 ÉDITER LES FICHIER ET RÉSoudRE LES CONFLITS

Supprimez les marqueurs et **combinez les deux** modifications, où bien **supprimez une des deux**)

```
package main

func isNice(num int) bool {
    return (num == 420 && num == 69)
}
```

### 15.2.3 AJOUTER ET FINALISER LA FUSION

```
git add .
git commit -m "E : Résolution des conflits"
```

**Note** : Lors d'un conflit de merge, git rentre dans un mode spécial. Si les conflits sont résolus, alors le prochain commit clôturera le merge des branches.

### 15.2.4 VÉRIFIER LE MERGE

```
git log
```

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git log
commit cf7357d89660def3a632f19ec2e6af66ba70abcd (HEAD -> add_customers)
Merge: b46b771 45a1d7e
Author: MAN-Luca <luca.mansutti7@gmail.com>
Date:   Wed Nov 20 14:06:04 2024 +0100

    E: resolution de conflit
```

## 15.3 ARRÊTER UN MERGE LORS D'UN CONFLIT

```
git merge --abort
```

Note :

- Cette commande annule la fusion en cours et restaure l'état de votre dépôt comme il était avant le début de la fusion.
- Elle supprime tous les fichiers conflictuels ou partiellement modifiés introduits par la tentative de merge.

## 15.4 OUTILS DE RÉSOLUTION DE CONFLITS INTÉGRÉE À GIT

### 15.4.1 "OURS" VS "THEIRS" (NOTRE VS LEURS)

Lorsque vous êtes en conflit de fusion dans Git, les termes "**ours**" et "**theirs**" désignent les branches impliquées dans le conflit :

- "**Ours**" : cela fait référence à la branche sur **laquelle vous êtes** actuellement (la branche dans laquelle vous êtes en train de fusionner).
- "**Theirs**" : cela fait référence à la branche que **vous fusionnez dans votre branche actuelle**.

### 15.4.2 CHECKOUT MERGE COMMAND

```
git checkout --theirs/--ours <PATH>
```

Note :

- **--ours** va écraser le fichier avec les modifications **de la branche sur laquelle vous êtes** actuellement et vers laquelle vous fusionnez.
- **--theirs** va écraser le fichier avec les modifications **de la branche que vous êtes en train de** fusionner avec la branche courante.

Une fois les problèmes résolus nous pouvons faire :

```
git add .
git commit -m "MESSAGE"
```

# 16 REBASE CONFLICTS

## 16.1 EXEMPLE DE REBASE CONFLICTS

### 16.1.1 SCÉNARIO :

- Vous travaillez sur une branche `banned` et souhaitez la mettre à jour avec les dernières modifications de `main`. Vous utilisez un *rebase*. Pendant ce temps, d'autres ont ajouté des modifications sur `main`

Premier fichier `banned.csv` de la branche "banned" contenant :

```
first_name,last_name,company,title  
Kayha,tbd,TheMarchOfTime,sidekick  
sam,ctrlman,closedai,ceo
```

Second fichier `banned.csv` de la branche "main" contenant :

```
first_name,last_name,company,title  
Ballan,Agrandian,Boots.lore,Protagonist  
sam,ctrlman,closedai,ceo
```

- En *rebasant*, vous essayez d'ajouter les changements de `banned` au-dessus de `main`, ce qui entraîne un conflit si les mêmes fichiers ont été modifiés dans les deux branches.

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git rebase main  
Auto-merging customers/banned.csv  
CONFLICT (add/add): Merge conflict in customers/banned.csv  
error: could not apply d49e040... I: ajout de banned  
hint: Resolve all conflicts manually, mark them as resolved with  
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".  
hint: You can instead skip this commit: run "git rebase --skip".  
hint: To abort and get back to the state before "git rebase", run "git rebase  
abort".  
Could not apply d49e040... I: ajout de banned
```

- Vous devez résoudre ce conflit dans un état spécial de Git appelé ***detached HEAD***.

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git branch  
* (no branch, rebasing banned)  
  banned  
  main
```

### 16.1.2 L'ÉTAT "DETACHED HEAD"

Lors d'un *rebase*, le HEAD montre la branche cible (`main` dans cet exemple), et non la branche source (`banned`). Vous êtes dans un état de "***detached HEAD***", vous permettant de résoudre les conflits avant de finaliser le processus de *rebase*. Cela signifie que vous n'êtes plus sur une branche spécifique mais sur un commit temporaire Résoudre les rebase conflicts

### 16.1.3 CHECKOUT REBASE COMMAND

```
git checkout --theirs/--ours <PATH>
```

**Attention :**

- **--ours** va **conserver** les modifications de la **branche en cours de rebase**
- **--theirs** va remplacer le contenu du fichier par les **modifications de la branche cible (main)**

### 16.1.4 TERMINER LE REBASE

```
git rebase --continue
```

**Note :** Contrairement aux merge conflicts qui nécessitent de commit pour terminer les modifications, le rebase nécessite **-continue**

# 17 RERERE (REUSE RECORDED RESOLUTION)

## 17.1 QU'EST-CE QUE GIT RERERE

Dans Git, les conflits répétitifs peuvent être frustrants, surtout lorsque vous travaillez sur des branches de fonctionnalités qui nécessitent un **rebase** ou un **merge** fréquent avec main.

Heureusement, Git propose une solution appelée rerere, qui signifie **reuse recorded resolution** :

- `rerere` enregistre la manière dont vous avez résolu un conflit spécifique.
- Si Git rencontre le **même conflit** à nouveau, il applique automatiquement la résolution que vous aviez utilisée précédemment.

Cela fonctionne avec les **rebases** et les **merges**

### 17.1.1 ACTIVER / DÉSACTIVER RERERE

```
git config [--local | --global] rerere.enabled {TRUE|FALSE}
```

### 17.1.2 SUPPRIMER LE DOSSIER RERERE

```
rm -rf .git/rr-cache
```

## 17.2 EXEMPLE D'UTILISATION DE RERERE

Non avons 3 branches : **main**, **favs** et **favs2**

Le but est de résoudre les conflits manuellement lors du rebase de main sur favs, ensuite de rebase main sur favs2. Pour le second rebase rerere permettra de ne pas devoir faire les changements manuellement, car il aura enregistré la manière de résoudre ce type de conflits.

### 17.2.1 PREMIER REBASE DE MAIN SUR FAVS (ENTRÉE DANS LA ZONE DE CONFLIT)

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git switch favs
Switched to branch 'favs'
lucam@lucam-VirtualBox:~/Documents/megacorp$ git rebase main
Auto-merging customers/favs.md
CONFLICT (add/add): Merge conflict in customers/favs.md
error: could not apply 97da6d6... K: added favs
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase
abort".
Recorded preimage for 'customers/favs.md'
Could not apply 97da6d6... K: added favs
```

**Note** : La ligne encadrée montre que Git enregistre la manière de gérer ce type de conflit.

## 17.2.2 RÉSOLUTION DES CONFLITS ET FERMETURE DU REBASE

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git add .
lucam@lucam-VirtualBox:~/Documents/megacorp$ git rebase --continue
Recorded resolution for 'customers/favs.md'.
[detached HEAD fb3d243] K: added favs
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/favs.
```

**Note :** La ligne encadrée montre que Git a enregistré la manière de gérer ce type de conflit.

## 17.2.3 REBASE DE MAIN SUR FAVS2 (RÉSOLUTION AUTOMATIQUE)

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git switch favs2
Switched to branch 'favs2'
lucam@lucam-VirtualBox:~/Documents/megacorp$ git rebase main
Auto-merging customers/favs.md
CONFLICT (add/add): Merge conflict in customers/favs.md
error: could not apply 97da6d6... K: added favs
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Resolved 'customers/favs.md' using previous resolution.
Could not apply 97da6d6... K: added favs
```

**Note :** La ligne encadrée montre que Git a résolu tout seul le conflit. Nous n'avons donc pas du gérer les conflits manuellement

## 17.2.4 FERMETURE DU REBASE

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git add .
lucam@lucam-VirtualBox:~/Documents/megacorp$ git rebase --continue
[detached HEAD fba80a7] K: added favs
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/favs2.
```

## 17.3 ANNULER UN COMMIT (ALORS QU'IL FALLAIT --CONTINUE)

### 17.3.1 PROBLÈME COURANT

Lors de la résolution d'un conflit pendant un **rebase**, il est facile d'oublier que l'on doit utiliser **git rebase --continue** au lieu de créer un commit. Si vous "commitez" accidentellement la résolution, Git enregistre ce commit dans l'historique du rebase, ce qui peut causer des incohérences.

### 17.3.2 ANNULER UN COMMIT ACCIDENTEL

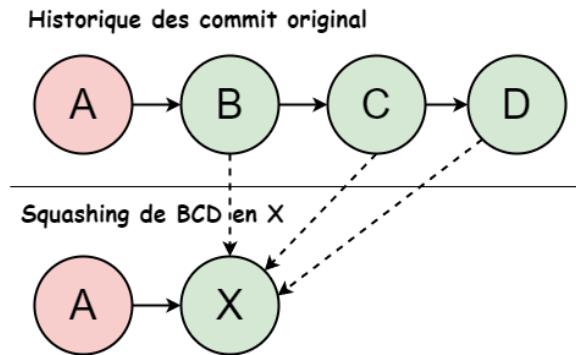
```
git reset --soft HEAD~1
```

**Note :** L'option **--soft** préserve vos modifications dans le staging area, sans supprimer les résolutions de conflit.

# 18 SQUASH

## 18.1 QU'EST-CE QUE LE SQUASHING ?

Le squashing consiste à combiner plusieurs commits en un seul. Cela permet de simplifier l'historique Git et de le rendre plus lisible



**Attention :** Le squashing est **destructif**, il faut donc toujours le faire sur une **branche temporaire** (temp\_main), puis remplacer main par la nouvelle branche temp\_main et pour finir renommer temp\_main en main

## 18.2 COMMENT FAIRE UN SQUASH ?

### 18.2.1 DÉMARRER UN REBASE INTERACTIF

```
git rebase -i HEAD~n
```

**Note :** Remplacez n par le nombre de commits que vous voulez inclure dans le squash.

Pour **inclure le commit initial** : `git rebase -i --root`

### 18.2.2 CHOISIR LES COMMITS À SQUASH

- Une fois la commande exécutée, une interface interactive s'ouvre dans votre éditeur par défaut. Chaque ligne correspond à un commit récent, comme ci-dessous :

```
pick 9238ca5 I: ajout de banned
pick fc2475d L: adding favs
pick fb3d243 K: added favs

# Rebase fad8ba1..fb3d243 onto fad8ba1 (3 commands)
```

- Changez **pick** en **squash** (ou simplement **s**) pour tous les commits que vous voulez fusionner avec le premier. Par exemple :

```
pick 9238ca5 I: ajout de banned
squash fc2475d L: adding favs
squash fb3d243 K: added favs

# Rebase fad8ba1..fb3d243 onto fad8ba1 (3 commands)
```

- Après avoir sauvegardé, Git vous demandera de rédiger le message pour le nouveau commit unique (supprimer les messages des anciens commits que vous voulez squash)

```
# This is a combination of 3 commits.
# This is the 1st commit message:
I: ajout de banned
# This is the commit message #2:
L: adding favs
# This is the commit message #3:
K: added favs
```

```
# This is a combination of 3 commits.
# This is the 1st commit message:
I: Squash de K et L
```

- Le squashing à bien été effectué

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git rebase -i HEAD~3
[detached HEAD 6732e10] I: Squash de K et L
Date: Wed Nov 20 15:01:31 2024 +0100
2 files changed, 4 insertions(+), 1 deletion(-)
create mode 100644 customers/favs.md
Successfully rebased and updated refs/heads/temp_main.
lucam@lucam-VirtualBox:~/Documents/megacorp$ git log --oneline
6732e10 (HEAD -> temp_main) I: Squash de K et L
```

## 18.3 OVERWRITE MAIN

Maintenant que nous avons fini d'actualiser la branche temporaire (temp\_main), nous pouvons remplacer la branche main par celle-ci

**Note :** Cette méthode n'est pas recommandée. La méthode classique est de créer une pull-request ([voir 16.6](#))

### 18.3.1 SUPPRIMER LA BRANCHE MAIN

```
git branch -D main
```

### 18.3.2 RENOMMER LA BRANCHE TEMPORAIRE PAR MAIN

```
git branch -m main
```

## 18.4 DANGERS DU SQUASHING

Squasher les commits peut être intimidant, car cela efface l'historique des modifications. Par exemple, si vous avez une série de commits comme :A - B - C - D

et que vous les réduisez en un seul commit : ABCD

vous supprimez les repères de chaque modification individuelle. Bien que toutes les modifications soient toujours présentes dans l'historique final, les points de contrôle individuels sont supprimés. Cela signifie que, une fois le squash effectué, vous ne pourrez plus revenir aux commits précédents pour examiner l'évolution du projet à chaque étape.

## 18.5 SQUASHING PULL REQUESTS

### 18.5.1 RÉSUMÉ DU FLUX DE TRAVAIL

1. Créez une nouvelle branche à partir de main.
2. Travaillez sur la branche de fonctionnalité en ajoutant des commits au fur et à mesure.
3. Une fois prêt, combinez tous vos commits en un seul.
4. Poussez la branche vers le dépôt distant.
5. Ouvrez une pull request depuis votre branche de fonctionnalité vers main.
6. Une fois la pull request approuvée, effectuez la fusion.

## 18.6 FORCE PUSH

### 18.6.1 QU'EST-CE QU'UN FORCE PUSH

Le force push nécessaire lorsque l'historique de votre branche locale est incompatible avec celui de la branche distante. Cela peut arriver après des opérations comme un **rebase** ou un **squash**, qui modifient l'ordre ou le contenu des commits. Dans ces situations, Git bloque un push classique afin de protéger l'historique de la branche distante.

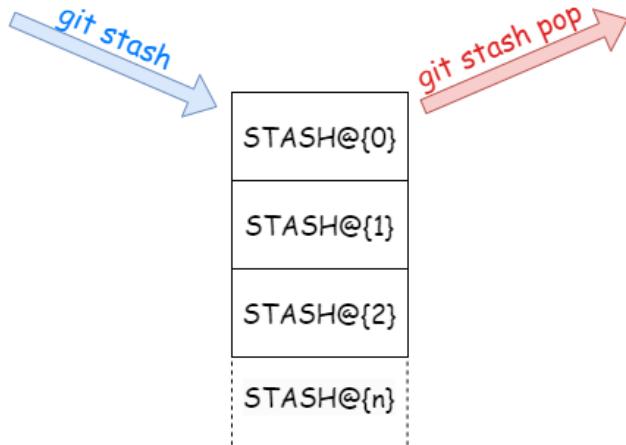
### 18.6.2 FORCE PUSH COMMAND

```
git push origin main --force
```

# 19 STASH

## 19.1 A QUOI SERT LE STASHING ?

La commande `git stash` enregistre l'état actuel de votre répertoire de travail et de l'index (zone de staging). Cela fonctionne un peu comme un presse-papiers : elle place temporairement les modifications de côté et revient à l'état du dernier commit de votre branche.



## 19.2 STASH COMMAND

### 19.2.1 CRÉER UN STASH

```
git stash [-m "MESSAGE"]
```

### 19.2.2 LISTER LES STASHES

```
git stash list
```

### 19.2.3 STASH POP

La commande `git stash pop` permet de réappliquer les modifications que vous avez "stash" précédemment et de les supprimer de la liste des stashes.

```
git stash pop stash@{n}
```

**Note** : Permet de réappliquer vos changements à votre répertoire de travail et **supprimer** l'entrée du stash

### 19.2.4 STASH APPLY

```
git stash apply stash@{n}
```

**Note** : Permet de réappliquer vos changements à votre répertoire de travail **MAIS ne supprime pas** l'entrée du stash

### 19.2.5 STASH DROP

```
git stash drop stash@{n}
```

**Note** : Permet de supprimer une entrée du stash **sans y retourner**

# 20 DIFF

## 20.1 DIFF COMMAND

La commande `git diff` vous permet de voir les différences entre différents états de votre code..

### 20.1.1 COMPARER L'ÉTAT ACTUEL ET LE DERNIER COMMIT

```
git diff
```

### 20.1.2 COMPARER UN COMMIT PRÉCÉDENT ET L'ÉTAT ACTUEL

```
git diff HEAD~{n}
```

### 20.1.3 COMPARER DEUX COMMITS SPÉCIFIQUES

```
git diff <COMMIT_HASH_1> <COMMIT_HASH_2>
```

### 20.1.4 COMPARER L'INDEX (ZONE DE STAGING) AVEC LA DERNIÈRE VALIDATION

```
git diff --cached
```

### 20.1.5 COMPARER L'ARBRE DE TRAVAIL AVEC L'INDEX

```
git diff HEAD
```

## EXEMPLE D'UTILISATION DE DIFF

Commit "Ajout helloworld.py"

```
for i in range(10):
    print("Hello world : ", i)
```

Commit "Update helloworld.py"

```
for i in range(10):
    print("Hello world : ",i)
```

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git log --oneline -n 2
834dfe2 (HEAD -> main) Update helloworld.py
b1dbee8 Ajout helloworld.py
lucam@lucam-VirtualBox:~/Documents/megacorp$ git diff 834dfe2 b1dbee8
diff --git a/hello_world.py b/hello_world.py
index 154a4fc..ee074fe 100644
--- a/hello_world.py
+++ b/hello_world.py
@@ -1,5 +1,5 @@
-    print("Hello world : ",i)
+    print(i, ": Hello world")
```

## 20.2 BLAME COMMAND

La commande git blame permet d'identifier l'auteur et l'origine de chaque ligne dans un fichier.

```
git blame [OPTIONS] <FILE>
```

**Options :**

- **-L <start>,<end>** : Affiche uniquement les lignes spécifiées (ex. lignes 5 à 15).
- **-e** : Montre l'adresse email de l'auteur en plus de son nom.
- **-C** : Suit les changements dans les fichiers copiés.
- **-M** : Suit les lignes déplacées ou renommées dans un fichier.

### 20.2.1 EXEMPLE DE SORTIE DE GIT BLAME

```
e94a23fc (John Doe 2023-11-01 12:45:32 +0100) if est_gentil:  
e94a23fc (John Doe 2023-11-01 12:45:32 +0100) print("Merci")  
a5d67b2d (Jane Smith 2023-10-29 15:10:12 +0100) return 0
```

Informations affichées

- **Commit hash** : Identifie le commit responsable de la ligne (les 8 premiers caractères suffisent en général).
- **Auteur** : La personne ayant modifié la ligne.
- **Date** : Quand la ligne a été modifiée.
- **Contenu** : Le texte exact de la ligne

## 21 CHERRY PICK

### 21.1 QUE FAIT LA COMMANDE CHERRY-PICK ?

**Extrait un commit spécifique** d'une branche sans pour autant merge ou rebase toute la branche

- **Rejoue les changements** du commit spécifié sur la branche actuelle.
- Crée un **nouveau commit** avec le même contenu mais un nouvel identifiant, tout en conservant l'historique de la modification.

### 21.2 CHERRY-PICK COMMAND

`git cherry-pick <COMMIT_HASH>`

En cas de conflits, vous devrez résoudre manuellement les différences, ajouter les fichiers résolus avec `git add`, puis continuer avec :

`git cherry-pick --continue`

Si nécessaire, annulez l'opération avec :

`git cherry-pick --abort`

## 22 BISECT

### 22.1 A QUOI SERT LE BISECT ?

`git bisect` utilise une **recherche dichotomique** pour **identifier un commit spécifique**, ce qui réduit considérablement le nombre de vérifications nécessaires.

**Bisect** est utilisé pour :

- **Bugs** : Identifier le commit qui a introduit un comportement erroné.
- **Régressions** : Trouver le commit à l'origine d'une baisse de performance ou d'un problème.
- **Changements non souhaités** : Localiser la source d'une modification indésirable.

### 22.2 MARCHE À SUIVRE DU BISECT

#### 22.2.1 COMMENCER LE BISECT

```
git bisect start
```

#### 22.2.2 MARQUER UN COMMIT FONCTIONNEL (BON COMMIT)

```
git bisect good <COMMIT_HASH>
```

#### 22.2.3 MARQUER UN COMMIT PROBLÉMATIQUE (MAUVAIS COMMIT)

```
git bisect bad <COMMIT_HASH>
```

#### 22.2.4 TESTER LE COMMIT INTERMÉDIAIRE

- Git vous positionne sur un commit entre les points "bon" et "mauvais".
- Testez si le bug est présent.
- Indiquez le résultat (étape suivante)

#### 22.2.5 DÉCLARER LE RÉSULTAT DU TEST

```
git bisect good  # Le commit est "bon".
git bisect bad   # Le commit est "mauvais".
```

#### 22.2.6 RÉPÉTER LES 2 ÉTAPES PRÉCÉDENTES

Git continue d'affiner la recherche en testant les commits restants jusqu'à identifier celui qui a introduit le bug.

#### 22.2.7 TERMINER LA SESSION DE BISECT

Une fois le commit trouvé, quitter le mode bisect :

```
git bisect reset
```

## **23 WORKTREE**

### **23.1 QU'EST-CE QU'UN WORKTREE ?**

Un **worktree** est une arborescence de travail associée à un dépôt Git. C'est un espace où vous pouvez manipuler les fichiers suivis par Git. Chaque dépôt possède :

- Un **main worktree** (principal) : là où se trouve le répertoire .git.
- D'éventuels **linked worktrees** (liés) : des arborescences légères, connectées au dépôt principal.

### **23.2 TYPES DE WORKTREES**

#### **23.2.1 MAIN WORKTREE**

- Contient le répertoire .git avec **toutes les données du dépôt**.
- Consomme plus d'espace disque car il contient tout l'historique Git.
- Nécessite un **git clone** ou un **git init** pour être créé.

#### **23.2.2 LINKED WORKTREE**

- Contient un fichier .git qui pointe vers le main worktree.
- Très léger, comparable à une branche.
- Utile pour travailler sur plusieurs branches ou ensembles de modifications sans perdre de temps à stasher ou changer de branche.

### **23.3 WORKTREE COMMANDS**

#### **23.3.1 LISTER TOUS LES WORKTREES ASSOCIÉS À UN REPOSITORY**

```
git worktree list
```

#### **23.3.2 CRÉER UN LINKED WORKTREE**

```
git worktree add <PATH> [<BRANCH>]
```

**Note** : <path> : emplacement du nouveau worktree. <branch> (optionnel) : branche à utiliser. Si omis, utilise le dernier segment du chemin comme nom de branche.

#### **23.3.3 AFFICHER LE CONTENU DU FICHIER .GIT D'UN LINKED WORKTREE**

```
cat <PATH> >/ .git
```

## 23.4 PAS DE BRANCHES PARTAGÉES ENTRE WORKTREES !

Les **linked worktrees** se comportent comme des dépôts Git normaux :

- Vous pouvez créer/supprimer des branches, passer d'une branche à une autre, créer des tags, etc.

Cependant, une restriction importante s'applique :

**Une branche ne peut être utilisée simultanément par plusieurs worktrees**

Exemple : Main worktree (**megacorp**) et linked worktree (**ultracorp**)

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ git worktree add ultracorp
Preparing worktree (checking out 'ultracorp')
HEAD is now at b7a9940 0: updated partners.txt
lucam@lucam-VirtualBox:~/Documents/megacorp$ cd ultracorp/
lucam@lucam-VirtualBox:~/Documents/megacorp/ultracorp$ git branch
+ main
* ultracorp
lucam@lucam-VirtualBox:~/Documents/megacorp/ultracorp$ git switch main
fatal: 'main' is already checked out at '/home/lucam/Documents/megacorp'
```

Résultat : Vous ne pouvez pas utiliser main dans deux worktrees en même temps.

Solution : Travailler sur une nouvelle branche

## 23.5 TRACKING DES LINKED WORKTREES

Le dossier **.git/worktrees** centralise les informations de tracking pour chaque linked worktree. Cela permet à Git de les gérer efficacement tout en maintenant une connexion au dépôt principal.

**ls .git/worktrees**

Note : Le dossier .git/worktrees contient un sous-dossier pour chaque linked worktree.

```
lucam@lucam-VirtualBox:~/Documents/megacorp$ ls .git/worktrees/
ultracorp
```

## 23.6 MAIN ET LINKED WORKTREES : SYSTÈME UNIQUE

Les linked worktrees fonctionnent directement avec le **dépôt principal**, car ils ne possèdent pas de répertoire .git. Cela signifie que toute modification dans un linked worktree est immédiatement reflétée dans le main worktree, et vice-versa.

Lorsque vous exécutez git branch, Git indique si une branche est déjà utilisée dans un autre worktree en affichant un message spécifique à côté de son nom. Cela vous permet de savoir quelle branche est actuellement "occupée" par un autre worktree.

## 23.7 SUPPRIMER UN WORKTREE

### 23.7.1 SUPPRIMER UN WORKTREE NE SUPPRIME PAS LA BRANCHE

La commande git worktree remove permet de supprimer un worktree **sans toucher à la branche associée**. Le worktree est uniquement une vue de votre dépôt, et il est séparé des branches réelles.

### 23.7.2 REMOVE COMMAND

```
git worktree remove <WORKTREE-NAME>
```

### 23.7.3 PRUNE COMMAND

Pour nettoyer complètement les références aux worktrees supprimés, vous pouvez utiliser la commande :

```
git worktree prune
```

**Note** : Cela va supprimer toute référence restante aux worktrees supprimés dans le dossier .git/worktrees de votre dépôt.

# 24 TAGS

## 24.1 QU'EST-CE QU'UN TAG ?

Un **tag** est un marqueur **immobile** lié à un commit spécifique. Contrairement aux branches, les **tags ne changent pas avec de nouveaux commits**. Ils sont principalement utilisés pour **marquer des versions importantes** ou des jalons dans un projet.

**Fun Fact :** Dans Git, vous pouvez utiliser un **tag** à la place d'un **hash de commit** dans la plupart des commandes, car les tags sont une forme de "**commitish**" (une référence vers un commit)



## 24.2 TAG COMMANDS

### 24.2.1 LISTER LES TAGS EXISTANTS

24.2.1.1 *LOCALEMENT*

```
git tag
```

24.2.1.2 *SUR UN DÉPÔT DISTANT*

```
git ls-remote --tags
```

AFFICHER LES DÉTAILS D'UN TAG

```
git show <TAG_NAME>
```

24.2.2 CRÉER UN TAG ANNOTÉ

```
git tag -a "TAG_NAME" <COMMIT_HASH> -m "TAG_MESSAGE"
```

**Note :** L'ajoute du **COMMIT\_HASH** permet de **créer un tag sur un commit spécifique**

24.2.3 SUPPRIMER UN TAG

24.2.3.1 *LOCALEMENT*

```
git tag -d "TAG_NAME"
```

24.2.3.2 *SUR LE DÉPÔT DISTANT*

```
git push origin --delete <TAG_NAME>
```

24.2.4 POUSSER LES TAGS VERS LE DÉPÔT DISTANT

```
git push origin --tags
```



---

## FORMULAIRE DES COMMANDES

---



## 25 COMMANDES FONDAMENTALES

### REPOSITORY

|  |  |
|--|--|
| <code>git init</code>                              | Initialise un dossier comme dépôt git                |
| <code>git status</code>                            | Affiche l'état des fichiers d'un dépôt               |
| <code>git add &lt;FILE   .&gt;</code>              | Ajoute des fichiers au staging                       |
| <code>git rm --cached &lt;FILE   .&gt;</code>      | Retire des fichiers du staging (arrêter de suivre)   |
| <code>git restore --staged &lt;FILE   .&gt;</code> | Retire des fichiers du staging (continuer de suivre) |
| <code>git commit -m "MESSAGE"</code>               | Faire un commit avec message                         |
| <code>git commit -am "MESSAGE"</code>              | Add + commit en une ligne (si fichier déjà suivi)    |
| <code>git commit -amend -m "MESSAGE"</code>        | Modifier message du dernier commit                   |

### LOG

|                                |   |
|--------------------------------|---|
| <code>git log [OPTIONS]</code> | Affiche historique complet des commits et leur hash |
|--------------------------------|---|

|                                |   |
|--------------------------------|---|
| --oneline                      | Affiche les commits d'une manière simplifiée          |
| --graph                        | Affiche l'arborescence des commits                    |
| --parents                      | Affiche les commits hash des parents de chaque commit |
| -n <nombre>                    | Limite l'affiche à x commits                          |
| --all                          | Affiche les commits de TOUTES les branches            |
| --stat -p                      | Affiche un résumé des fichiers modifiés + n° lignes   |
| --decorate=<full   short   no> | Affiche toute/normal/pas de références                |
| <remote>/<branch>              | Affiche l'historique d'une branche distante           |

### FONCTIONNEMENT INTERNE

|  |  |
|--|--|
| <code>ls -al .git/objects</code>                     | Affiche contenu du dossier .git/objects  |
| <code>ls -al .git/objects/&lt;COMMIT_HASH&gt;</code> | Trouve l'objet correspondant à un commit |
| <code>git cat-file -p &lt;HASH&gt;</code>            | Affiche le contenu d'un commit/tree/blob |
| <code>find .git/refs/heads -type f</code>            | Afficher toutes les branches d'un dépôt  |
| <code>cat .git/refs/heads/&lt;BRANCH&gt;</code>      | Afficher le COMMIT_HASH d'une branche    |

## CONFIGURATIONS

|  |                                   |
|--|-----------------------------------|
| <code>git config [options] &lt;SECTION&gt;.&lt;KEY&gt; "valeur"</code> | Gérer les configs globales/locale |
| <code>git config --list</code>   | Consulter config de Git           |

|                              |   |
|------------------------------|---|
| --global                     | Modifie globalement (fichier .gitconfig)  |
| --local                      | Modifie localement (fichier .git local)   |
| --add                        | Ajoute une configuration                  |
| --unset                      | Supprime une configuration                |
| --unset -all <section>.<key> | Supprime toutes les occurrences d'une clé |
| --remove-section <section>   | Supprime toute une section                |

Config de git :

- `user.username "nom"` : Set le nom de l'utilisateur
- `user.email "email"` : Set l'email de l'utilisateur
- `init.defaultBranch "nom_branche"` : Set le nom de la branche par défaut
- `core.editor "nom_éditeur"` : Set l'éditeur par defaut
- `core.ignorecase "boolean"` : Set la casse par défaut

## BRANCHING

|   |   |
|---|---|
| <code>git branch [--list]</code>                                | Afficher toutes les branches d'un dépôt |
| <code>git switch &lt;BRANCH&gt;</code>                          | Changer de branche                      |
| <code>git branch -d / -D &lt;BRANCH&gt;</code>                  | Supprimer une branche                   |
| <code>git branch -m &lt;OLD_NAME&gt; &lt;NEW_NAME&gt;</code>    | Renommer une branche                    |
| <code>git branch &lt;BRANCH&gt; [&lt;COMMIT_HASH&gt;]</code>    | Créer une branche                       |
| <code>git switch -c &lt;BRANCH&gt; [&lt;COMMIT_HASH&gt;]</code> | Créer une branche et y basculer         |

## MERGE

|  |                                  |
|--|----------------------------------|
| <code>git switch &lt;MAIN_BRANCH&gt;</code>              | Aller dans la branche principale |
| <code>git merge &lt;OTHER_BRANCH&gt; -m "MESSAGE"</code> | Merge 2 branches                 |

## REBASE

|  |  |
|--|--|
| <code>git switch &lt;OTHER_BRANCH&gt;</code> | Aller dans la branche secondaire                         |
| <code>git rebase &lt;MAIN_BRANCH&gt;</code>  | Amène les modifications de main vers la branche actuelle |

## RESET

|  |   |
|--|---|
| <code>git reset --soft &lt;COMMIT_HASH&gt;</code>  | Revenir à ancien commit (garde le staging)    |
| <code>git reset --hard &lt;COMMIT_HASH&gt;</code>  | Revenir à ancien commit (supprime le staging) |
| <code>git restore --staged &lt;FILE   .&gt;</code> | Retirer un fichier du staging                 |
| <code>git restore &lt;FILE   .&gt;</code>          | Revenir à un ancien commit                    |
| <code>git revert &lt;COMMIT_HASH&gt;</code>        | Remplace un commit par un nouveau             |

## REMOTE

|   |  |
|---|--|
| <code>git remote add &lt;REMOTE&gt; &lt;REPO_URL&gt;</code> | Ajouter un remote                              |
| <code>git fetch [&lt;REMOTE&gt;]</code>                     | Récupérer les modifications d'un remote        |
| <code>git merge &lt;REMOTE&gt;/&lt;BRANCH&gt;</code>        | Synchroniser le contenu d'un remote localement |

## GITHUB

|   |  |
|---|--|
| <code>gh auth login / logout</code>                                     | S'authentifier/se déco de GitHub avec GH CLI     |
| <code>git remote add origin &lt;REPO_URL&gt;</code>                     | Lier le dépôt local au remote sur GitHub         |
| <code>git ls-remote</code>  | Vérifier la liaison                              |
| <code>git push [-u] origin &lt;BRANCH&gt;</code>                        | Pousser les modifications vers le dépôt distant  |
| <code>git push origin &lt;LOCAL_BRANCH&gt;:&lt;REMOTE_BRANCH&gt;</code> | Pousser une branche locale vers branche distante |
| <code>git push origin :&lt;REMOTE_BRANCH&gt;</code>                     | Supprimer une branche distante (branche vide)    |
| <code>git pull [&lt;REMOTE&gt;/&lt;BRANCH&gt;]</code>                   | Récupérer des modifis du remote dans dépôt local |

## 26 COMMANDES AVANCÉES

### REFLOG

|  |   |
|--|---|
| <code>git reflog</code>                  | Affiche l'historique des déplacements récents     |
| <code>git merge &lt;COMMITISH&gt;</code> | Permet de merge avec le commitish au lieu du hash |

### MERGE CONFLICTS

|   |  |
|---|--|
| <code>git merge --abort</code>                  | Arrêter un merge lors d'un conflit         |
| <code>git checkout --ours &lt;PATH&gt;</code>   | Ecraser le fichier de la branche actuelle  |
| <code>git checkout --theirs &lt;PATH&gt;</code> | Ecraser le fichier de la branche en fusion |

### REBASE CONFLICTS

|   |   |
|---|---|
| <code>git rebase --abort</code>                 | Arrêter un rebase lors d'un conflit                   |
| <code>git rebase --continue</code>              | Permet de terminer un rebase lors d'un conflit        |
| <code>git checkout --ours &lt;PATH&gt;</code>   | Conserver les modifs de la branche en cours de rebase |
| <code>git checkout --theirs &lt;PATH&gt;</code> | Remplacer les modifs de la branche cible (main)       |
| <code>git reset --soft HEAD~1</code>            | Annuler un commit (alors qu'il fallait --continue)    |

### RERERE

|   |                             |
|---|-----------------------------|
| <code>git config rerere.enabled {TRUE/FALSE}</code> | Activer/désactiver RERERE   |
| <code>Rm -rf .git/rr-cache</code>                   | Supprimer le dossier RERERE |

### SQUASH

|   |                       |
|---|-----------------------|
| <code>git rebase -i HEAD~n</code>         | Démarrer le squashing |
| <code>git push origin main --force</code> | Forcer un push        |

### STASH

|  |   |
|--|---|
| <code>git stash [-m "MESSAGE"]</code>  | Créer un stash                                    |
| <code>git stash list</code>            | Lister les stashes                                |
| <code>git stash pop stash@{n}</code>   | Revenir au stash et supprimer le stash du stack   |
| <code>git stash apply stash@{n}</code> | Revenir au stash SANS supprimer le stash du stack |
| <code>git stash drop stash@{n}</code>  | Supprimer un stash sans y retourner               |

## DIFF

|   |  |
|---|--|
| <code>git diff</code>   | Compare l'état actuel et le dernier commit         |
| <code>git diff HEAD~{n}</code>                                    | Comparer l'état actuel et un certain commit        |
| <code>git diff &lt;COMMIT_HASH_1&gt; &lt;COMMIT_HASH_2&gt;</code> | Compare 2 commits                                  |
| <code>git diff --cached</code>                                    | Compare le staging avec la dernière validation     |
| <code>git diff HEAD</code>  | Compare l'arbre de travail avec la zone de staging |

## CHERRY PICK

|  |  |
|--|--|
| <code>git cherry-pick &lt;COMMIT_HASH&gt;</code> | Extraire un commit spécifique sans merge |
| <code>git cherry-pick --continue/--abort</code>  | Continuer/annuler à la fin d'un conflit  |

## BISECT

|  |                               |
|--|-------------------------------|
| <code>git bisect start</code>                    | Commencer le bisect           |
| <code>git bisect good &lt;COMMIT_HASH&gt;</code> | Marquer un commit fonctionnel |
| <code>git bisect bad &lt;COMMIT_HASH&gt;</code>  | Marquer un mauvais commit     |
| <code>git bisect good/bad</code>                 | Déclarer le résultat du test  |
| <code>git bisect reset</code>                    | Terminer le bisect            |

## WORKTREE

|   |   |
|---|---|
| <code>git worktree list</code>                              | Lister tous les worktrees d'un repo             |
| <code>git worktree add &lt;PATH&gt; [&lt;BRANCH&gt;]</code> | Créer un linked worktree                        |
| <code>git worktree remove &lt;WORKTREE&gt;</code>           | Supprimer un worktree                           |
| <code>git worktree prune</code>                             | Nettoyer les références des worktrees supprimés |

## TAGS

|  |  |
|--|--|
| <code>git tag</code>   | Lister les tags locaux                 |
| <code>git ls-remote --tags</code>                              | Lister les tags distants               |
| <code>git tag -a "TAG" &lt;COMMIT_HASH&gt; -m "MESSAGE"</code> | Créer un tag annoté                    |
| <code>git tag -d "TAG_NAME"</code>                             | Supprimer un tag localement            |
| <code>git push origin --delete &lt;TAG_NAME&gt;</code>         | Supprimer un tag distant               |
| <code>git push origin --tags</code>                            | Pousser les tags vers le dépôt distant |



---

## SOURCES

---



## **27 REFERENCES**

- Ben Straub, S. C. (2014). *Pro Git (2e édition)*. Apress.
- Git Documentation. (s.d.). *About Version Control*. Récupéré sur Git SCM: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
- Git Documentation. (s.d.). *Complete list of commands*. Récupéré sur Git SCM: [https://git-scm.com/docs/git#\\_git\\_commands](https://git-scm.com/docs/git#_git_commands)
- Git Documentation. (s.d.). *Git Documentation*. Récupéré sur Git SCM: <https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository>
- Git Documentation. (s.d.). *Git Tutorials and Training*. Récupéré sur Git SCM: <https://git-scm.com/docs/gittutorial>
- GitHub. (s.d.). *About GitHub and Git*. Récupéré sur GitHub: <https://docs.github.com/en/get-started/quickstart/about-github-and-git>
- GitHub. (s.d.). *Account and profile*. Récupéré sur GitHub Docs: <https://docs.github.com/en/account-and-profile>
- GitHub. (s.d.). *Authentification*. Récupéré sur GitHub Docs: <https://docs.github.com/en/authentication>
- GitHub. (s.d.). *Codespaces*. Récupéré sur GitHub Docs: <https://docs.github.com/en/codespaces>
- GitHub. (s.d.). *Get started*. Récupéré sur GitHub Docs: <https://docs.github.com/en/get-started>
- GitHub. (s.d.). *GitHub CLI*. Récupéré sur GitHub Docs: <https://docs.github.com/en/github-cli>
- GitHub. (s.d.). *Pull requests*. Récupéré sur GitHub Docs: <https://docs.github.com/en/pull-requests>
- GitHub. (s.d.). *Repositories*. Récupéré sur GitHub Docs: <https://docs.github.com/en/repositories>
- Ogdolo LLC, dba Boot.dev. (s.d.). *Learn Git*. Récupéré sur Boot.Dev: [www.boot.dev](http://www.boot.dev)
- Ogdolo LLC, dba Boot.dev. (s.d.). *Learn Git 2*. Récupéré sur Boot.Dev: [www.boot.dev](http://www.boot.dev)
- W3School. (s.d.). *Git Tutorial*. Récupéré sur W3School: <https://www.w3schools.com/git/>