

# Guide d'utilisation de git

## Sommaire

1. Présentation .....	2
2. Ressources .....	2
3. Gestion de versions (VCS) .....	3
3.1. Notion de version .....	3
3.2. VCS vs DVCS .....	4
3.3. Logiciels de gestion de versions .....	5
4. Fonctionnement interne .....	5
5. Les commandes .....	7
6. Premier pas .....	8
6.1. Installation .....	8
6.2. Configuration .....	9
7. Les commandes de base .....	10
7.1. Introduction .....	11
7.2. Initialiser un dépôt git .....	11
7.3. Travailler avec git .....	12
7.3.1. Ajouter un nouveau fichier .....	14
7.3.2. Modifier un fichier .....	15
7.3.3. Ignorer des fichiers .....	17
7.3.4. Visualiser des différences .....	19
7.3.5. Effacer des fichiers .....	24
7.3.6. Nettoyer son répertoire de travail .....	27
7.3.7. Déplacer/Renommer des fichiers .....	28
7.3.8. Visualiser l'historique .....	30
7.3.9. Annuler des actions .....	35
7.3.10. Étiqueter des versions .....	37
7.3.11. Publier une version .....	37
7.3.12. Utiliser le mode interactif .....	38
7.4. Conclusion .....	40
8. Les branches .....	41
8.1. Introduction .....	41
8.2. Travailler avec les branches .....	42
8.3. Retour sur le fonctionnement interne .....	49
8.4. Conclusion .....	51
9. Git hébergé .....	52
9.1. Notion de dépôt distant .....	53
9.2. Création d'un dépôt distant .....	54

9.3. Cloner un dépôt distant . . . . .	55
9.4. Exemple détaillé : développeur seul . . . . .	61
9.5. Branche de suivi . . . . .	69
9.6. Travailler dans GitHub . . . . .	84
9.7. Pull Request et Révision de code . . . . .	88
9.8. Travail collaboratif . . . . .	89
10. La fusion . . . . .	91
10.1. Stratégies de fusion . . . . .	91
10.2. Le conflit de fusion . . . . .	94
11. Workflow git et Gitflow . . . . .	105
12. Environnement de développement intégré (EDI ou IDE) . . . . .	108
12.1. Visual Studio Code . . . . .	108
13. Les outils graphiques . . . . .	111

Thierry Vaira - <[tvaira@free.fr](mailto:tvaira@free.fr)> - version v0.2 - 23/08/2021 - [tvaira.free.fr](http://tvaira.free.fr)

Objectif : Utiliser git en ligne de commande et avec GitHub.

# 1. Présentation

Git est un logiciel de **gestion de versions décentralisé** (DVCS). C'est un logiciel libre créé par **Linus Torvalds** en 2005. Il s'agit maintenant du logiciel de gestion de versions le plus populaire devant **Subversion (svn)** qu'il a remplacé avantageusement.



Site officiel : <https://git-scm.com/>

# 2. Ressources

- [Manuel de référence](#)
- [Livre Pro Git en français](#)
- [Livre Git Community Book en français](#)
- [Wikilivre Git en français](#)
- [Wikipedia Git](#)
- [Git Handbook sur Github](#)

# 3. Gestion de versions (VCS)

La [gestion de version](#) (*Version Control* ou *Revision Control*) consiste à gérer l'ensemble des versions d'un ou plusieurs fichiers (généralement en texte).



On préfère parfois le terme « révision » (une modification) afin de ne pas confondre la version d'un fichier et la version d'un logiciel, qui est une étape de distribution sous forme « finie » (*release*).

Un gestionnaire de version est donc un système (un outil logiciel) qui enregistre l'évolution d'un fichier (ou d'un ensemble de fichiers) au cours du temps dans un historique. Il permet de ramener un fichier à un état précédent, de ramener le projet complet à un état précédent, de visualiser les changements au cours du temps, de voir qui a modifié quelque chose et quand, et plus encore ...

Les fichiers ainsi versionnés sont mis à disposition sur un dépôt (*repository*). C'est un espace de stockage géré par un logiciel de gestion de versions.

Essentiellement utilisée dans le développement logiciel, elle concerne surtout la gestion des codes source.



[Le Code civil français sous Git !](#) Lire l'[interview de Stéeve Morin](#).

## 3.1. Notion de version

Les différentes versions (ou révision) sont nécessairement liées à travers des modifications : une modification est un ensemble d'ajouts, de modifications, et de suppressions de données.

La gestion de version repose sur deux mécanismes de base :

- un calcul de la différence entre deux versions ([diff](#) / [patch](#))

**diff**

Compare des fichiers ligne à ligne

**patch**

Utilise la différence entre deux fichiers pour passer d'une version à l'autre.

- un gestionnaire d'historique des [diff](#) pour conserver les modifications

Exemple :

*diff en action :*

```
$ cat toto-1.txt
toto :
Hello wordl!

$ cat toto-2.txt
toto :
Bonjour le monde !

$ diff toto-1.txt toto-2.txt
2c2
< Hello wordl!
---
> Bonjour le monde !
```

*patch en action :*

```
$ diff toto-1.txt toto-2.txt > toto.patch

$ patch toto-1.txt toto.patch
patching file toto-1.txt

$ cat toto-1.txt
toto :
Bonjour le monde !
```

La première ligne de la sortie de **diff** indique les numéros de ligne qui contiennent des différences et le type de modifications qui ont été apportées. Le **c** indique que le contenu a été remplacé, sinon **a** pour un ajout et **d** pour une suppression.



Les caractères **>** et **<** dans la sortie pointent dans la direction du fichier dans lequel se trouve le contenu. Ainsi, pour la commande ci-dessus, le **<** fait référence aux lignes de **toto-1.txt** et **>** fait référence aux lignes de **toto-2.txt**.

Le principe est donc le suivant : on passera de la version N à la version N+1 en appliquant une modification M. Un logiciel de gestion de versions applique ou retire ces modifications une par une pour fournir la version du fichier voulue.

## 3.2. VCS vs DVCS

Un système de gestion de version ou **VCS** (*Version Control System*) :

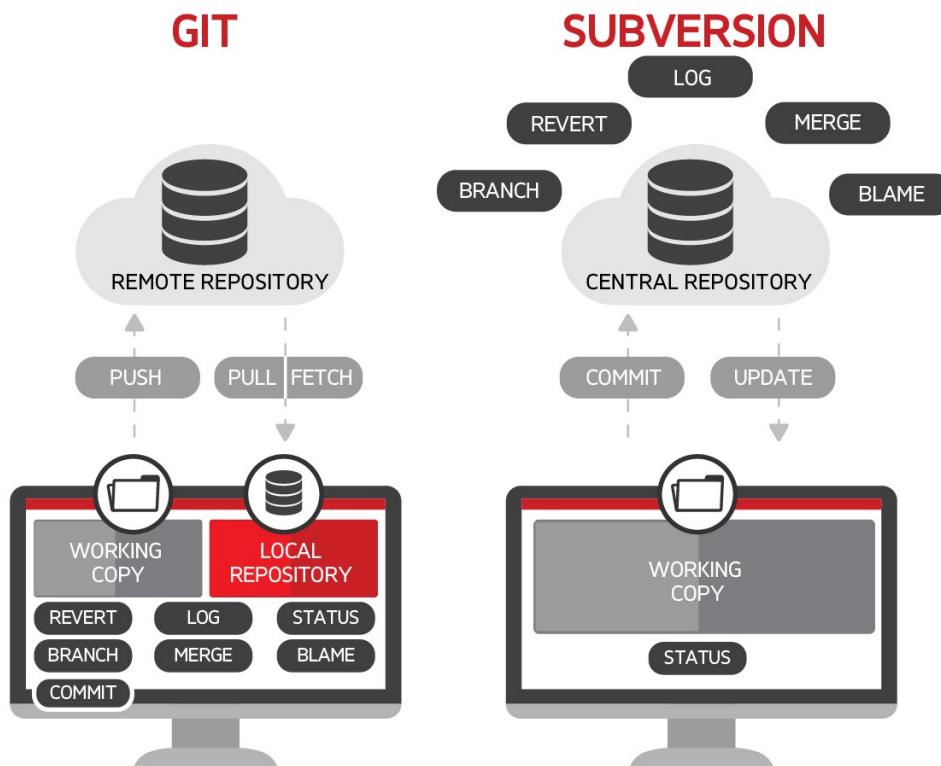
- maintient l'ensemble des versions d'un logiciel ;
- conserve l'historique (les révisions successives) du projet dans un seul dépôt (*repository*) qui fait référence : possibilités de revenir en arrière, de voir les changements ;
- facilite la collaboration entre les intervenants : chacun travaille avec son environnement,

plusieurs personnes travaillent sur les mêmes fichiers simultanément ;

- fournit des outils pour gérer le tout.

Un **DVCS** (*Distributed Version Control*) offre les mêmes services qu'un VCS sur une **architecture décentralisée** (ou distribuée).

<https://www.tabnine.com/blog/svn-vs-git/>



La plupart des opérations de Git sont locales.

Lien : [Systèmes centralisés et décentralisés](#)

### 3.3. Logiciels de gestion de versions

- Logiciels libres : [SCCS](#) → [GNU RCS](#) (standard de fait) → [CVS](#) → [Subversion \(svn\)](#) → [Git](#).  
Alternatives : [Bazaar](#) ou [Mercurial](#).
- Logiciels propriétaires : ClearCase (IBM©), Visual Source Safe et Team Foundation Server (Microsoft©), ...

## 4. Fonctionnement interne

Git a été conçu comme un système de fichiers versionnés.

Par bien des aspects, vous pouvez considérer Git comme un simple système de fichiers.

— Linus Torvalds (auteur du noyau Linux et de git)

Git possède deux structures de données : une base d'objets et un cache de répertoires.

Il existe quatre types d'objets :

- l'objet **blob** (*binary large object*), qui représente le contenu d'un fichier ;
- l'objet **tree** (arbre), qui décrit une arborescence de fichiers. Il est constitué d'une liste d'objets de type *blobs* et des informations qui leur sont associées, tel que le nom du fichier et les permissions. Il peut contenir récursivement d'autres *trees* pour représenter les sous-répertoires ;
- l'objet **commit** (résultat de l'opération du même nom signifiant « valider une transaction »), qui correspond à une arborescence de fichiers (*tree*) enrichie de métadonnées comme un message de description, le nom de l'auteur, etc. Il pointe également vers un ou plusieurs objets *commit* parents pour former un graphe d'historiques ;
- l'objet **tag** (étiquette) qui est une manière de nommer arbitrairement un *commit* spécifique pour l'identifier plus facilement. Il est en général utilisé pour marquer certains *commits*, par exemple par un numéro ou un nom de version.

La base des objets peut contenir n'importe quel type d'objets.

Une couche intermédiaire, utilisant des index (les sommes de contrôle), établit un lien entre les objets de la base et l'arborescence des fichiers.

Git indexe les fichiers d'après leur somme de contrôle calculée avec la [fonction de hachage SHA-1](#) qui génère un « *hash* » (une clé) de 160 bits.



Une empreinte SHA-1 est une chaîne de caractères composée de 40 caractères hexadécimaux (de '0' à '9' et de 'a' à 'f') calculée en fonction du contenu du fichier. Dans Git, c'est une signature unique qui sert de référence

*sha1sum en action :*

```
$ sha1sum toto-1.txt  
b6c3339dcaa25beabff0af919a49e8c44d800dab  toto-1.txt  
  
$ echo "Fin" >> toto-1.txt  
  
$ sha1sum toto-1.txt  
0610e586db143df27558d98a5bd4c2c792b0bf28  toto-1.txt
```



Dans Git, il est possible d'utiliser une empreinte SHA-1 courte (au moins 4 caractères) lorsqu'elle ne correspond pas à plusieurs *commits*. En règle générale, entre 8 et 10 caractères sont largement suffisants pour assurer l'unicité dans un projet. Par exemple, en février 2019, le noyau Linux avait de plus de 875 000 commits et presque sept millions d'objets dont les empreintes SHA sont uniques à partir des 12 premiers caractères.

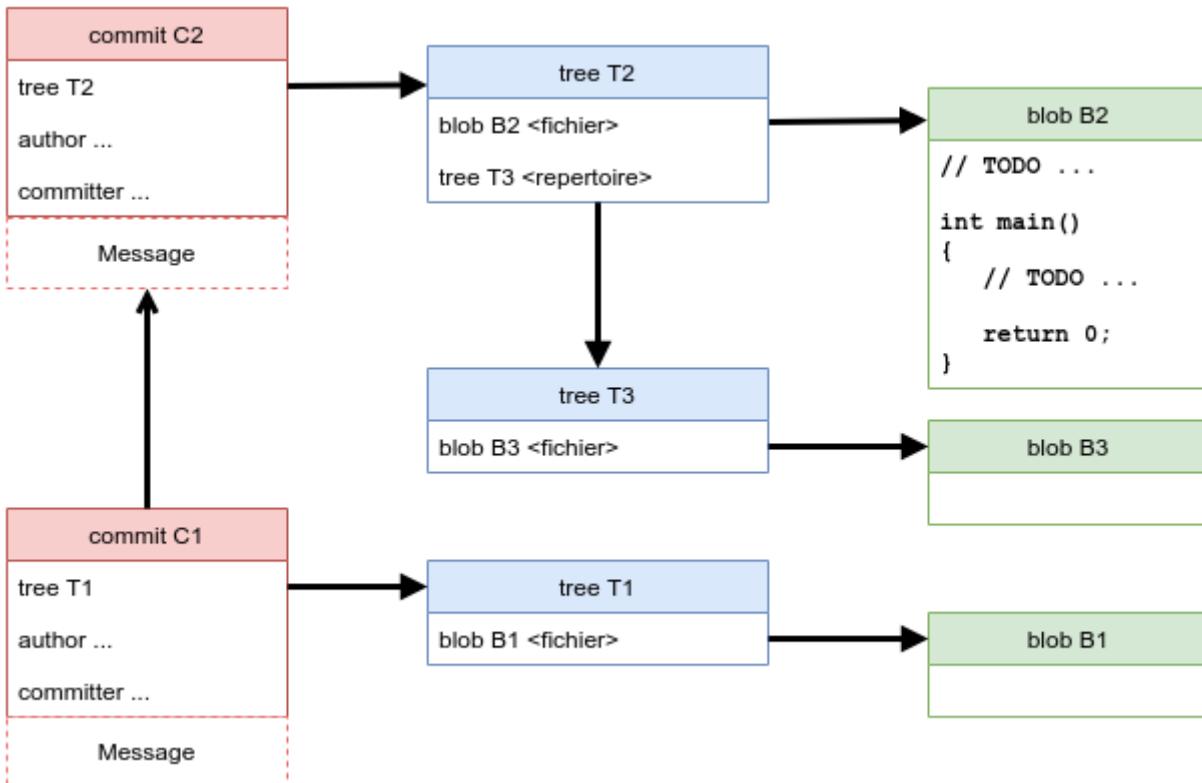
Git enregistre chaque révision dans un fichier en tant qu'objet *blob* unique.



En général, les objets *blobs* sont stockés dans leur intégralité en utilisant la compression de la **zlib**.

Une différence majeure entre Git et les autres VCS (comme Subversion) réside dans l'historique. La plupart des autres systèmes gèrent une liste de modifications de fichiers (des différences). Git ne fait pas ça : il stocke un instantané (un *commit*) de la représentation de tous les fichiers du projet dans une structure hiérarchisée. Pour être efficace, si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau mais seulement une référence vers celui-ci.

Exemple d'historique du « point de vue » de Git :



## 5. Les commandes

Git est un ensemble de commandes indépendantes dont les principales sont :

- **git init** crée un nouveau dépôt ;
- **git clone** clone un dépôt distant ;
- **git add** ajoute le contenu du répertoire de travail dans la zone d'index pour le prochain *commit* ;
- **git status** montre les différents états des fichiers du répertoire de travail et de l'index ;
- **git diff** montre les différences ;
- **git commit** enregistre dans la base de données (le dépôt) un nouvel instantané avec le contenu des fichiers qui ont été indexés puis fait pointer la branche courante dessus ;
- **git branch** liste les branches ou crée une nouvelle branche ;

- `git checkout` permet de basculer de branche et d'en extraire le contenu dans le répertoire de travail ;
- `git merge` fusionne une branche dans une autre ;
- `git log` affiche la liste des *commits* effectués sur une branche ;
- `git fetch` récupère toutes les informations du dépôt distant et les stocke dans le dépôt local ;
- `git push` publie les nouvelles révisions sur le dépôt distant ;
- `git pull` récupère les dernières modifications distantes du projet et les fusionne dans la branche courante ;
- `git tag` liste ou crée des *tags* ;
- `git stash` stocke de côté un état non commisé afin d'effectuer d'autres tâches.

Liens :

- [AIDE MÉMOIRE GITHUB GIT PDF](#)
- [Git CHEATSHEET](#)
- [Git Cheat Sheet](#)

*Obtenir de l'aide :*

```
$ git help
$ git --help
$ man git

$ git help <commande>
$ git <commande> --help
$ man git-<commande>
```

## 6. Premier pas

### Objectif

Installer et configurer Git Sous GNU/Linux Ubuntu

### 6.1. Installation

*Sous GNU/Linux Ubuntu :*

```
$ sudo apt-get install git gitk

$ git --version
git version 2.17.1
```



**gitk** est une interface graphique pour git. C'est un paquet optionnel !

Sous Mac OS X :

Il y a plusieurs façons d'installer **git** sous Mac OS X, en voici une : [git-osx-installer](#)

Sous Windows :

Le projet [Git for Windows](#) fournit une procédure d'installation : <https://github.com/git-for-windows/git/releases/latest>

## 6.2. Configuration

*Configuration du compte :*

```
$ git config --global user.name "<votre nom>"  
$ git config --global user.email "<votre email>"
```

*Choix de l'éditeur de texte :*

```
$ git config --global core.editor vim
```

*Activation de la coloration :*

```
$ git config --global color.diff auto  
$ git config --global color.status auto  
$ git config --global color.branch auto
```

etc ...



Le fichier de configuration **.gitconfig** est situé à la racine de votre répertoire personnel. Il peut exister un fichier **/etc/gitconfig** qui contient les valeurs pour tous les utilisateurs et tous les dépôts du système. Sinon la configuration sera complétée par le fichier **.git/config** du dépôt en cours d'utilisation. On peut alors utiliser la commande **git config --local**.

*Visualiser le fichier de configuration*

```
$ cat $HOME/.gitconfig
```

```
[color]
diff = auto
status = auto
branch = auto
[user]
name = tvaira
email = tvaira@free.fr
```

*Visualiser la configuration*

```
$ git config --list

$ git config user.name
tvaira
```

Stockage des identifiants :

Lien : <https://git-scm.com/book/fr/v2/Utilitaires-Git-Stockage-des-identifiants>

Le mode « cache » conserve en mémoire les identifiants pendant un certain temps. Aucun mot de passe n'est stocké sur le disque et les identifiants sont oubliés après 15 minutes par défaut.

```
$ git config --global credential.helper cache
```

L'assistant `cache` accepte une option `--timeout <secondes>` qui modifie la période de maintien en mémoire (par défaut, 900, soit 15 minutes).

*Exemple pour 8 heures :*

```
$ git config --global credential.helper 'cache --timeout 28800'
```

## 7. Les commandes de base

### Objectif

Découvrir les commandes de base nécessaires pour utiliser avec Git en local :

- configurer et initialiser un dépôt,
- commencer et arrêter le suivi de version de fichiers,
- indexer et valider des modifications.

On abordera aussi :

- le paramétrage de Git pour ignorer certains fichiers,

- revenir sur les erreurs rapidement et facilement,
- parcourir l'historique du projet et voir les modifications entre deux validations.

## 7.1. Introduction

La fonction principale de Git est de suivre les différentes versions d'un projet. Un projet est un ensemble de fichiers.

Le *commit* est l'élément central de Git. Un *commit* (ou instantané) représente un ensemble cohérent de modifications sur le projet.

## 7.2. Initialiser un dépôt git

*Création d'un répertoire :*

```
$ mkdir tp-git-sequence-1
mkdir: création du répertoire 'tp-git-sequence-1'

$ cd ./tp-git-sequence-1
```



Il est évidemment possible de commencer à partir d'un répertoire existant.

*Initialisation d'un dépôt git :*

```
$ git init
Dépôt Git vide initialisé dans $HOME/tp-git-sequence-1/.git/
```

Cela crée un nouveau sous-répertoire nommé **.git** qui contient tous les fichiers nécessaires au dépôt :

```
$ ls -al
...
drwxrwxr-x 7 tv tv 4096 juil. 28 10:58 .git

$ tree -L 1 .git
.git
├── config      # configuration des préférences
├── description # description du projet
├── HEAD        # pointeur vers la branche courante
├── hooks       # pre/post actions hooks
├── index       # l'index
├── logs        # historique
├── objects     # les objets (commits, trees, blobs, tags)
└── refs        # pointeurs vers les branches

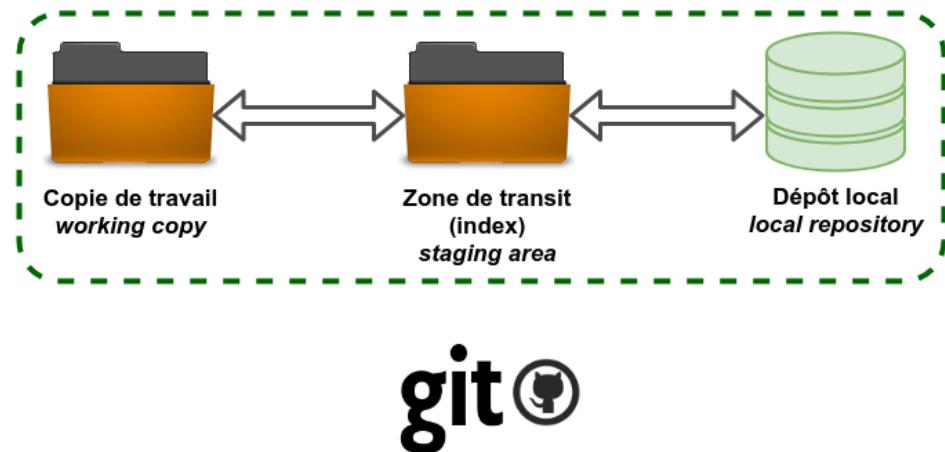
...
```



Pour l'instant, aucun fichier n'est encore versionné.

On distingue trois zones :

- le répertoire de travail (*working directory*) : répertoire (ici `tp-git-sequence-1`) du système de fichiers qui contient une extraction unique d'une version du projet pour pouvoir travailler
- l'index ou zone de transit (*staging area*) : un simple fichier (ici `.git/index`) qui stocke les informations concernant ce qui fera partie du prochain instantané (*commit*)
- le dépôt local (*local repository*) : répertoire (ici `.git`) qui stocke tout l'historique des instantanés (*commits*) et les méta-données du projet

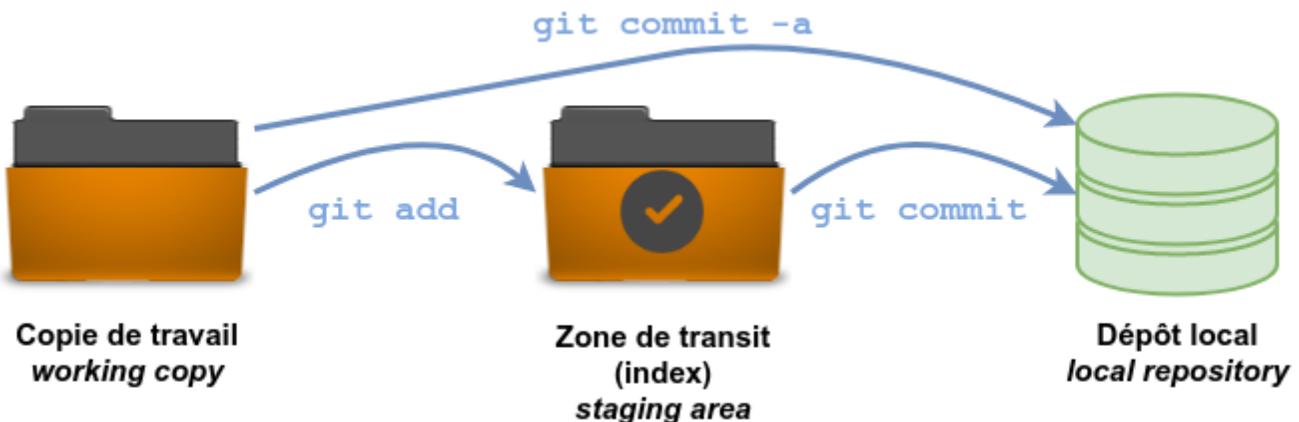


On peut considérer qu'il existe une quatrième zone nommée "remise" qui s'utilise avec la commande `git stash`.

## 7.3. Travailler avec git

L'utilisation standard de Git se passe comme suit :

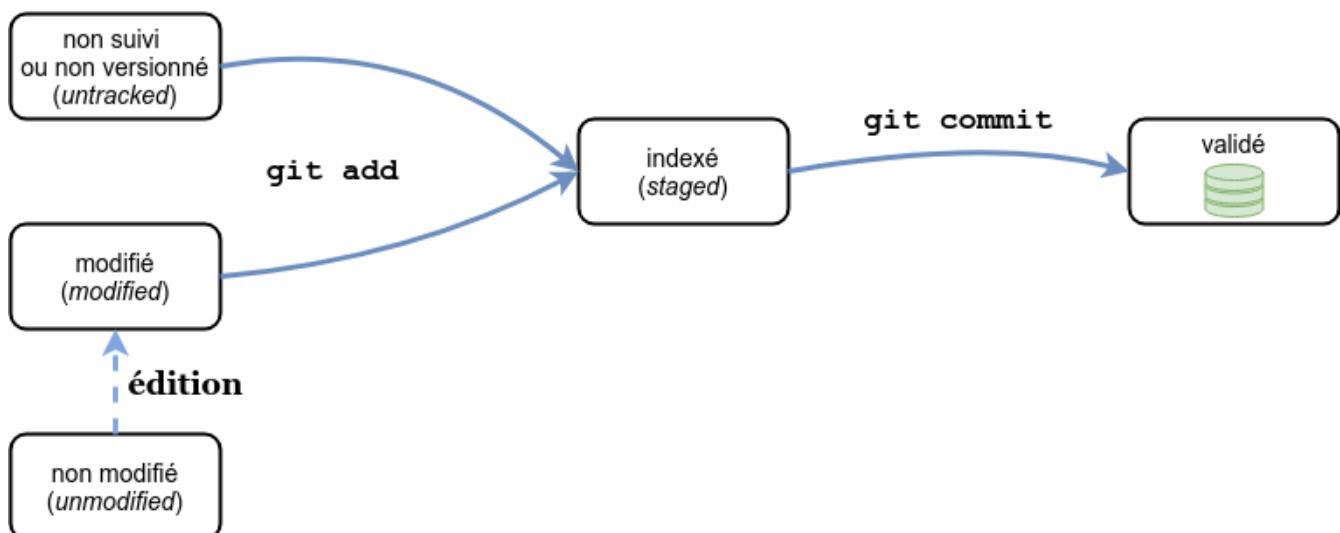
- on édite des fichiers dans le répertoire de travail (*working directory*) ;
- on indexe les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index (*staging area*) ;
- on valide les modifications, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans le dépôt local (*local repository*).



Lien : <https://ndpsoftware.com/git-cheatsheet.html>

Les différents états d'un fichier :

- non suivi ou non versionné (*untracked*) : aucun instantané existe pour ce fichier
- non modifié (*unmodified*) : non modifié depuis le dernier instantané
- modifié (*modified*) : modifié depuis le dernier instantané mais n'a pas été indexé
- indexé (*staged*) : modifié et ajouté dans la zone d'index
- validé : une version particulière d'un fichier



Pour obtenir l'état des fichiers du répertoire de travail (*working directory*), on utilise (très souvent) la commande **git status** :

*git status en action :*

```
$ git status --help
$ git help status

$ git status
Sur la branche master

Aucun commit

rien à valider (créez/copiez des fichiers et utilisez "git add" pour les suivre)

$ git status -s
$ git status -b
$ git status --long
$ git status -v
```



**master** (ou **main**) désigne la branche principale (cf. [Travailler avec les branches](#)).

### 7.3.1. Ajouter un nouveau fichier

*Création d'un fichier vide :*

```
$ touch bienvenue.cpp

$ git status -s
?? bienvenue.cpp

$ git status
Sur la branche master

Aucun commit

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

    bienvenue.cpp

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents
  (utilisez "git add" pour les suivre)
```

Le fichier **bienvenue.cpp** est non suivi (*untracked*).

**git add** est une commande multi-usage, elle peut être utilisée pour :

- pour placer un fichier sous suivi de version,
- pour indexer un fichier
- ou pour d'autres actions telles que marquer comme résolus des conflits de fusion de fichiers.

Sa signification s'approche plus de « ajouter ce contenu pour la prochaine validation » (*commit*).

*Ajout d'un fichier dans l'index :*

```
$ git add bienvenue.cpp

$ git status -s
A bienvenue.cpp

$ git status -v
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)

    nouveau fichier : bienvenue.cpp

diff --git a/bienvenue.cpp b/bienvenue.cpp
new file mode 100644
index 0000000..e69de29
```

Le fichier **bienvenue.cpp** est maintenant suivi et indexé (*staged*).

*Validation des changements dans le dépôt :*

```
$ git commit -m "Ajout du fichier bienvenue.cpp"
[master (commit racine) bb344f4] Ajout du fichier bienvenue.cpp
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 bienvenue.cpp
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

Le fichier **bienvenue.cpp** est validé dans le dépôt local.

### 7.3.2. Modifier un fichier

*Édition du fichier :*

```
$ vim bienvenue.cpp
```

```
// TODO Indiquer ce que fait le programme

int main()
{
    // TODO Afficher un message de bienvenue

    return 0;
}
```

```
$ git status -s
M bienvenue.cpp

$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la
copie de travail)

  modifié :      bienvenue.cpp

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit
-a")
```

Avant d'indexer le fichier modifié, il est plus prudent de vérifier son utilisation :

*Fabrication de l'exécutable :*

```
$ g++ -c bienvenue.cpp
$ ls -l
-rw-rw-r-- 1 tv tv 119 juil. 28 20:46 bienvenue.cpp
-rw-rw-r-- 1 tv tv 1232 juil. 28 20:48 bienvenue.o

$ g++ -o bienvenue bienvenue.o
$ ls -l
-rwxrwxr-x 1 tv tv 8168 juil. 28 20:48 bienvenue
-rw-rw-r-- 1 tv tv 119 juil. 28 20:46 bienvenue.cpp
-rw-rw-r-- 1 tv tv 1232 juil. 28 20:48 bienvenue.o

$ ./bienvenue
```

On peut maintenant indexer le fichier :

```
$ git add bienvenue.cpp
```

```

$ git status -s
M bienvenue.cpp
?? bienvenue
?? bienvenue.o

$ git status -v
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    modifié :      bienvenue.cpp

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

    bienvenue
    bienvenue.o

diff --git a/bienvenue.cpp b/bienvenue.cpp
index e69de29..d315a70 100644
--- a/bienvenue.cpp
+++ b/bienvenue.cpp
@@ -0,0 +1,9 @@
+// TODO Indiquer ce que fait le programme
+
+int main()
+{
+  // TODO Afficher un message de bienvenue
+
+  return 0;
+}
+

```



**HEAD** désigne le commit le plus récent de la branche courante.

Puis valider les modifications :

```

$ git commit -m "Création du programme principal"
[master 973e4f7] Création du programme principal
 1 file changed, 9 insertions(+)

```

### 7.3.3. Ignorer des fichiers

Il apparaît souvent que certains types de fichiers présents dans la copie de travail ne doivent pas être ajoutés au dépôt :

```
$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
```

```
  bienvenue
  bienvenue.o
```

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents  
(utilisez "git add" pour les suivre)

Ici, ce sont les fichiers issus de la fabrication (exécutable, fichiers objets, ...).

Pour simplement les ignorer dans git, il faut les ajouter dans un fichier spécial `.gitignore` :

*Création d'un fichier `.gitignore` :*

```
$ touch .gitignore
$ echo '*.[oa]' >> .gitignore
$ echo '*~' >> .gitignore
```

```
$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
```

```
  .gitignore
  bienvenue
```

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents  
(utilisez "git add" pour les suivre)

```
$ echo 'bienvenue' >> .gitignore
```

```
$ git status
```

Sur la branche master

Fichiers non suivis:

(utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

```
  .gitignore
```

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents  
(utilisez "git add" pour les suivre)

On peut aussi l'ajouter au dépôt :

```
$ git add .gitignore
$ git commit -m "Ajout du fichier .gitignore"
[master af1dcc8] Ajout du fichier .gitignore
 1 file changed, 3 insertions(+)
 create mode 100644 .gitignore
```

Vérification :

```
$ git status
Sur la branche master
rien à valider, la copie de travail est propre

$ git status --ignored
Sur la branche master
Fichiers ignorés:
  (utilisez "git add -f <fichier>..." pour inclure dans ce qui sera validé)

    bienvenue
    bienvenue.o

rien à valider, la copie de travail est propre

$ ls bienvenue*
bienvenue  bienvenue.cpp  bienvenue.o

$ git check-ignore bienvenue*
bienvenue
bienvenue.o
```

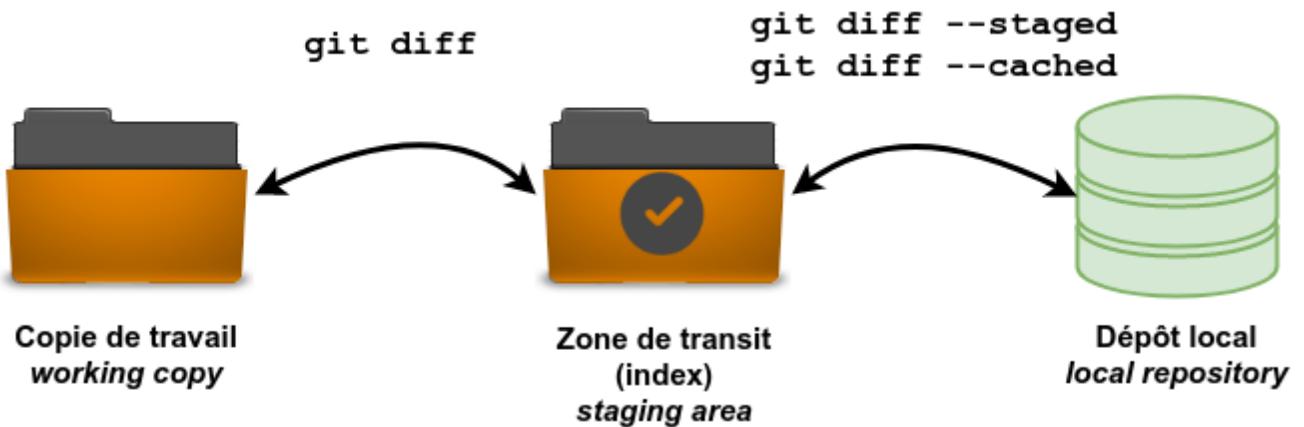
Liens :

- [.gitignore](#)
- [Une collection de modèles .gitignore](#)

#### 7.3.4. Visualiser des différences

En complément de `git status`, on utilisera la commande `git diff` pour visualiser les lignes exactes qui ont été ajoutées, modifiées ou effacées :

- qu'est-ce qui a été modifié mais pas encore indexé ?
- quelle modification a été indexée et qui est prête pour la validation ?



On modifie le programme principal :

```
$ vim bienvenue.cpp
```

```
// Affiche un message de bienvenue

#include <iostream>

int main()
{
    std::cout << "Bienvenue le monde !" << std::endl;

    return 0;
}
```

```
$ g++ -c bienvenue.cpp
$ g++ bienvenue.o -o bienvenue
$ ./bienvenue
Bienvenue le monde !
```

Le fichier est dans l'état modifié dans le répertoire de travail :

```
$ git status -s
 M bienvenue.cpp
```

La commande **git diff** compare le contenu du répertoire de travail avec la zone d'index. Cela affiche les modifications réalisées mais non indexées :

Voir les différences avec l'index :

```
$ git diff
diff --git a/bienvenue.cpp b/bienvenue.cpp
index d315a70..e8d46fe 100644
--- a/bienvenue.cpp
+++ b/bienvenue.cpp
@@ -1,8 +1,10 @@
-// TODO Indiquer ce que fait le programme
+// Affiche un message de bienvenue
+
+">#include <iostream>

int main()
{
- // TODO Afficher un message de bienvenue
+ std::cout << "Bienvenue le monde !" << std::endl;

    return 0;
}
```

On indexe le fichier :

```
$ git add bienvenue.cpp
```

Et :

```
$ git diff
Aucune différence
```

La commande `git diff --staged` compare les fichiers indexés et le dernier instantané (*commit*). Cela affiche les modifications indexées qui feront partie de la prochaine validation :

Contenu de l'index :

```
$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

  modifié :      bienvenue.cpp
```

Voir les différences avec le dernier commit :

```
$ git diff --staged
diff --git a/bienvenue.cpp b/bienvenue.cpp
index d315a70..e8d46fe 100644
--- a/bienvenue.cpp
+++ b/bienvenue.cpp
@@ -1,8 +1,10 @@
-// TODO Indiquer ce que fait le programme
+// Affiche un message de bienvenue
+
+">#include <iostream>

int main()
{
- // TODO Afficher un message de bienvenue
+ std::cout << "Bienvenue le monde !" << std::endl;

    return 0;
}
```

On valide :

```
$ git commit -m "Affiche un message de bienvenue"
[master 4717082] Affiche un message de bienvenue
 1 file changed, 4 insertions(+), 2 deletions(-)
```

Et :

```
$ git diff --staged
Aucune différence

$ git status
Sur la branche master
rien à valider, la copie de travail est propre

$ cat bienvenue.cpp
```

```
// Affiche un message de bienvenue

#include <iostream>

int main()
{
    std::cout << "Bienvenue le monde !" << std::endl;

    return 0;
}
```

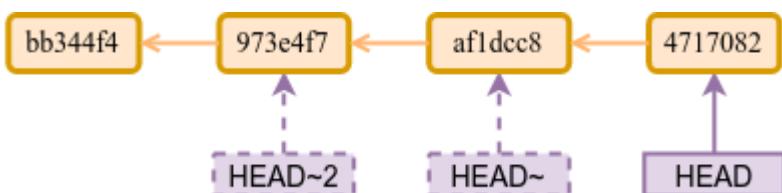


--staged est un synonyme de --cached.

La commande `git diff <commit>` sert à visualiser les modifications présentes dans le répertoire de travail par rapport au `<commit>` indiqué. On peut aussi utiliser la référence `HEAD` pour le comparer au commit le plus récent.



`HEAD` est une référence symbolique pointant vers l'endroit où l'on se trouve dans l'historique. Si on fait un *commit*, `HEAD` se déplacera. `HEAD^` signifie le premier parent immédiat de la pointe de la branche actuelle. `HEAD^1` est l'abréviation de `HEAD^1`. Pour un *commit* avec un seul parent, `HEAD~` et `HEAD^` signifient la même chose. cf. [Exemple de déplacement avec HEAD](#)



Les développeurs utilisent aussi des outils graphiques ou externes pour visualiser les différences. Dans ce cas, il faut utiliser `git difftool` au lieu de `git diff`.

Pour connaître les applications disponibles :

```
$ git difftool --tool-help
'git difftool --tool=<tool>' may be set to one of the following:
    araxis
    kompare
    meld
    vimdiff
    vimdiff2
    vimdiff3
```

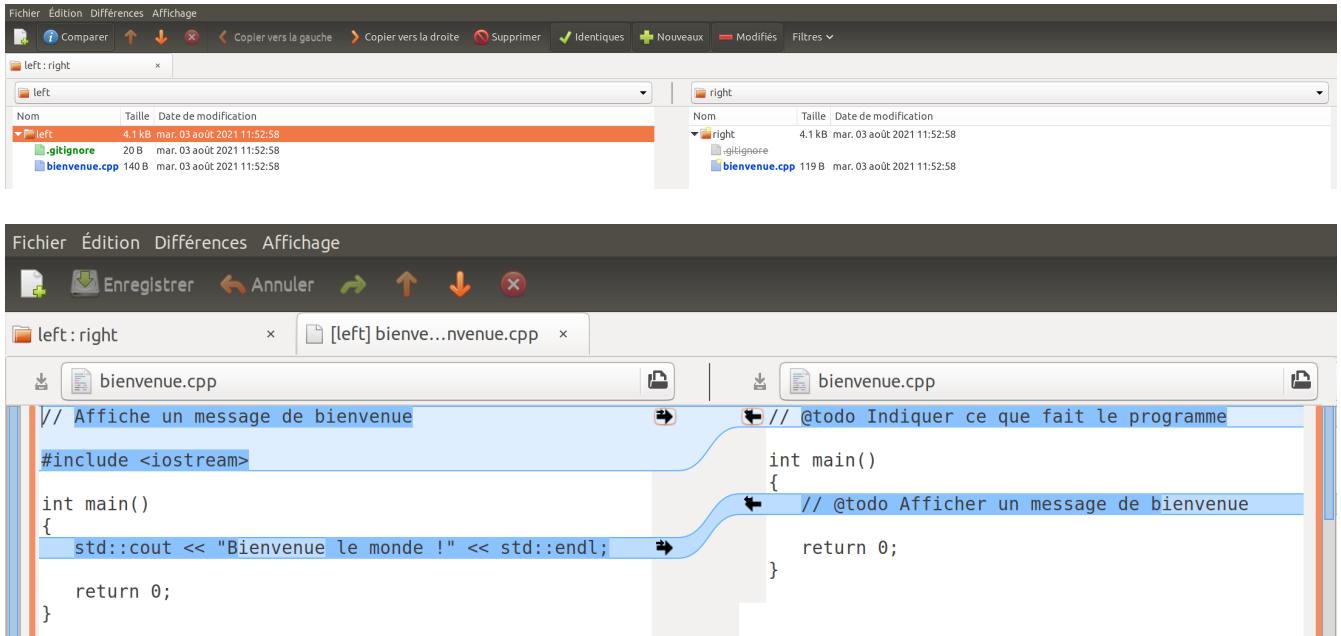
...

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

Exemples avec **Meld** :

```
$ git difftool -t meld --dir-diff  
$ git difftool -t meld --dir-diff 4717082 973e4f7
```

*Meld en action :*



La commande **git blame** annote les lignes de n'importe quel fichier avec des informations : le *commit* du dernier changement avec son auteur et l'horodatage.

```
$ git blame bienvenue.cpp
47170829 (tvaira 2021-07-28 21:30:16 +0200 1) // Affiche un message de bienvenue
47170829 (tvaira 2021-07-28 21:30:16 +0200 2)
47170829 (tvaira 2021-07-28 21:30:16 +0200 3) #include <iostream>
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 4)
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 5) int main()
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 6) {
47170829 (tvaira 2021-07-28 21:30:16 +0200 7)     std::cout << "Bienvenue le monde !"
<< std::endl;
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 8)
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 9)     return 0;
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 10) }
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 11)
```

### 7.3.5. Effacer des fichiers

Pour effacer un fichier de Git, il faut l'effacer dans la zone d'index puis valider. La commande **git rm** réalise cette action mais efface aussi ce fichier de la copie de travail.

Pour conserver le fichier dans la copie de travail, il faut utiliser l'option **--cached**.

Il existe une mesure de sécurité pour empêcher un effacement accidentel lorsqu'un fichier a été

modifié et indexé. Il est alors possible de forcer son élimination avec l'option **-f**.

*Ajout d'un fichier :*

```
$ touch README

$ ls -l README
-rw-rw-r-- 1 tv tv 0 juil. 31 11:49 README

$ git add README

$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

  nouveau fichier : README
```

*Suppression forcée d'un fichier :*

```
$ git rm README
error: le fichier suivant a des changements indexés :
  README
(utilisez --cached pour garder le fichier, ou -f pour forcer la suppression)

$ git rm README -f
rm 'README'

$ git status
Sur la branche master
rien à valider, la copie de travail est propre

$ ls -l README
ls: impossible d'accéder à 'README': Aucun fichier ou dossier de ce type
```

*Ajout d'un fichier :*

```
$ touch README

$ ls -l README
-rw-rw-r-- 1 tv tv 0 juil. 31 11:52 README

$ git add README

$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    nouveau fichier : README
```

*Suppression d'un fichier de l'index :*

```
$ git rm README --cached
rm 'README'

$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

    README

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents
(utilisez "git add" pour les suivre)

$ ls -l README
-rw-rw-r-- 1 tv tv 0 juil. 31 11:52 README
```

*Ajout d'un fichier au dépôt :*

```
$ git add README

$ git commit -m "Ajout README"
[master e60cc7e] Ajout README
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README
```

*Suppression d'un fichier du dépôt :*

```
$ git rm README
rm 'README'

$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    supprimé :      README

$ git commit -m "Suppression README"
[master 357d005] Suppression README
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 README

$ git status
Sur la branche master
rien à valider, la copie de travail est propre

$ ls -l README
ls: impossible d'accéder à 'README': Aucun fichier ou dossier de ce type
```

### 7.3.6. Nettoyer son répertoire de travail

Par défaut, la commande `git clean` ne va supprimer que les fichiers non-suivis qui ne sont pas ignorés (cf. `.gitignore`).

Les options intéressantes sont :

- `-n` : ne supprime rien mais montre simplement ce qui serait fait.
- `-f` : pour forcer la suppression
- `-x` : supprime aussi les fichiers ignorés (`-X` supprime seulement les fichiers ignorés)
- `-i` ou `--interactive` : utilise le mode interactif pour choisir ce qui sera fait

*git clean en action :*

```
$ touch hello  
  
$ git clean -n  
Supprimerait hello  
  
$ git clean  
fatal: clean.requireForce à true par défaut et ni -i, -n ou -f fourni ; refus de  
nettoyer  
  
$ git clean -f  
Suppression de hello
```



Il est (souvent) impossible de récupérer le contenu des fichiers après un `git clean`. Une option plus sécurisée consisterait à "remiser" l'ensemble avec `git stash --all`. Lien : <https://git-scm.com/docs/git-stash/fr>

### 7.3.7. Déplacer/Renommer des fichiers

La commande `git mv` permet de renommer un fichier. Cela évite de faire successivement les commandes `mv`, `git rm` et `git add`.

*Ajout d'un fichier au dépôt :*

```
$ touch README  
$ git add README  
$ git commit -m "Ajout README"  
[master f937b30] Ajout README  
 1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 README
```

*Renommage d'un fichier du dépôt :*

```
$ git mv README README.md

$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    renommé : README -> README.md

$ git commit -m "Renommage README.md"
[master 948859b] Renommage README.md
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README => README.md (100%)

$ ls -l README*
-rw-rw-r-- 1 tv tv 0 juil. 31 11:59 README.md
```

```
$ vim README.md
```

```
# Bienvenue
```

```
Programme C++ qui affiche "Bienvenue"
```

*Ajout d'un fichier modifié dans l'index :*

```
$ git add README.md

$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    modifié : README.md
```

*Renommage d'un fichier dans l'index :*

```
$ git mv README.md README  
  
$ git status  
Sur la branche master  
Modifications qui seront validées :  
(utilisez "git reset HEAD <fichier>..." pour désindexer)  
  
    nouveau fichier : README  
    supprimé :       README.md  
  
$ git commit -m "Renommage README"  
[master bb6ef9f] Renommage README  
 2 files changed, 4 insertions(+)  
  create mode 100644 README  
  delete mode 100644 README.md  
  
$ ls -l README*  
-rw-rw-r-- 1 tv tv 52 juil. 31 12:02 README  
  
$ cat README
```

```
# Bienvenue  
  
Programme C++ qui affiche "Bienvenue"
```

### 7.3.8. Visualiser l'historique

Après avoir créé plusieurs instanés (*commits*), il est possible de consulter l'historique avec la commande `git log`. C'est une commande importante et puissante disposant de nombreuses options.

Par défaut, `git log` affiche les *commits* réalisés en ordre chronologique inversé. Cela signifie que les *commits* les plus récents apparaissent en premier. Sinon, on utilisera l'option `--reverse`.

Les options les plus utilisés sont :

- `git log -<nombree>` Limiter le nombre de *commits*
- `git log --oneline` Affiche chaque *commit* sur une seule ligne
- `git log -p` Affiche la différence complète de chaque *commit*
- `git log --graph --decorate` Affiche l'historique sous forme de graphe
- `git log --stat` Affiche l'historique avec des statistiques
- `git log -- <fichier>` Affiche uniquement les *commits* contenant le fichier spécifié
- `git blame <fichier>` Affiche qui a modifié le fichier et quand
- `git log <depuis>..<jusqu'à>` Affiche les validations qui se produisent entre deux commits en utilisant une référence comme un ID de validation, un nom de branche, `HEAD` ou tout autre type

de référence de révision.



Il est possible d'appliquer des critères de recherche avec les options `--author`, `--grep` et `-S`. Voir aussi : `--since`, `--after`, `--until` et `--before`.

*Historique complet :*

```
$ git log  
commit bb6ef9fbb54b4aa856bbd6effbc30601d38acffb (HEAD -> master)  
Author: tvaira <tvaira@free.fr>  
Date:   Sat Jul 31 12:05:07 2021 +0200
```

Renommage README

```
commit 948859bdcac73aff903fa13fe340658daefc4922  
Author: tvaira <tvaira@free.fr>  
Date:   Sat Jul 31 12:00:32 2021 +0200
```

Renommage README.md

```
commit f937b306dfb405a77a26688bf8aecf0312d33799  
Author: tvaira <tvaira@free.fr>  
Date:   Sat Jul 31 11:59:57 2021 +0200
```

Ajout README

```
commit 357d00546a9968556fafd680b1721b37d58bb70f  
Author: tvaira <tvaira@free.fr>  
Date:   Sat Jul 31 11:56:26 2021 +0200
```

Suppression README

```
commit e60cc7eae4f55b7cb4c53c20827f904697308898  
Author: tvaira <tvaira@free.fr>  
Date:   Sat Jul 31 11:55:48 2021 +0200
```

Ajout README

```
commit 47170829ef8654ec28f6d3b74d00b2a0baeaefa9  
Author: tvaira <tvaira@free.fr>  
Date:   Wed Jul 28 21:30:16 2021 +0200
```

Affiche un message de bienvenue

```
commit af1dcc83807624005b76a1ca5d7e790ce6f1737a  
Author: tvaira <tvaira@free.fr>  
Date:   Wed Jul 28 21:03:28 2021 +0200
```

Ajout du fichier .gitignore

```
commit 973e4f7d830313e4ac9b08a332db767cdf28941f
```

```
Author: tvaira <tvaira@free.fr>
Date:   Wed Jul 28 20:55:12 2021 +0200
```

Création du programme principal

```
commit bb344f417dbbf7f6725b24b293af2909bad6a519
```

```
Author: tvaira <tvaira@free.fr>
Date:   Wed Jul 28 20:33:06 2021 +0200
```

Ajout du fichier bienvenue.cpp

*Les plus anciens en premier :*

```
$ git log --reverse
commit bb344f417dbbf7f6725b24b293af2909bad6a519
Author: tvaira <tvaira@free.fr>
Date:   Wed Jul 28 20:33:06 2021 +0200
```

Ajout du fichier bienvenue.cpp

```
commit 973e4f7d830313e4ac9b08a332db767cdf28941f
Author: tvaira <tvaira@free.fr>
Date:   Wed Jul 28 20:55:12 2021 +0200
```

Création du programme principal

...

Avec les différences :

```
$ git log -p
...
commit 47170829ef8654ec28f6d3b74d00b2a0baeaefa9 (HEAD -> master)
Author: tvaira <tvaira@free.fr>
Date:   Wed Jul 28 21:30:16 2021 +0200

    Affiche un message de bienvenue

diff --git a/bienvenue.cpp b/bienvenue.cpp
index d315a70..e8d46fe 100644
--- a/bienvenue.cpp
+++ b/bienvenue.cpp
@@ -1,8 +1,10 @@
-// TODO Indiquer ce que fait le programme
+// Affiche un message de bienvenue
+
+#include <iostream>

int main()
{
-    // TODO Afficher un message de bienvenue
+    std::cout << "Bienvenue le monde !" << std::endl;

    return 0;
}

...
```

Affiche chaque commit sur une seule ligne :

```
$ git log --oneline
bb6ef9f (HEAD -> master) Renommage README
948859b Renommage README.md
f937b30 Ajout README
357d005 Suppression README
e60cc7e Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp
```

*Affichage sous forme de graphe :*

### *Affichage personnalisé :*

```
$ git log --pretty=format:"%h - %an, %ar : %s"  
...
```

La commande `git blame` annote les lignes de n'importe quel fichier avec des informations : le commit du dernier changement avec son auteur et l'horodatage :

```
$ git blame bienvenue.cpp
47170829 (tvaira 2021-07-28 21:30:16 +0200 1) // Affiche un message de bienvenue
47170829 (tvaira 2021-07-28 21:30:16 +0200 2)
47170829 (tvaira 2021-07-28 21:30:16 +0200 3) #include <iostream>
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 4)
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 5) int main()
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 6) {
47170829 (tvaira 2021-07-28 21:30:16 +0200 7)     std::cout << "Bienvenue le monde !"
<< std::endl;
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 8)
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 9)     return 0;
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 10) }
973e4f7d (tvaira 2021-07-28 20:55:12 +0200 11)
```

### 7.3.9. Annuler des actions

Il est possible de modifier le dernier *commit* (plutôt de le remplacer complètement par un nouveau *commit*) avec la commande `git commit --amend`.

```
$ vim README
```

```
# Bienvenue
```

```
Programme C++ qui affiche "Bienvenue le monde!"
```

```
$ git add README
```

```
$ git commit -m "Modification README.md"
[master 25dbef1] Modification README.md
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Modification du dernier commit :

```
$ git commit --amend
Modification README

# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Date :      Tue Aug 10 11:41:26 2021 +0200
#
# Sur la branche master
# Modifications qui seront validées :
#       modifié :      README
#
[master e29d1f8] Modification README
Date: Tue Aug 10 11:41:26 2021 +0200
1 file changed, 1 insertion(+), 1 deletion(-)
```

Vérification :

```
$ git log --oneline
e29d1f8 (HEAD -> master) Modification README
bb6ef9f Renommage README
948859b Renommage README.md
f937b30 Ajout README
357d005 Suppression README
e60cc7e Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp
```

On peut aussi annuler des modifications dans la zone d'index et la zone de travail :

- `git reset HEAD <fichier>` pour désindexer un fichier
- `git checkout -- <fichier>` pour annuler les modifications dans la copie de travail



`git reset` peut être une commande dangereuse, notamment avec l'option `--hard`. De manière générale, il est déconseillé de modifier l'historique dans le cas d'un travail collaboratif. La version 2.25.0 de Git a introduit une nouvelle commande : `git restore`. C'est fondamentalement une alternative à `git reset`.



Pour annuler un commit, on peut l'inverser (*revert*) : `git revert` crée un *commit* qui applique l'exact opposé des modifications introduites par le *commit* ciblé.

### 7.3.10. Étiqueter des versions

Git donne la possibilité d'étiqueter un certain état dans l'historique. On l'utilise pour marquer (*tag*) les états de publication comme des versions (1.0 par exemple).



Git utilise deux types principaux d'étiquettes : légères et annotées (avec l'option `-a`). Une étiquette légère est considérée comme un pointeur sur un commit spécifique. Par contre, les étiquettes annotées sont stockées en tant qu'objets à part entière dans la base de données de Git.

Étiqueter une version :

```
$ git tag -a 1.0 -m 'La version 1.0'  
  
$ git tag  
1.0  
  
$ git show 1.0  
tag 1.0  
Tagger: tvaira <tvaira@free.fr>  
Date:   Wed Aug 11 15:40:13 2021 +0200  
  
La version 1.0  
...
```



Il est possible d'étiqueter après coup. Pour cela, il faut spécifier le *commit* en fin de commande : `git tag -a v1.2 <commit>`

### 7.3.11. Publier une version

Pour publier une version, il est nécessaire de créer une archive à partir d'un instantané (généralement une étiquette de version).

La commande dédiée à cette action est `git archive` :

```
# Exemple :  
# git archive --prefix=src-directory-name tag --format=zip > `git describe master`.zip  
  
$ git archive --prefix='tp-git-sequence-1-vaira/' 1.0 | gzip > tp-git-sequence-1-  
vaira.tar.gz  
$ git archive --prefix='tp-git-sequence-1-vaira/' 1.0 --format=zip > tp-git-sequence-  
1-vaira.zip
```



N'oubliez pas d'ajouter l'option `--prefix` (et de préciser votre nom comme identifiant) avant de rendre une archive de TP !!!



Il est possible de recopier le dépôt avec la commande : `cp -Rf tp-git-sequence-1 <destination>`. Git fournit aussi la commande `git clone`. Mais en pratique, on utilisera plutôt des dépôts hébergés ([GitHub](#), [GitLab](#), [Bitbucket](#), ...).

### 7.3.12. Utiliser le mode interactif

Git propose quelques scripts qui "guident" les opérations en ligne de commande avec l'option `-i` ou `--interactive`.

Le mode interactif s'utilise principalement avec les commandes :

- `git add --interactive` : pour choisir les fichiers ou les parties d'un fichier à incorporer à un *commit*
- `git clean --interactive` : pour choisir les fichiers qui seront supprimés du répertoire de travail
- `git rebase --interactive` : pour choisir les *commits* à "rejouer"

Git ne possède pas d'outil de modification d'historique mais, il est possible d'utiliser l'outil `rebase` en mode interactif pour :

- Réordonner les *commits*
- Écraser un *commit*
- Diviser un *commit*
- Supprimer un *commit*

Il est également possible de prendre une série de *commits* et de les rassembler en un seul avec l'outil de rebasage interactif :

```
$ git log --oneline
e29d1f8 (HEAD -> master) Modification README
bb6ef9f Renommage README
948859b Renommage README.md
f937b30 Ajout README
357d005 Suppression README
e60cc7e Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp
```



Rappel : `HEAD` est une référence symbolique pointant vers l'endroit où l'on se trouve dans l'historique. Si on fait un *commit*, `HEAD` se déplacera. `HEAD^` signifie le premier parent immédiat de la pointe de la branche actuelle. `HEAD^` est l'abréviation de `HEAD^1`. Pour un *commit* avec un seul parent, `HEAD~` et `HEAD^` signifient la même chose. cf. [Exemple de déplacement avec HEAD](#)

Rebasage des 6 derniers commits :

```

$ git rebase -i HEAD~6
pick e60cc7e Ajout README
squash 357d005 Suppression README
squash f937b30 Ajout README
squash 948859b Renommage README.md
squash bb6ef9f Renommage README
squash e29d1f8 Modification README

# Rebasage de 4717082..e29d1f8 sur 4717082 (6 commandes)
#
# Commandes :
# p, pick = utiliser le commit
# r, reword = utiliser le commit, mais reformuler son message
# e, edit = utiliser le commit, mais s'arrêter pour le modifier
# s, squash = utiliser le commit, mais le fusionner avec le précédent
# f, fixup = comme "squash", mais en éliminant son message
# x, exec = lancer la commande (reste de la ligne) dans un shell
# d, drop = supprimer le commit
#
# Vous pouvez réordonner ces lignes ; elles sont exécutées de haut en bas.
#
# Si vous éliminez une ligne ici, LE COMMIT CORRESPONDANT SERA PERDU.
#
# Cependant, si vous effacez tout, le rebasage sera annulé.
#
# Veuillez noter que les commits vides sont en commentaire

# Ceci est la combinaison de 6 commits.
# Ceci est le premier message de validation :

Ajout README

# Ceci est le message de validation numéro 2 :

#Suppression README

# Ceci est le message de validation numéro 3 :

#Ajout README

# Ceci est le message de validation numéro 4 :

#Renommage README.md

# Ceci est le message de validation numéro 5 :

#Renommage README

# Ceci est le message de validation numéro 6 :

#Modification README

```

```

# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Date :      Sat Jul 31 11:55:48 2021 +0200
#
# rebasage interactif en cours ; sur 4717082
# Dernières commandes effectuées (6 commandes effectuées) :
#   squash bb6ef9f Renommage README
#   squash e29d1f8 Modification README
# Aucune commande restante.
# Vous êtes en train de rebaser la branche 'master' sur '4717082'.
#
# Modifications qui seront validées :
#   nouveau fichier : README
#

```

```

[HEAD détachée 7cbe84f] Ajout README
Date: Sat Jul 31 11:55:48 2021 +0200
1 file changed, 4 insertions(+)
create mode 100644 README
Successfully rebased and updated refs/heads/master.

```

Vérification :

```

$ git log --oneline
7cbe84f (HEAD -> master) Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp

```



De manière générale, il est déconseillé de modifier l'historique centralisé dans le cas d'un travail collaboratif. Voir : [Nettoyer son historique local avant de publier](#).

## 7.4. Conclusion

### Cycle de travail

- Éditer des fichiers (`vim` ou un `EDI`)
- Ajouter les changement (`git add <fichier>`)
- Valider les changements (`git commit -m "Message"`)

Les commandes que l'on utilise tout le temps :

- `git status`
- `git log` ...

## 8. Les branches

### Objectif

Découvrir l'utilisation des branches.

### 8.1. Introduction

En général, les gestionnaires de version (VCS) proposent une gestion de branches. Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne.

La branche par défaut dans Git s'appelle `master` ou `main`. Au fur et à mesure des validations, la branche `master` pointe vers le dernier des *commits* réalisés. À chaque validation, le pointeur de la branche `master` avance automatiquement.



La branche `master` ou `main` n'est pas une branche spéciale. Elle est identique à toutes les autres branches. La seule raison pour laquelle chaque dépôt en a une est que la commande `git init` la crée par défaut.

D'un point de vue technique, une branche dans Git est simplement un pointeur déplaçable vers un *commit*.

Pour créer une nouvelle branche, on utilise la commande `git branch <nom-branche>`. Cela crée simplement un nouveau pointeur vers le *commit* courant.

Git connaît la branche actuelle avec le pointeur spécial appelé `HEAD`. Dans Git, il s'agit simplement d'un pointeur sur la branche locale où l'on se trouve.

Pour l'instant, on se trouve toujours sur la branche `master`. En effet, la commande `git branch` n'a fait que créer une nouvelle branche et elle n'a pas fait basculer la copie de travail vers cette branche.

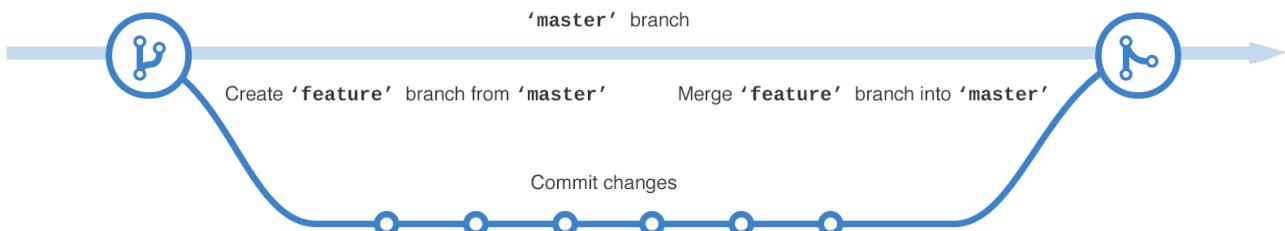
Pour basculer sur une branche existante, il suffit d'exécuter la commande `git checkout <nom-branche>`. Cela déplace `HEAD` pour le faire pointer vers la branche `<nom-branche>`.

Il est habituel de créer une nouvelle branche et de vouloir basculer sur cette nouvelle branche en même temps : pour cela on exécutera la commande `git checkout -b <nouvelle-branche>` (voir aussi `git switch`).



Il est important de noter que lorsque l'on change de branche avec Git, les fichiers du répertoire de travail sont modifiés. Si la copie de travail ou la zone d'index contiennent des modifications non validées qui sont en conflit avec la branche à extraire, Git n'autorisera pas le changement de branche. Le mieux est donc d'avoir une copie de travail propre au moment de changer de branche.

Une fois le travail réalisé (terminé et testé) dans la branche, il est prêt à être fusionné dans la branche `master`. On réalise ceci au moyen de la commande `git merge`.



À présent que le travail a été fusionné, on n'a plus besoin de la branche. On peut la supprimer avec l'option `-d` de la commande `git branch`.

## 8.2. Travailler avec les branches

On crée des nouvelles branches depuis `master` ou `main` à chaque nouvelle fonctionnalité ou nouvelle modification qu'il faut apporter au projet. Git permet de gérer plusieurs branches en parallèle et ainsi de cloisonner les travaux et d'éviter ainsi de mélanger des modifications du code source qui n'ont rien à voir entre elles.

En gardant une branche `master` ou `main` saine, on conserve ainsi une version du logiciel prête à être livrée à tout instant puisqu'on ne fusionne (`merge`) dedans que lorsque le développement d'une branche est bien terminé.



Un dépôt Git peut maintenir de nombreuses branches de développement.

Liens :

- [Manuel de référence en français](#) dans le chapitre "Les branches avec Git"
- [Wikilivre Git - Branches](#)

On commence par lister les branches existantes :

```
$ git branch -vv
* master 7cbe84f Ajout README

$ git branch --all
* master
```

On crée une branche pour réaliser un "travail" sur le projet :

```
$ git branch fonction-bienvenue
```

```
$ git branch  
  fonction-bienvenue  
* master
```

On bascule sur la nouvelle branche :

```
$ git checkout fonction-bienvenue  
Basculement sur la branche 'fonction-bienvenue'
```

```
$ git branch  
* fonction-bienvenue  
  master
```

On travaille dans la branche :

```
$ touch fonction-bienvenue.h  
$ touch fonction-bienvenue.cpp  
  
$ vim fonction-bienvenue.h
```

```
#ifndef FONCTION_BIENVENUE_H  
#define FONCTION_BIENVENUE_H  
  
void afficherBienvenue();  
  
#endif // FONCTION_BIENVENUE_H
```

```
$ vim fonction-bienvenue.cpp
```

```
#include "fonction-bienvenue.h"  
#include <iostream>  
  
void afficherBienvenue()  
{  
    std::cout << "Bienvenue le monde !" << std::endl;  
}
```

```
$ vim bienvenue.cpp
```

```
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
    afficherBienvenue();

    return 0;
}
```

On crée un **Makefile** :

```
$ touch Makefile
$ vim Makefile
```

```
TARGET := bienvenue
MODULE := fonction-bienvenue

CXX = g++ -c
LD = g++ -o
RM = rm -f
CXXFLAGS = -Wall -std=c++11
LDFLAGS =

$(info Fabrication du programme : $(TARGET))

all : $(TARGET)

$(TARGET): $(TARGET).o $(MODULE).o
    $(LD) $@ $(LDFLAGS) $^

$(TARGET).o: $(TARGET).cpp $(MODULE).h
    $(CXX) $(CXXFLAGS) $<

$(MODULE).o: $(MODULE).cpp $(MODULE).h
    $(CXX) $(CXXFLAGS) $<

.PHONY: clean

clean:
    $(RM) *.o

cleanall:
    $(RM) *.o $(TARGET)

rebuild: clean all
```

On teste le travail :

```
$ make
Fabrication du programme : bienvenue
g++ -c -Wall -std=c++11 bienvenue.cpp
g++ -c -Wall -std=c++11 fonction-bienvenue.cpp
g++ -o bienvenue bienvenue.o fonction-bienvenue.o

./bienvenue
Bienvenue le monde !
```

On valide les modifications :

```
$ git status
Sur la branche fonction-bienvenue
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la
copie de travail)

  modifié :      bienvenue.cpp
```

Fichiers non suivis:
 (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

```
Makefile
fonction-bienvenue.cpp
fonction-bienvenue.h
```

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")

```
$ git add Makefile fonction-bienvenue.cpp fonction-bienvenue.h
$ git add bienvenue.cpp
```

```
$ git status
Sur la branche fonction-bienvenue
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

  nouveau fichier : Makefile
  modifié :      bienvenue.cpp
  nouveau fichier : fonction-bienvenue.cpp
  nouveau fichier : fonction-bienvenue.h
```

```
$ git commit -m "Ajout de la fonction afficherBienvenue()"  
[fonction-bienvenue c8824fc] Ajout de la fonction afficherBienvenue()  
 4 files changed, 46 insertions(+), 3 deletions(-)  
  create mode 100644 Makefile  
  create mode 100644 fonction-bienvenue.cpp  
  create mode 100644 fonction-bienvenue.h
```

Vérification :

```
$ git status  
Sur la branche fonction-bienvenue  
rien à valider, la copie de travail est propre  
  
$ git log --oneline  
c8824fc (HEAD -> fonction-bienvenue) Ajout de la fonction afficherBienvenue()  
7cbe84f (master) Ajout README  
4717082 Affiche un message de bienvenue  
af1dcc8 Ajout du fichier .gitignore  
973e4f7 Création du programme principal  
bb344f4 Ajout du fichier bienvenue.cpp
```

On fusionne la branche dans **master** (ou **main**) :

*Basculement sur la branche principale :*

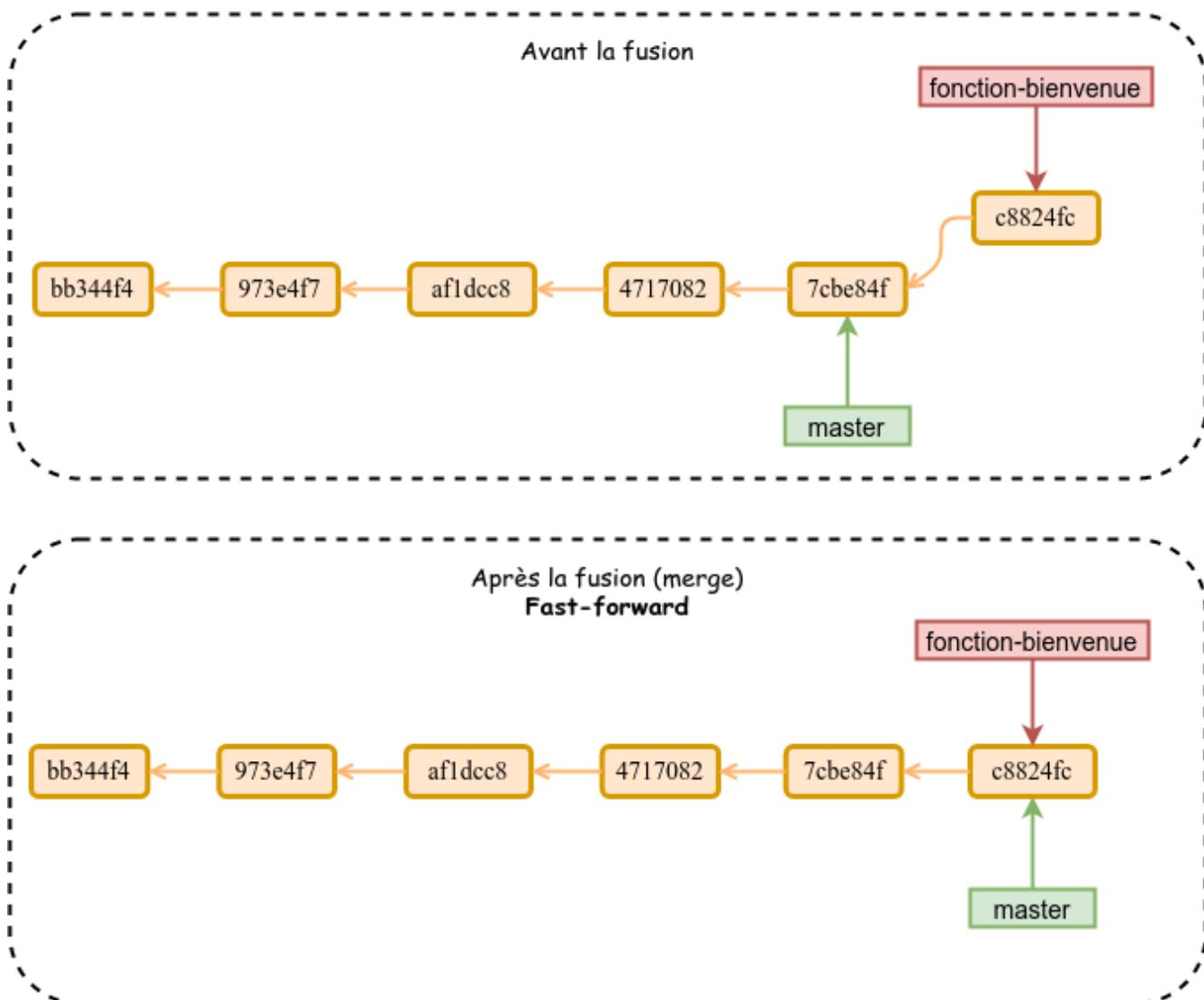
```
$ git checkout master  
Basculement sur la branche 'master'  
  
$ ls -l  
-rwxrwxr-x 1 tv tv 9008 août 11 14:08 bienvenue  
-rw-rw-r-- 1 tv tv 140 août 11 14:14 bienvenue.cpp  
-rw-rw-r-- 1 tv tv 1432 août 11 14:08 bienvenue.o  
-rw-rw-r-- 1 tv tv 2816 août 11 14:08 fonction-bienvenue.o  
-rw-rw-r-- 1 tv tv 63 août 11 10:11 README  
  
$ git status  
Sur la branche master  
rien à valider, la copie de travail est propre
```

## Fusion :

```
$ git merge fonction-bienvenue
Mise à jour 7cbe84f..c8824fc
Fast-forward
 Makefile           | 31 ++++++-----+
 bienvenue.cpp      |  5 +---+
 fonction-bienvenue.cpp |  7 ++++++
 fonction-bienvenue.h |  6 ++++++
 4 files changed, 46 insertions(+), 3 deletions(-)
 create mode 100644 Makefile
 create mode 100644 fonction-bienvenue.cpp
 create mode 100644 fonction-bienvenue.h
```



Lors de la fusion (**merge**), Git a simplement déplacé le pointeur (vers l'avant) : le *commit* **7cbe84f** vers **c8824fc**. Lorsque l'on cherche à fusionner un *commit* qui peut être atteint en parcourant l'historique depuis le *commit* d'origine, Git se contente d'avancer le pointeur car il n'y a pas de travaux divergents à fusionner. Ceci s'appelle un *fast-forward* (avance rapide).



Vérification :

```
$ ls -l
-rwxrwxr-x 1 tv tv 9008 août 11 14:08 bienvenue
-rw-rw-r-- 1 tv tv 122 août 11 14:16 bienvenue.cpp
-rw-rw-r-- 1 tv tv 1432 août 11 14:08 bienvenue.o
-rw-rw-r-- 1 tv tv 135 août 11 14:16 fonction-bienvenue.cpp
-rw-rw-r-- 1 tv tv 117 août 11 14:16 fonction-bienvenue.h
-rw-rw-r-- 1 tv tv 2816 août 11 14:08 fonction-bienvenue.o
-rw-rw-r-- 1 tv tv 459 août 11 14:16 Makefile
-rw-rw-r-- 1 tv tv 63 août 11 10:11 README

$ make
Fabrication du programme : bienvenue
g++ -c -Wall -std=c++11 bienvenue.cpp
g++ -c -Wall -std=c++11 fonction-bienvenue.cpp
g++ -o bienvenue bienvenue.o fonction-bienvenue.o
$ ./bienvenue
Bienvenue le monde !
```

On supprime la branche (cf. branche thématique dans la [Conclusion](#)) :

Avant :

```
$ git branch -vv
  fonction-bienvenue c8824fc Ajout de la fonction afficherBienvenue()
* master           c8824fc Ajout de la fonction afficherBienvenue()
```

Suppression d'une branche :

```
$ git branch -d fonction-bienvenue
Branche fonction-bienvenue supprimée (précédemment c8824fc).
```

Après :

```
$ git branch -vv
* master c8824fc Ajout de la fonction afficherBienvenue()

$ git log --oneline
c8824fc (HEAD -> master) Ajout de la fonction afficherBienvenue()
7cbe84f Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp
```

## 8.3. Retour sur le fonctionnement interne

Le répertoire de travail et dépôt local tp-git-sequence-1 actuel :

```
$ ls -l
-rwxrwxr-x 1 tv tv 9008 août 11 14:17 bienvenue
-rw-rw-r-- 1 tv tv 122 août 11 14:16 bienvenue.cpp
-rw-rw-r-- 1 tv tv 1432 août 11 14:17 bienvenue.o
-rw-rw-r-- 1 tv tv 135 août 11 14:16 fonction-bienvenue.cpp
-rw-rw-r-- 1 tv tv 117 août 11 14:16 fonction-bienvenue.h
-rw-rw-r-- 1 tv tv 2816 août 11 14:17 fonction-bienvenue.o
-rw-rw-r-- 1 tv tv 459 août 11 14:16 Makefile
-rw-rw-r-- 1 tv tv 111 août 11 17:17 README.md

$ git log --oneline
c479e51 (HEAD -> main, origin/main) Renommage README.md
470794d Modification du fichier README
c8824fc (tag: 1.0) Ajout de la fonction afficherBienvenue()
7cbe84f Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp
```

Le dépôt local contient l'historique des instantanés (*commits*). C'est une base de "données" (d'objets) qui peut contenir n'importe quel type d'objets (*commit*, *tree*, *blob* et *tag*). Git utilise des index (somme de contrôle calculée avec la [fonction de hachage SHA-1](#)) pour référencer les objets de la base.

L'objet **commit** correspond à une arborescence de fichiers (*tree*) enrichie de métadonnées comme un message de description, le nom de l'auteur, etc.

*Visualiser le contenu d'un objet commit :*

```
$ git show -s --pretty=raw bb344f4
commit bb344f417dbbf7f6725b24b293af2909bad6a519
tree e789bf9e379f78fefad662d9f0e3dffad003a8ba
author tvaira <tvaira@free.fr> 1627497186 +0200
committer tvaira <tvaira@free.fr> 1627497186 +0200
```

Ajout du fichier bienvenue.cpp

Il pointe également vers un ou plusieurs objets *commit* parents pour former un graphe d'historiques :

Son objet commit parent :

```
$ git show -s --pretty=raw 973e4f7
commit 973e4f7d830313e4ac9b08a332db767cdf28941f
tree a660d022174d628ff4ac03a086fb87f5e41e20ea
parent bb344f417dbbf7f6725b24b293af2909bad6a519
author tvaira <tvaira@free.fr> 1627498512 +0200
committer tvaira <tvaira@free.fr> 1627498512 +0200
```

Création du programme principal

L'objet **tree** décrit une arborescence de fichiers. Il est constitué d'une liste d'objets de type *blobs* (et des informations qui leur sont associées, tel que le nom du fichier et les permissions). Il peut contenir d'autres objets *trees* pour représenter les sous-répertoires.

Visualiser le contenu d'un objet tree :

```
$ git ls-tree e789bf9e
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    bienvenue.cpp

$ git ls-tree a660d022
100644 blob d315a7024964809e7a893ef0e5888023c8b833dd    bienvenue.cpp
```

L'objet **blob** (*binary large object*) représente le contenu d'un fichier. Git enregistre chaque révision dans un fichier en tant qu'objet *blob* unique.

Visualiser le contenu d'un objet blob :

```
$ git show e69de29b

$ git show d315a702
// TODO Indiquer ce que fait le programme

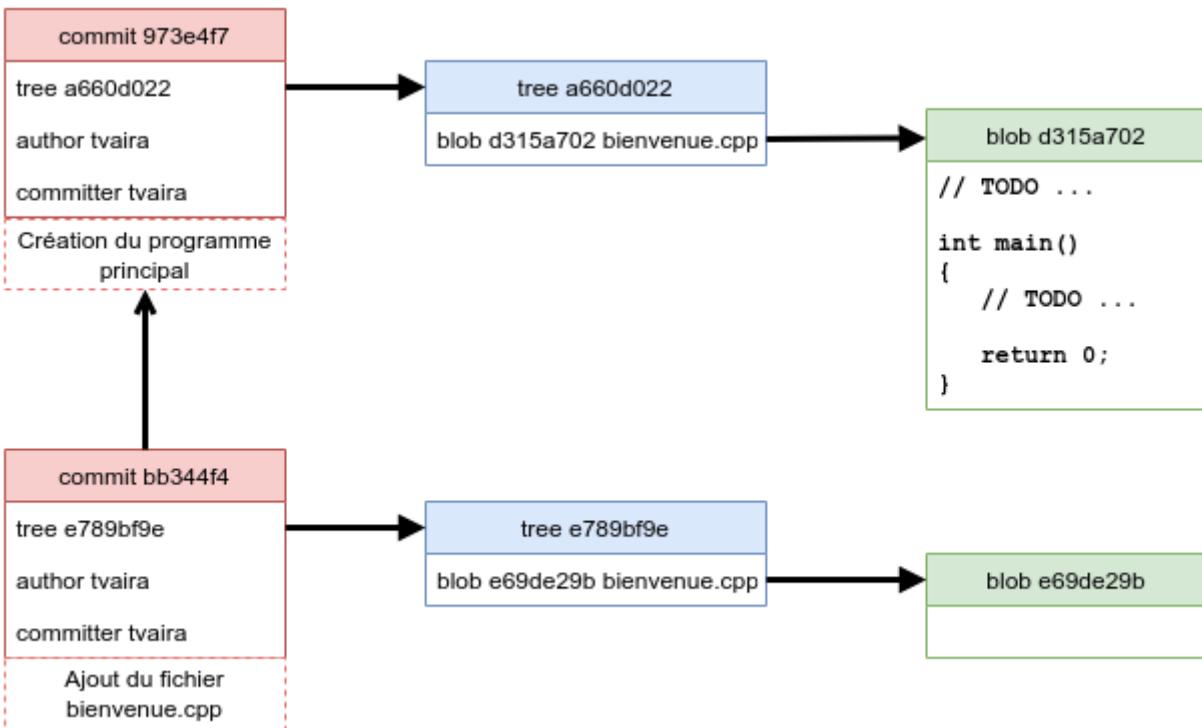
int main()
{
    // TODO Afficher un message de bienvenue

    return 0;
}
```



Un objet *blob* ne contient que le contenu du fichier. Il ne fait référence à rien d'autres : aucun attribut, même pas le nom de fichier !

On obtient cette « vue » de l'historique pour les deux premiers *commits* :



L'objet **tag** est une manière de nommer arbitrairement un *commit* spécifique pour l'identifier plus facilement. Il est en général utilisé pour marquer certains *commits*, par exemple par un numéro ou un nom de version. Un objet *tag* contient un nom d'objet (simplement nommé *object*), un type d'objet (ici *commit*), un nom de tag, le nom du « taggeur » et un message :

*Visualiser le contenu d'un objet tag :*

```
$ git cat-file tag 1.0
object c8824fc9dbb24745e722ad237a02105461bb7c3f
type commit
tag 1.0
tagger tvaira <tvaira@free.fr> 1628689213 +0200
```

La version 1.0

## 8.4. Conclusion

Dans Git, créer, développer, fusionner et supprimer des branches plusieurs fois par jour est un travail "normal".

On peut distinguer plusieurs types de branches :

- les branches au long cours : ce sont des branches ouvertes en permanence pour les différentes phases du cycle de développement.
- les branches thématiques : une branche thématique est une branche ayant une courte durée de vie créée et utilisée pour une fonctionnalité ou une tâche particulière (un correctif par exemple). On y réalise quelques *commits* et on supprime la branche immédiatement après l'avoir fusionnée dans la branche principale. Les branches thématiques sont utiles quelle que soit la taille du projet.



De nombreux développeurs travaillent avec Git en utilisant une méthode de développement basée sur les branches (cf. [Workflow git](#) et [Gitflow](#)).

## Cycle de travail

- Créer une branche thématique et basculer dessus (`git branch <branche>` puis `git checkout <branche>` ou `git checkout -b <branche>`)
  - Éditer des fichiers (`vim` ou un [EDI](#))
  - Ajouter les changement (`git add <fichier>`)
  - Valider les changements (`git commit -m "Message"`)
- Basculer sur la branche principale et fusionner la branche thématique (`git checkout master` puis `git merge <branche>`)
- Supprimer la branche thématique (`git branch -d <branche>`)

Les commandes que l'on utilise tout le temps :

- `git status`
- `git log ...`
- `git branch --all -vv`

## 9. Git hébergé

### Objectif

Mettre en oeuvre l'utilisation d'un dépôt distant.

Il est possible d'héberger des projets Git sur un site externe dédié à l'hébergement.

Liste : <https://git.wiki.kernel.org/index.php/GitHosting>

Quelques hébergeurs :

- [GitHub](#) est un service web d'hébergement (lancé en 2008) et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions [Git](#). Site officiel : <https://github.com/>



- [GitLab](https://about.gitlab.com/) est un logiciel libre de forge basé sur [Git](#) proposant les fonctionnalités de wiki, un système de suivi des bugs, l'intégration continue et la livraison continue. Site officiel : <https://about.gitlab.com/>



- [Bitbucket](https://bitbucket.org/) est un service web d'hébergement et de gestion de développement logiciel utilisant le logiciel de gestion de versions [Git](#). Site officiel : <https://bitbucket.org/>

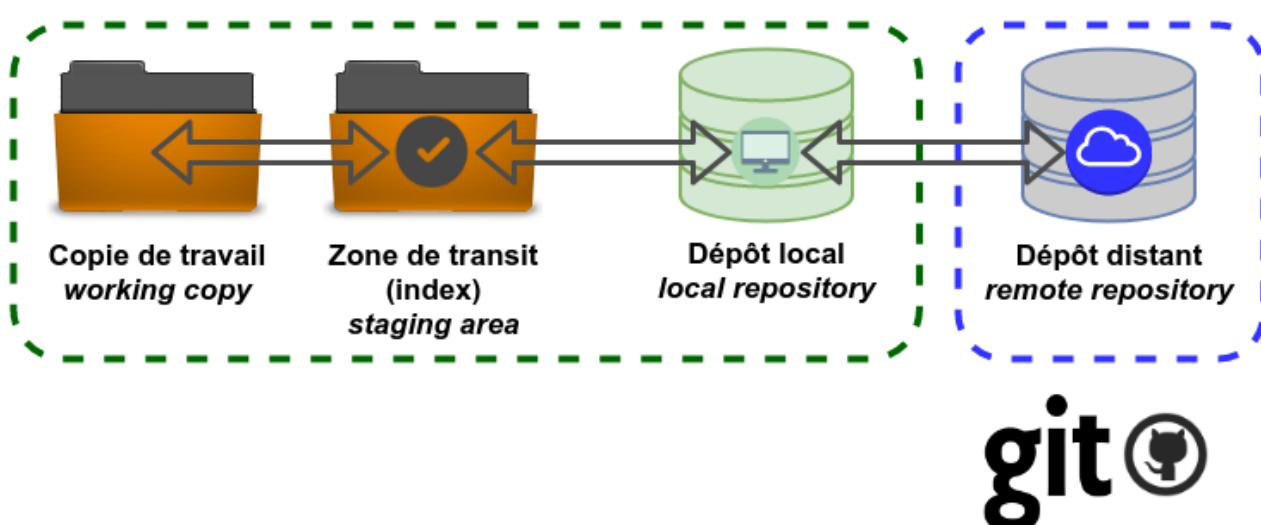


Ressources :

- [Git Handbook sur Github](#)
- [Git and GitHub learning resources](#)
- [Hello World](#)
- [Bitbucket Cloud resources](#)
- [Tutoriels](#)

## 9.1. Notion de dépôt distant

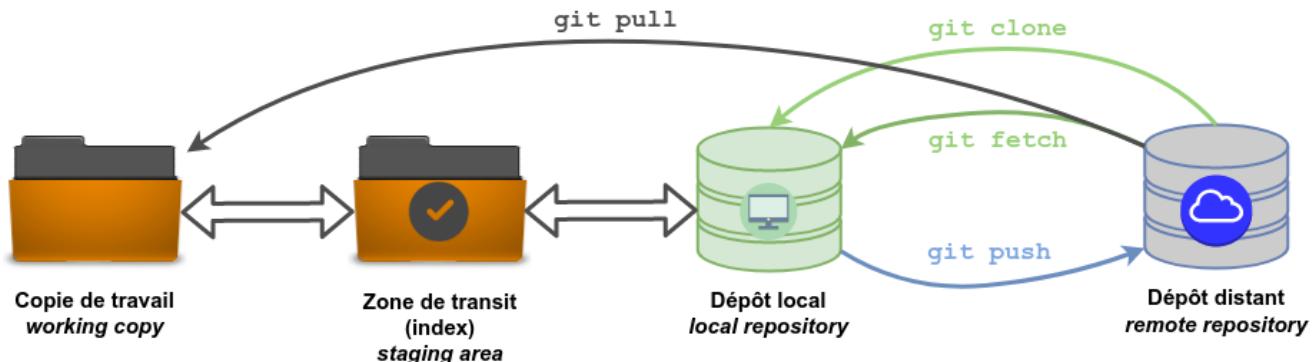
Un dépôt distant est un dépôt hébergé sur un serveur, généralement sur Internet.



Des commandes spécifiques seront utilisées pour synchroniser les dépôts local et distant :

- `git push` publie ("pousse") les nouvelles révisions du dépôt local sur le dépôt distant ;

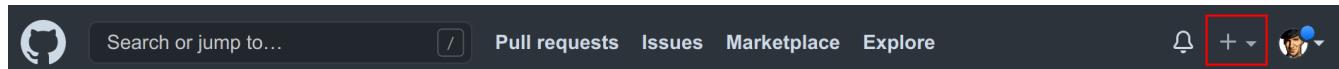
- `git fetch` récupère l'ensemble des changements (qui n'ont pas déjà été rapatriés localement) présents sur le serveur et met à jour la base de donnée locale (le dépôt local). Elle ne modifie pas le répertoire de travail.
- `git pull` consiste essentiellement en un `git fetch` immédiatement suivi par un `git merge` dans la plupart des cas. Le répertoire de travail peut donc être modifié.



## 9.2. Crédit d'un dépôt distant

Création d'un dépôt distant (*remote repository*) sur [GitHub](#) :

- On clique sur **+** pour créer un nouveau dépôt :



- On complète les informations du dépôt :

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*



Repository name \*

tp-cplusplus



Great repository names are short and memorable. Need inspiration? How about [animated-octo-engine](#)?

Description (optional)

TP C++ - Deuxième année BTS SNIR

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

**Initialize this repository with:**

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: C++ ▾

Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)

This will set main as the default branch. Change the default name in your [settings](#).

**Create repository**

Il y a deux façons de récupérer un dépôt Git :

- soit le dépôt local est déjà existant (`git init`) et il faut donc le relier à un dépôt distant (`git remote add origin https://github.com/nomutilisateur/dépot-distant.git`)
- soit le dépôt distant existe et il faut le copier (`git clone`) pour obtenir un dépôt local

## 9.3. Cloner un dépôt distant

La commande `git clone` effectuera les actions suivantes :

- créé un répertoire du nom du dépôt existant, initialisé avec un répertoire `.git` à l'intérieur,
- nomme automatiquement le serveur distant (*remote*) `origin`,

- tire l'historique,
- crée un pointeur sur l'état actuel de la branche `main` et l'appelle localement `origin/main`
- crée également une branche locale `main` qui démarre au même endroit que la branche `main` distante



`main` (ou `master`) et `origin` sont des noms donnés par défaut.

Clonage du dépôt :

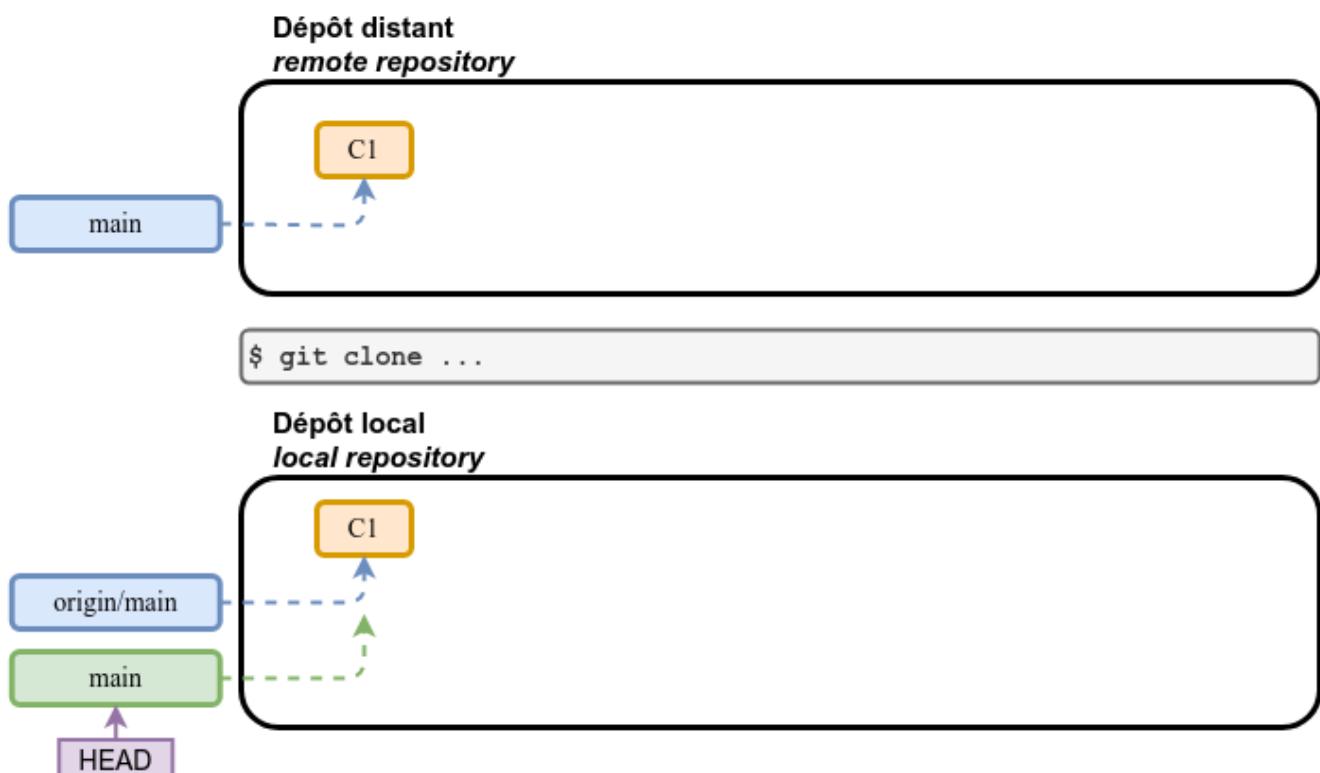
```
$ git clone https://github.com/tvaira/tp-cplusplus.git
```

État du dépôt :

```
$ cd tp-cplusplus/
$ ls -l
-rw-rw-r-- 1 tv tv 50 août 11 20:17 README.md

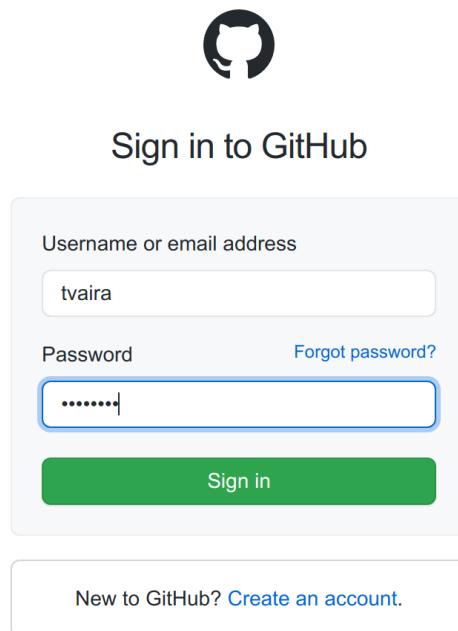
$ cat README.md
# tp-cplusplus
TP C++ - Deuxième année BTS SNIR

$ git remote -v
origin https://github.com/tvaira/tp-cplusplus.git (fetch)
origin https://github.com/tvaira/tp-cplusplus.git (push)
```



On suppose qu'un compte sur [GitHub](#) a été créé.

On se connecte :



Il est possible d'interagir avec le dépôt sur [GitHub](#) de plusieurs manières :

- [SSH](#)

L'URL d'accès au dépôt en SSH sera de la forme : <https://github.com/user/repo.git>

A screenshot of a GitHub repository page. At the top, there are three buttons: "Go to file", "Add file ▾", and "Code ▾". Below these is a "Clone" section with a "Clone" button and a help icon. It shows two cloning options: "HTTPS" and "SSH", with "SSH" underlined. The SSH URL is "git@github.com:tvaira/tp-git-sequence-...". Below this is a note "Use a password-protected SSH key." and a "Download ZIP" button at the bottom.

- Étape n°1 : générer des clés SSH

Sous GNU/Linux Ubuntu :

```
$ ssh-keygen -t ed25519 -C "tvaira@free.fr"  
  
$ eval "$(ssh-agent -s)"  
Agent pid 13867  
  
$ ssh-add ~/.ssh/id_ed25519  
Enter passphrase for ~/.ssh/id_ed25519:  
Identity added: ~/.ssh/id_ed25519 (tvaira@free.fr)  
  
$ sudo apt-get -y install xclip  
  
$ xclip -selection clipboard < ~/.ssh/id_ed25519.pub
```

- Étape n°2 : ajouter les clés SSH au compte GitHub
- Étape n°3 (facultative) : tester la connexion SSH

Accès en SSH :

```
$ git clone git@github.com:tvaira/tp-cplusplus.git  
Clonage dans 'tp-cplusplus'...  
remote: Enumerating objects: 4, done.  
remote: Counting objects: 100% (4/4), done.  
remote: Compressing objects: 100% (4/4), done.  
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0  
Réception d'objets: 100% (4/4), fait.
```

- HTTPS

L'URL d'accès au dépôt en HTTPS sera de la forme : <https://github.com/user/repo.git>

[Go to file](#)[Add file ▾](#)[Code ▾](#)

## Clone



[HTTPS](#) [SSH](#) [GitHub CLI](#)

<https://github.com/tvaira/tp-git-seque>



Use Git or checkout with SVN using the web URL.

## Download ZIP



Il faut maintenant créer un jeton d'accès personnel à utiliser à la place du mot de passe (<https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token>)

Accès en HTTPS :

```
$ git clone https://github.com/tvaira/tp-cplusplus.git
Clonage dans 'tp-cplusplus'...
Username for 'https://github.com': tvaira
Password for 'https://tvaira@github.com':
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Dépaquetage des objets: 100% (4/4), fait.
```

- [GitHub CLI](#)

La commande `gh` permet l'utilisation de GitHub en la ligne de commande (CLI).

[Go to file](#)[Add file ▾](#)[Code ▾](#)

## Clone

[HTTPS](#) [SSH](#) [GitHub CLI](#)

```
gh repo clone tvaira/tp-git-sequence-1
```



Work fast with our official CLI. [Learn more.](#)

## Download ZIP

Liens : <https://cli.github.com/>

Sous GNU/Linux Ubuntu, on peut installer `gh` avec la commande `sudo snap install gh`

```
$ tldr gh
```

Work seamlessly with GitHub from the command-line.

More information: <https://cli.github.com/>.

- Clone a GitHub repository locally:  
`gh repo clone owner/repository`
- Create a new issue:  
`gh issue create`
- View and filter the open issues of the current repository:  
`gh issue list`
- Create a pull request:  
`gh pr create`
- Locally check out the branch of a pull request, given its number:  
`gh pr checkout pr_number`
- Check the status of a repository's pull requests:  
`gh pr status`

Avant d'utiliser `gh`, il faut s'authentifier : `gh auth login`

Puis, on peut cloner un dépôt :

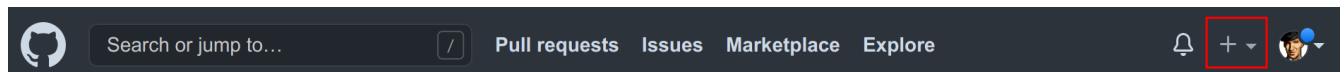
```
$ gh repo clone tvaira/tp-cplusplus.git
```

## 9.4. Exemple détaillé : développeur seul

On utilise le répertoire `tp-git-sequence-1` qui contient un dépôt local.

Un dépôt distant (*remote repository*) doit exister sur [GitHub](#) :

- On clique sur `+` pour créer un nouveau dépôt :



- On complète les informations du dépôt :

# Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner \*



tvaira ▾

Repository name \*

/ tp-git-sequence-1



Great repository names are short and memorable. Need inspiration? How about **studious-tribble**?

Description (optional)

 **Public**

Anyone on the internet can see this repository. You choose who can commit.

 **Private**

You choose who can see and commit to this repository.

## Initialize this repository with:

Skip this step if you're importing an existing repository.

**Add a README file**

This is where you can write a long description for your project. [Learn more](#).

**Add .gitignore**

Choose which files not to track from a list of templates. [Learn more](#).

**Choose a license**

A license tells others what they can and can't do with your code. [Learn more](#).

**Create repository**

À la fin, GitHub fournit les indications en fonction de la situation :

**Quick setup — if you've done this kind of thing before**

or    HTTPS    SSH    git@github.com:tvaira/tp-git-sequence-1.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 "# tp-git-sequence-1" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin git@github.com:tvaira/tp-git-sequence-1.git  
git push -u origin main
```

**...or push an existing repository from the command line**

```
git remote add origin git@github.com:tvaira/tp-git-sequence-1.git  
git branch -M main  
git push -u origin main
```

**...or import code from another repository**

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Il est possible alors de l'ajouter comme dépôt distant pour le dépôt local de l'ordinateur de travail et de synchroniser les deux emplacements.

```
$ git remote add origin git@github.com:tvaira/tp-git-sequence-1.git
```

Il faut renommer la branche **master** en **main** (l'option **-M** est un raccourci pour les options **--move** et **--force**) :

```
$ git branch -M main  
  
$ git branch -vv  
* main c8824fc Ajout de la fonction afficherBienvenue()
```

Puis, on synchronise les deux emplacements (local et distant) :

```
$ git push -u origin main
Décompte des objets: 21, fait.
Delta compression using up to 12 threads.
Compression des objets: 100% (17/17), fait.
Écriture des objets: 100% (21/21), 2.32 KiB | 395.00 KiB/s, fait.
Total 21 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To github.com:tvaira/tp-git-sequence-1.git
 * [new branch]      main -> main
La branche 'main' est paramétrée pour suivre la branche distante 'main' depuis
'origin'.

$ git pull
Déjà à jour.
```



L'option `--set-upstream` (alias `-u`) crée une référence qui permettra ensuite d'utiliser `git push` et `git pull` directement sans argument.

Lister les dépôts distants :

```
$ git remote -v
origin  git@github.com:tvaira/tp-git-sequence-1.git (fetch)
origin  git@github.com:tvaira/tp-git-sequence-1.git (push)

$ git branch --all
* main
  remotes/origin/main
```



Le serveur distant (*remote*) est nommé `origin` par défaut.

On modifie le fichier `README` sur le dépôt local :

```
$ vim README
# Bienvenue

Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction
'afficherBienvenue()'.

$ git add README

$ git commit -a -m "Modification du fichier README"
[main 470794d] Modification du fichier README
 1 file changed, 2 insertions(+), 1 deletion(-)
```

```
$ git status
Sur la branche main
Votre branche est en avance sur 'origin/main' de 1 commit.
  (utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
```

Et on l'envoie sur le dépôt distant :

```
$ git push
Décompte des objets: 3, fait.
Delta compression using up to 12 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 352 bytes | 352.00 KiB/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:tvaira/tp-git-sequence-1.git
  c8824fc..470794d  main -> main
```

```
$ git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.

rien à valider, la copie de travail est propre
```

```
$ git pull
Déjà à jour.
```

Dans [GitHub](#) :

The screenshot shows a GitHub repository interface. At the top, there are buttons for 'main' (with a dropdown arrow), '1 branch' (with a dropdown arrow), '0 tags', 'Go to file', 'Add file', and 'Code'. A commit from 'tvaira' titled 'Modification du fichier README' is shown, made 2 minutes ago with 7 commits. Below the commit history is a list of files: '.gitignore', 'Makefile', 'README', 'bienvenue.cpp', 'fonction-bienvenue.cpp', and 'fonction-bienvenue.h', each with their respective descriptions and timestamps. Below this is the 'README' file content:

```
# Bienvenue

Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction
`afficherBienvenue()`.
```



GitHub traite automatiquement le format [Markdown](#) si l'extension du fichier est `.md`. Ce qui n'est pas le cas ici ! Il faudra donc renommer le fichier `README.md`.

On peut éditer le nom du fichier directement dans [GitHub](#) :

The screenshot shows the GitHub file editor for the 'README.md' file. The file content is identical to the one above. In the commit changes dialog, the 'Renommage README.md' field is highlighted with a red box. The dialog also includes fields for an optional extended description and two radio button options: 'Commit directly to the `main` branch.' (selected) and 'Create a new branch for this commit and start a pull request.' Below the dialog are 'Commit changes' and 'Cancel' buttons.

Et c'est mieux :

main ▾ 1 branch 0 tags Go to file Add file ▾ Code ▾

 tvaira Renommage README.md c479e51 now ⏲ 8 commits

 .gitignore	Ajout du fichier .gitignore	14 days ago
 Makefile	Ajout de la fonction afficherBienvenue()	3 hours ago
 README.md	Renommage README.md	now
 bienvenue.cpp	Ajout de la fonction afficherBienvenue()	3 hours ago
 fonction-bienvenue.cpp	Ajout de la fonction afficherBienvenue()	3 hours ago
 fonction-bienvenue.h	Ajout de la fonction afficherBienvenue()	3 hours ago

**README.md** 

## Bienvenue

Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction `afficherBienvenue()`.

On re-synchronise les deux emplacements :

*Avant :*

```
$ ls -l
-rw-rw-r-x 1 tv tv 9008 août 11 14:17 bienvenue
-rw-rw-r-- 1 tv tv 122 août 11 14:16 bienvenue.cpp
-rw-rw-r-- 1 tv tv 1432 août 11 14:17 bienvenue.o
-rw-rw-r-- 1 tv tv 135 août 11 14:16 fonction-bienvenue.cpp
-rw-rw-r-- 1 tv tv 117 août 11 14:16 fonction-bienvenue.h
-rw-rw-r-- 1 tv tv 2816 août 11 14:17 fonction-bienvenue.o
-rw-rw-r-- 1 tv tv 459 août 11 14:16 Makefile
-rw-rw-r-- 1 tv tv 111 août 11 17:03 README
```

Récupère les modifications du dépôt distant :

```
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
Dépaquetage des objets: 100% (2/2), fait.
Depuis github.com:tvaira/tp-git-sequence-1
  470794d..c479e51  main      -> origin/main
Mise à jour 470794d..c479e51
Fast-forward
 README => README.md | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README => README.md (100%)
```

Après :

```
$ ls -l
-rwxrwxr-x 1 tv tv 9008 août 11 14:17 bienvenue
-rw-rw-r-- 1 tv tv 122 août 11 14:16 bienvenue.cpp
-rw-rw-r-- 1 tv tv 1432 août 11 14:17 bienvenue.o
-rw-rw-r-- 1 tv tv 135 août 11 14:16 fonction-bienvenue.cpp
-rw-rw-r-- 1 tv tv 117 août 11 14:16 fonction-bienvenue.h
-rw-rw-r-- 1 tv tv 2816 août 11 14:17 fonction-bienvenue.o
-rw-rw-r-- 1 tv tv 459 août 11 14:16 Makefile
-rw-rw-r-- 1 tv tv 111 août 11 17:17 README.md

$ git log --oneline
c479e51 (HEAD -> main, origin/main) Renommage README.md
470794d Modification du fichier README
c8824fc (tag: 1.0) Ajout de la fonction afficherBienvenue()
7cbe84f Ajout README
4717082 Affiche un message de bienvenue
af1dcc8 Ajout du fichier .gitignore
973e4f7 Création du programme principal
bb344f4 Ajout du fichier bienvenue.cpp
```

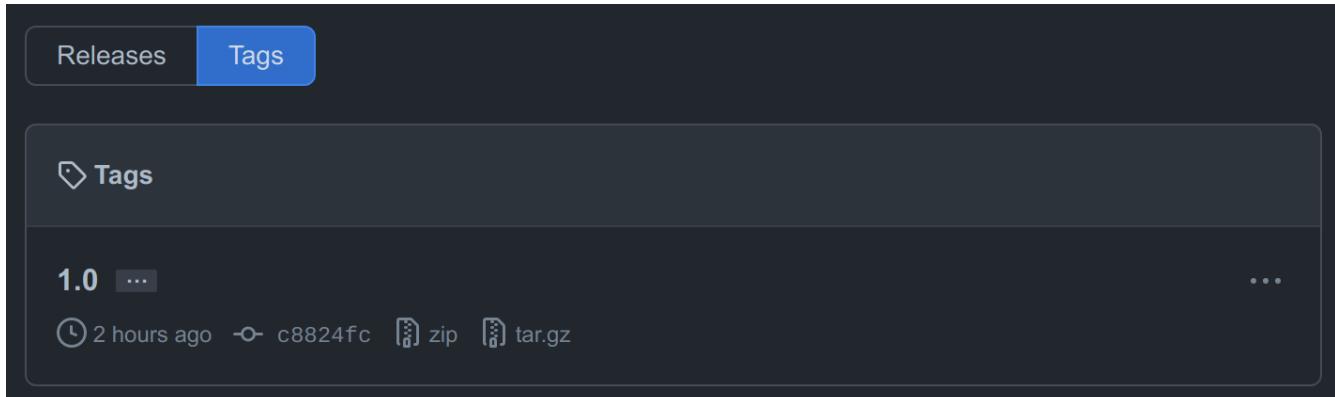


Les tags ne sont pas poussés (*push*) automatiquement.



```
$ git push --tags
Décompte des objets: 1, fait.
Écriture des objets: 100% (1/1), 154 bytes | 154.00 KiB/s, fait.
Total 1 (delta 0), reused 0 (delta 0)
To github.com:tvaira/tp-git-sequence-1.git
 * [new tag]          1.0 -> 1.0
```

Maintenant, le *tag* 1.0 est accessible à partir de [GitHub](#) :



On peut récupérer une archive compressée du projet au format **zip** ou **tar.gz** !

## 9.5. Branche de suivi

Une branche de suivi (*tracking branch*) est une branche locale qui est en relation directe avec une branche distante (*upstream branch*).

Les branches de suivi peuvent servir :

- à sauvegarder son travail sur la branche dans un dépôt distant
- partager son travail sur la branche avec d'autres développeurs



Dans le cadre d'un travail collaboratif, on pourra aussi décider d'utiliser des branches locales privées que l'on ne souhaite pas partager.

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement une branche de suivi (c'est l'option par défaut **--track** de la commande **git checkout**). Si la branche distante n'existe pas encore, il faudra utiliser l'option **-u** ou **--set-upstream-to** pour créer le suivi.

Si on se trouve sur une branche de suivi :

- **git push** sélectionne automatiquement le serveur vers lequel pousser les modifications.
- **git pull** récupère toutes les références distantes et fusionne automatiquement la branche distante correspondante dans la branche actuelle.

On souhaite modifier la fonction `afficherBienvenue()` pour qu'elle soit plus "générique" en recevant en argument le message à afficher. Pour cela on crée une branche qui va permettre de réaliser ce travail de manière isolée.

L'état du dépôt local est le suivant :

```
$ git branch -v
* main c479e51 [origin/main] Renommage README.md

$ git log --oneline
c479e51 (HEAD -> main, origin/main) Renommage README.md
470794d Modification du fichier README
...
...
```

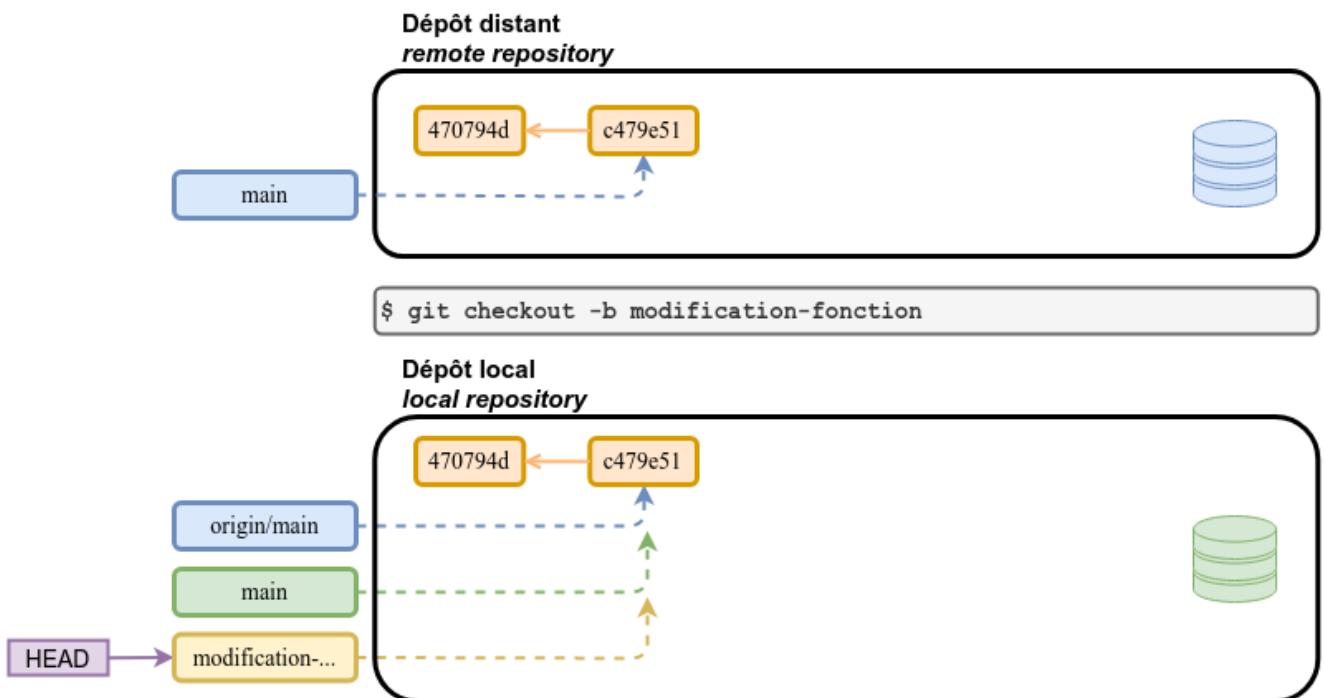
The screenshot shows a GitHub repository interface. At the top, there's a dropdown menu set to 'main'. Below it, a tree view shows two main commit sections. The first section, 'Commits on Aug 11, 2021', lists four commits by user 'tvaira': 'Renommage README.md' (commit c479e51), 'Modification du fichier README' (commit 470794d), 'Ajout de la fonction afficherBienvenue()' (commit c8824fc), and 'Ajout README' (commit 7cbe84f). The second section, 'Commits on Jul 28, 2021', lists four commits by user 'tvaira': 'Affiche un message de bienvenue' (commit 4717082), 'Ajout du fichier .gitignore' (commit af1dcc8), 'Création du programme principal' (commit 973e4f7), and 'Ajout du fichier bienvenue.cpp' (commit bb344f4).

On crée une branche `modification-fonction` et on bascule dessus :

```
$ git checkout -b modification-fonction
Basculement sur la nouvelle branche 'modification-fonction'

$ git branch -v
  main           c479e51 [origin/main] Renommage README.md
* modification-fonction c479e51 Renommage README.md

$ git log --oneline
c479e51 (HEAD -> modification-fonction, origin/main, main) Renommage README.md
...
...
```



On "travaille" sur le code :

```
$ vim fonction-bienvenue.h
```

```
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

#include <string>

void afficherMessage(std::string message);

#endif // FONCTION_BIENVENUE_H
```

```
$ vim fonction-bienvenue.cpp
```

```
#include "fonction-bienvenue.h"
#include <iostream>

void afficherMessage(std::string message)
{
    std::cout << message << std::endl;
}
```

```
$ vim bienvenue.cpp
```

```
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
    afficherMessage("Bienvenue le monde !");

    return 0;
}
```

On teste :

```
$ make rebuild
$ ./bienvenue
Bienvenue le monde !
```

On ajoute les fichiers dans l'index :

```
$ git add fonction-bienvenue.h
$ git add fonction-bienvenue.cpp
$ git add bienvenue.cpp
```

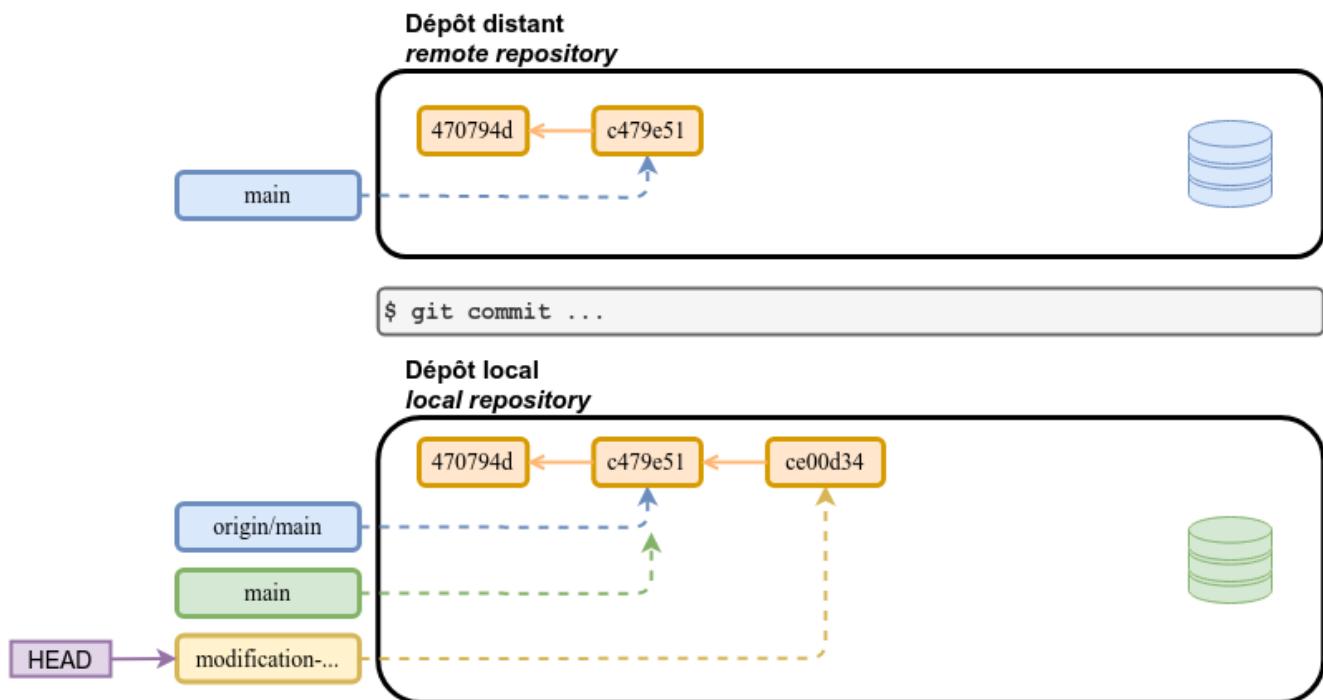
Et on valide les changements (*commit*) :

```
$ git commit -m "Modification afficherBienvenue en afficherMessage"
```

Vérification :

```
$ git branch -v
  main           c479e51 [origin/main] Renommage README.md
* modification-fonction ce00d34 Modification afficherBienvenue en afficherMessage

$ git log --oneline
ce00d34 (HEAD -> modification-fonction) Modification afficherBienvenue en
afficherMessage
c479e51 (origin/main, main) Renommage README.md
470794d Modification du fichier README
...
```



On crée une branche de suivi :

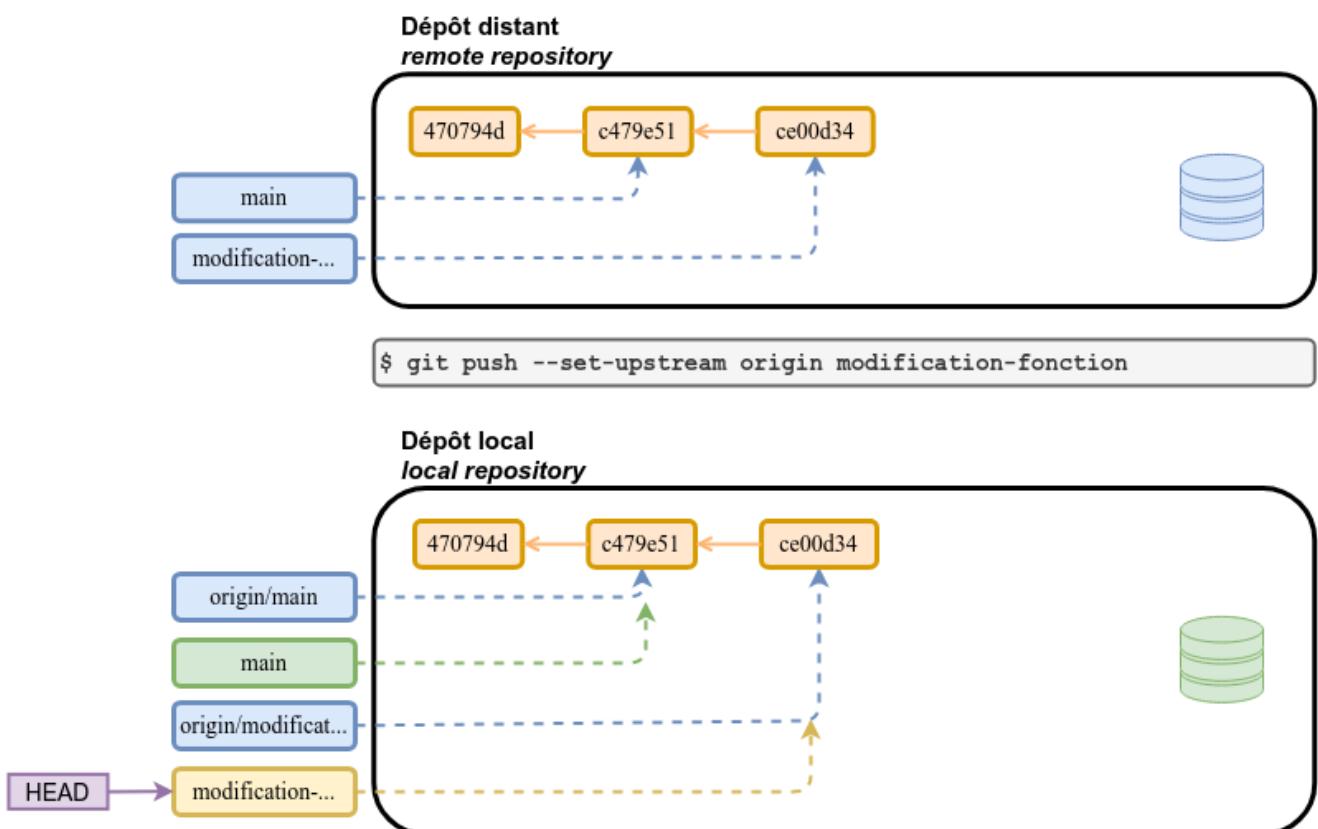
```
$ git push --set-upstream origin modification-fonction
Décompte des objets: 5, fait.
Delta compression using up to 12 threads.
Compression des objets: 100% (5/5), fait.
Écriture des objets: 100% (5/5), 660 bytes | 660.00 KiB/s, fait.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'modification-fonction' on GitHub by visiting:
remote:     https://github.com/tvaira/tp-git-sequence-1/pull/new/modification-
fonction
remote:
To github.com:tvaira/tp-git-sequence-1.git
 * [new branch]      modification-fonction -> modification-fonction
La branche 'modification-fonction' est paramétrée pour suivre la branche distante
'modification-fonction' depuis 'origin'.
```

Vérification :

```
$ git branch -vv
  main           c479e51 [origin/main] Renommage README.md
* modification-fonction ce00d34 [origin/modification-fonction] Modification
  afficherBienvenue en afficherMessage

git log --oneline
ce00d34 (HEAD -> modification-fonction, origin/modification-fonction) Modification
  afficherBienvenue en afficherMessage
c479e51 (origin/main, main) Renommage README.md
470794d Modification du fichier README

$ git ls-remote
From git@github.com:tvaira/tp-git-sequence-1.git
c479e51a64712908cf823b053b72d75a015d2cf8    HEAD
c479e51a64712908cf823b053b72d75a015d2cf8    refs/heads/main
ce00d3449aa40aa44d51effcc66c796eb9b2ed20    refs/heads/modification-fonction
```



This branch is 1 commit ahead of main.

Contribute

tvaira Modification afficherBienvenue en afficherMessage ce00d34 19 minutes ago 9 commits

À partir d'ici, il y a deux possibilités pour fusionner la branche dans la branche principale. Dans le cadre d'un travail collaboratif, on pourrait (devrait ?) créer une *Pull Request* (une demande modification) comme l'indique le message "*Create a pull request for 'modification-fonction' on GitHub ...*" (cf. [Pull Request et Révision de code](#)). *Pull Request* peut être traduit par « Proposition de révision » (PR). C'est l'action qui consiste à demander au détenteur du dépôt de référence de prendre en compte des modifications d'un autre dépôt (*fork* ou local).



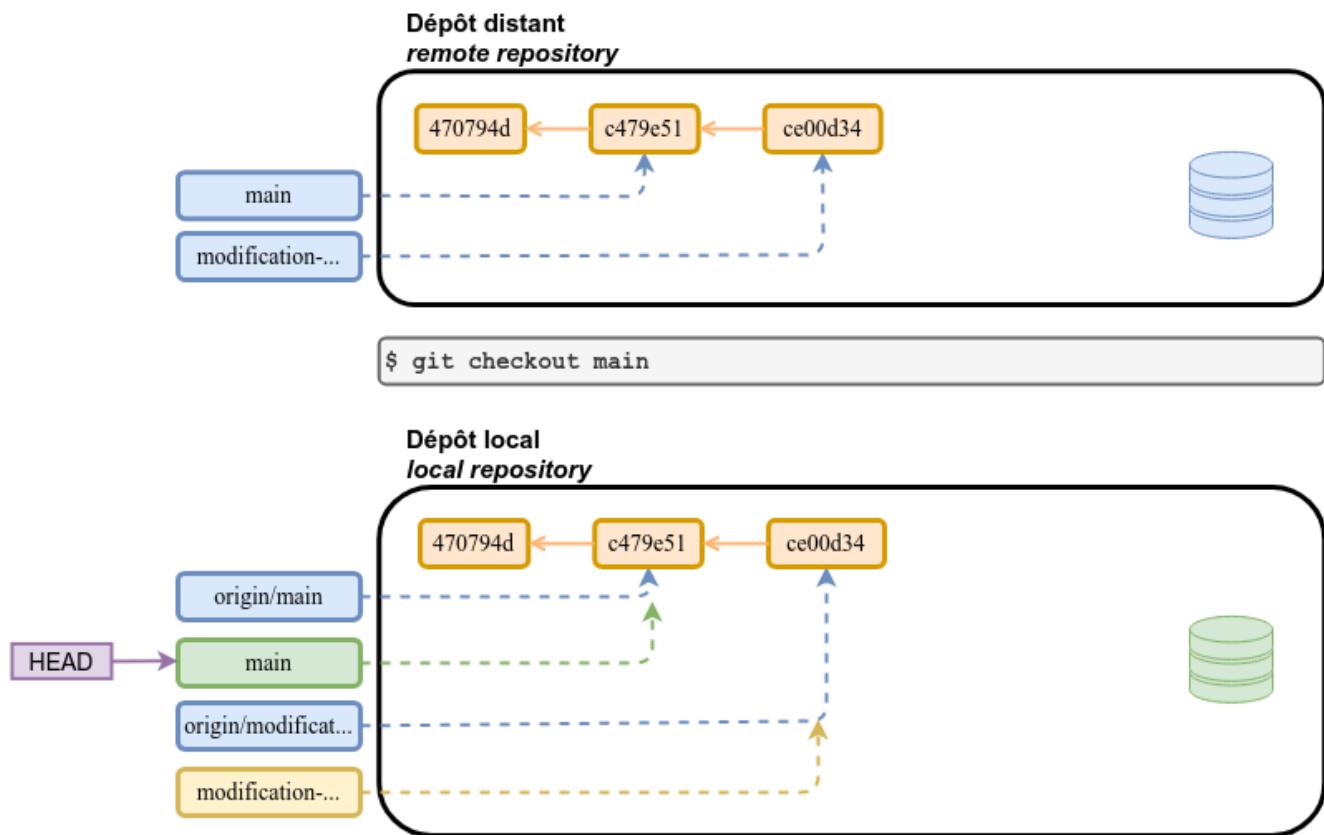
Dans une situation "Développeur seul", cela n'a pas d'intérêt donc on peut continuer sur le dépôt local pour faire la fusion (*merge*) puis la "publier" (*push*) sur le dépôt distant.

Il faut basculer sur la branche principale :

```
$ git checkout main
Basculement sur la branche 'main'
Votre branche est à jour avec 'origin/main'.

$ git branch -vv
* main           c479e51 [origin/main] Renommage README.md
  modification-fonction ce00d34 [origin/modification-fonction] Modification
  afficherBienvenue en afficherMessage

$ git log --oneline
c479e51 (HEAD -> main, origin/main) Renommage README.md
470794d Modification du fichier README
...
```



On fusionne la branche **modification-fonction** dans **main** :

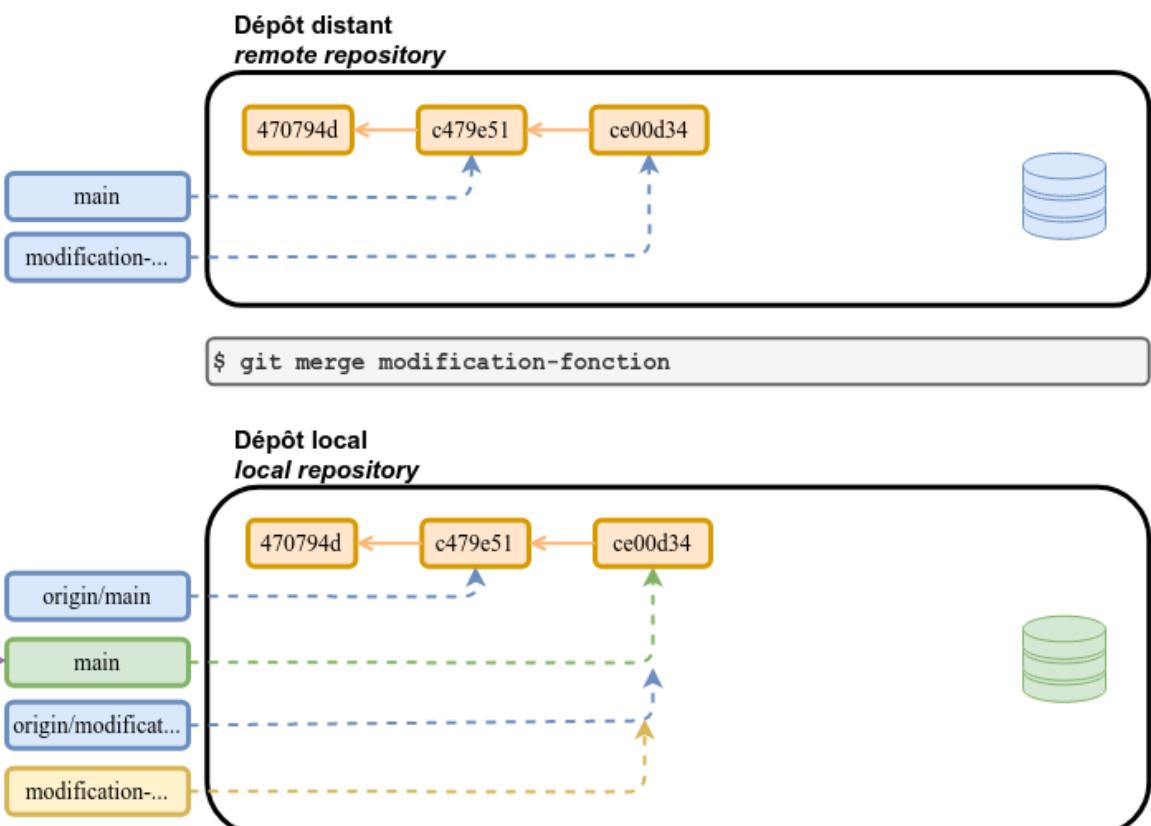
```

$ git merge modification-fonction
Mise à jour c479e51..ce00d34
Fast-forward
 bienvenue.cpp      | 2 ++
 fonction-bienvenue.cpp | 4 +---+
 fonction-bienvenue.h | 4 +---+
 3 files changed, 6 insertions(+), 4 deletions(-)
  
```

## Vérification :

```
$ git branch -vv
* main           ce00d34 [origin/main: en avance de 1] Modification
  afficherBienvenue en afficherMessage
    modification-fonction ce00d34 [origin/modification-fonction] Modification
  afficherBienvenue en afficherMessage

$ git log --oneline
ce00d34 (HEAD -> main, origin/modification-fonction, modification-fonction)
Modification afficherBienvenue en afficherMessage
c479e51 (origin/main) Renommage README.md
470794d Modification du fichier README
...
rien à valider, la copie de travail est propre
```



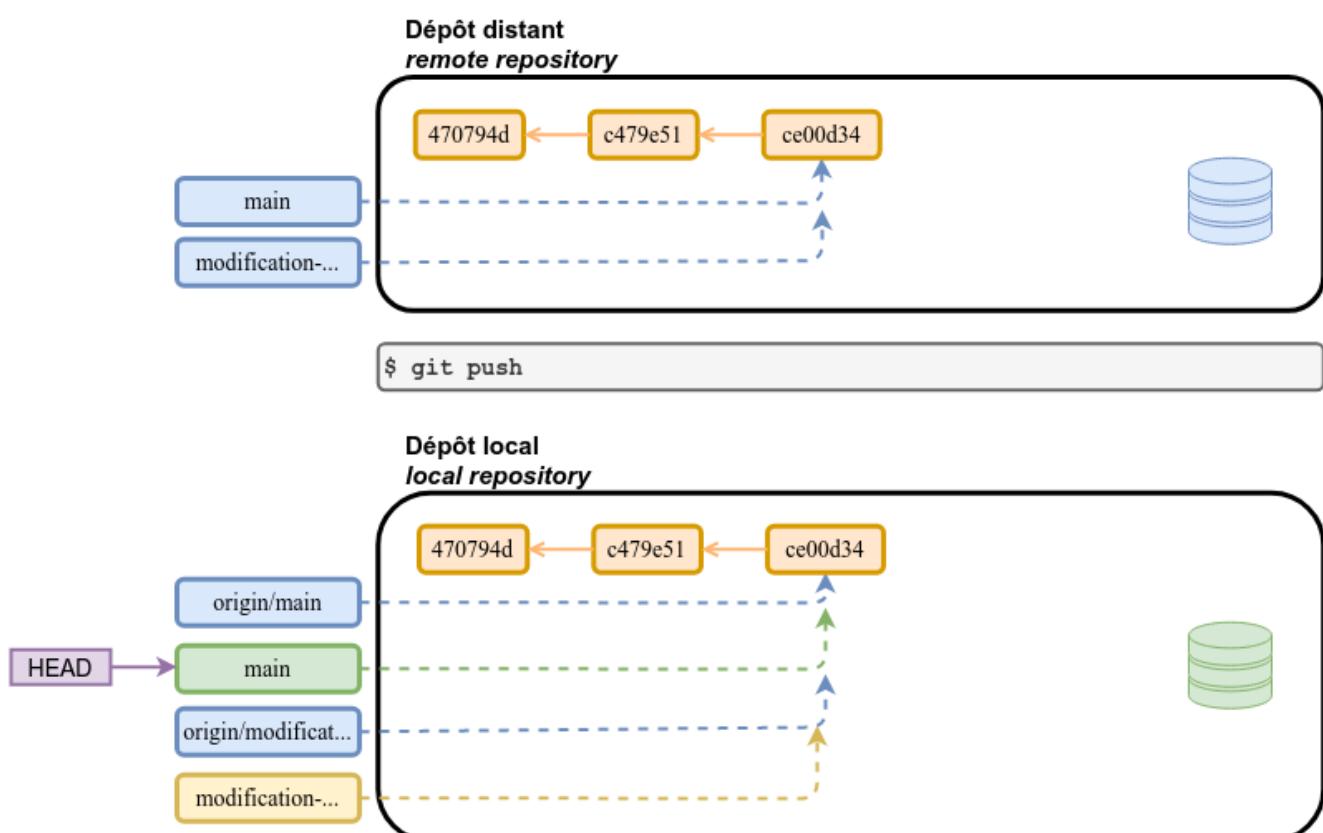
Il faut synchroniser le dépôt distant :

```
$ git push
Total 0 (delta 0), reused 0 (delta 0)
To github.com:tvaira/tp-git-sequence-1.git
  c479e51..ce00d34  main -> main
```

Vérification :

```
$ git branch -vv
* main           ce00d34 [origin/main] Modification afficherBienvenue en
afficherMessage
               modification-fonction ce00d34 [origin/modification-fonction] Modification
afficherBienvenue en afficherMessage

$ git log --oneline
ce00d34 (HEAD -> main, origin/modification-fonction, origin/main, modification-
fonction) Modification afficherBienvenue en afficherMessage
c479e51 Renommage README.md
470794d Modification du fichier README
...
```



On peut visualiser le *commit* de fusion :

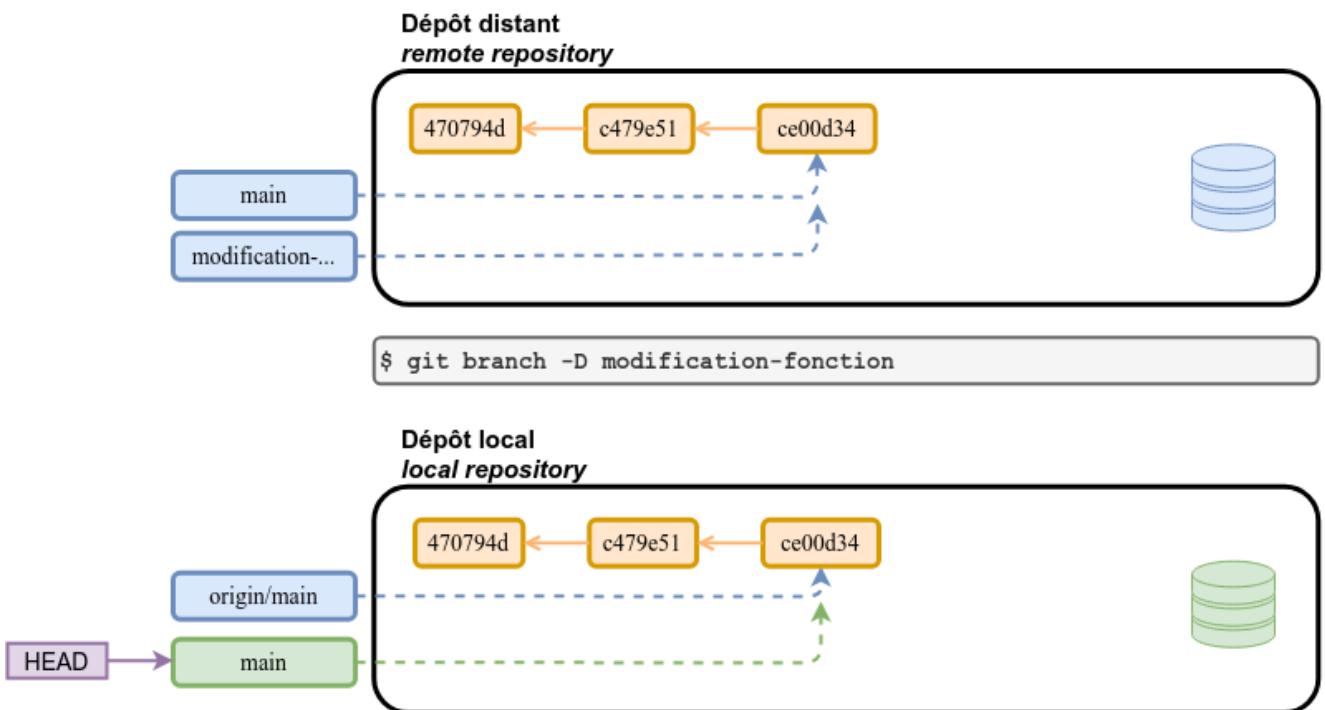
On supprime la branche locale (c'est une branche thématique) : (l'option **-D** force la suppression)

```
$ git branch -D modification-fonction
Branche modification-fonction supprimée (précédemment ce00d34).
```

Vérification :

```
$ git branch -vv
* main ce00d34 [origin/main] Modification afficherBienvenue en afficherMessage

$ git log --oneline
ce00d34 (HEAD -> main, tag: 1.1, origin/modification-fonction, origin/main)
Modification afficherBienvenue en afficherMessage
c479e51 Renommage README.md
470794d Modification du fichier README
...
...
```

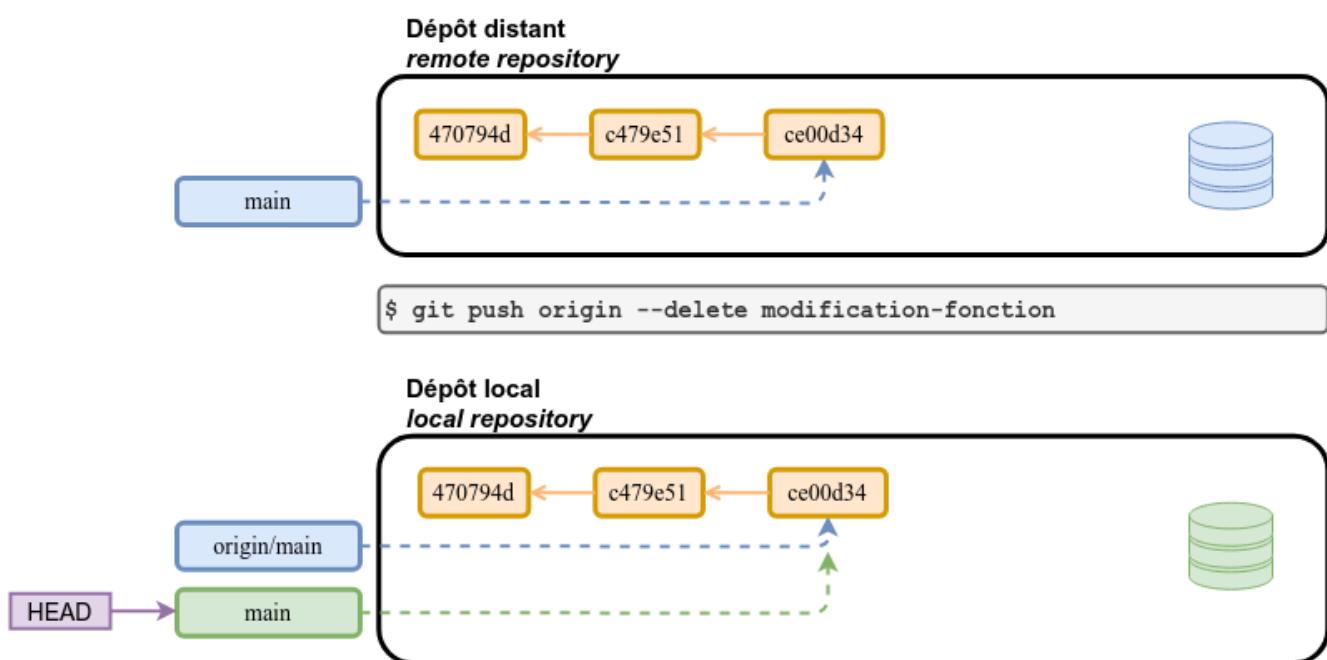


On supprime la branche distante :

```
$ git push origin --delete modification-fonction
To github.com:tvaira/tp-git-sequence-1.git
 - [deleted]           modification-fonction
```

Vérification :

```
$ git log --oneline  
ce00d34 (HEAD -> main, tag: 1.1, origin/main) Modification afficherBienvenue en  
afficherMessage  
c479e51 Renommage README.md  
470794d Modification du fichier README  
  
$ git ls-remote  
From git@github.com:tvaira/tp-git-sequence-1.git  
ce00d3449aa40aa44d51effcc66c796eb9b2ed20    HEAD  
ce00d3449aa40aa44d51effcc66c796eb9b2ed20    refs/heads/main  
...
```



On peut créer et publier un nouveau *tag* :

```
$ git tag -a 1.1 -m 'La version 1.1'  
  
$ git push --tags  
Décompte des objets: 1, fait.  
Écriture des objets: 100% (1/1), 153 bytes | 153.00 KiB/s, fait.  
Total 1 (delta 0), reused 0 (delta 0)  
To github.com:tvaira/tp-git-sequence-1.git  
 * [new tag]      1.1 -> 1.1
```

The screenshot shows the GitHub 'Tags' page for a repository. At the top, there are tabs for 'Releases' and 'Tags', with 'Tags' being the active tab. Below the tabs, there is a header 'Tags' with a back arrow icon. Two releases are listed: '1.1' and '1.0'. Release '1.1' was created 1 minute ago from commit 'ce00d34' and includes zip and tar.gz files. Release '1.0' was created 5 days ago from commit 'c8824fc' and also includes zip and tar.gz files.

Sur [GitHub](#), on peut créer des versions livrables "release" :

The screenshot shows the GitHub 'Releases' page for a repository. The 'Releases' tab is active. A 'Target: main' dropdown is set to 'main'. A message says 'Excellent! This tag will be created from the target when you publish this release.' A text input field contains 'v1.1'. Below it are 'Write' and 'Preview' buttons. A rich text editor toolbar is visible. A large text area for 'Describe this release' is empty. Below it is a file upload section with a placeholder 'Attach files by dragging & dropping, selecting or pasting them.' A file named 'bienvenue' is listed with a size of '(0.01 MB)' and a delete 'X' button. Below this is a dashed box for attaching binaries with a placeholder '↓ Attach binaries by dropping them here or selecting them.'. A checkbox labeled 'This is a pre-release' is checked, with a note: 'We'll point out that this release is identified as non-production ready.' At the bottom are 'Publish release' and 'Save draft' buttons.



On peut ajouter des fichiers comme ici l'exécutable.

The screenshot shows the GitHub release page for version 1.1. It includes a summary of the release, a list of assets (bienvenue, Source code (zip), Source code (tar.gz)), and a detailed commit history.

Au final, l'état du projet sur [GitHub](#) est le suivant :

The screenshot shows the GitHub repository page for 'tvaira/bienvenue'. It displays the repository structure, a README.md file containing a 'Bienvenue' section, and various repository statistics such as releases, packages, and languages.

Quelques commandes supplémentaires :

```
$ git remote show origin
* distante origin
  URL de rapatriement : https://github.com/tvaira/tp-git-sequence-1.git
  URL push : https://github.com/tvaira/tp-git-sequence-1.git
  Branche HEAD : main
  Branches distantes :
    main              suivi
    modification-fonction suivi
  Branches locales configurées pour 'git pull' :
    main              fusionne avec la distante main
    modification-fonction fusionne avec la distante modification-fonction
  Références locales configurées pour 'git push' :
    main              pousse vers main          (à jour)
    modification-fonction pousse vers modification-fonction (à jour)
```

Les branches qui ont été fusionnées (ou pas) :

```
$ git branch --merged | grep -v \* | xargs  
modification-fonction  
  
$ git branch --no-merged | grep -v \* | xargs
```

Pour éviter de conserver en local des anciennes branches généralement fusionnées, on les nettoye avec :

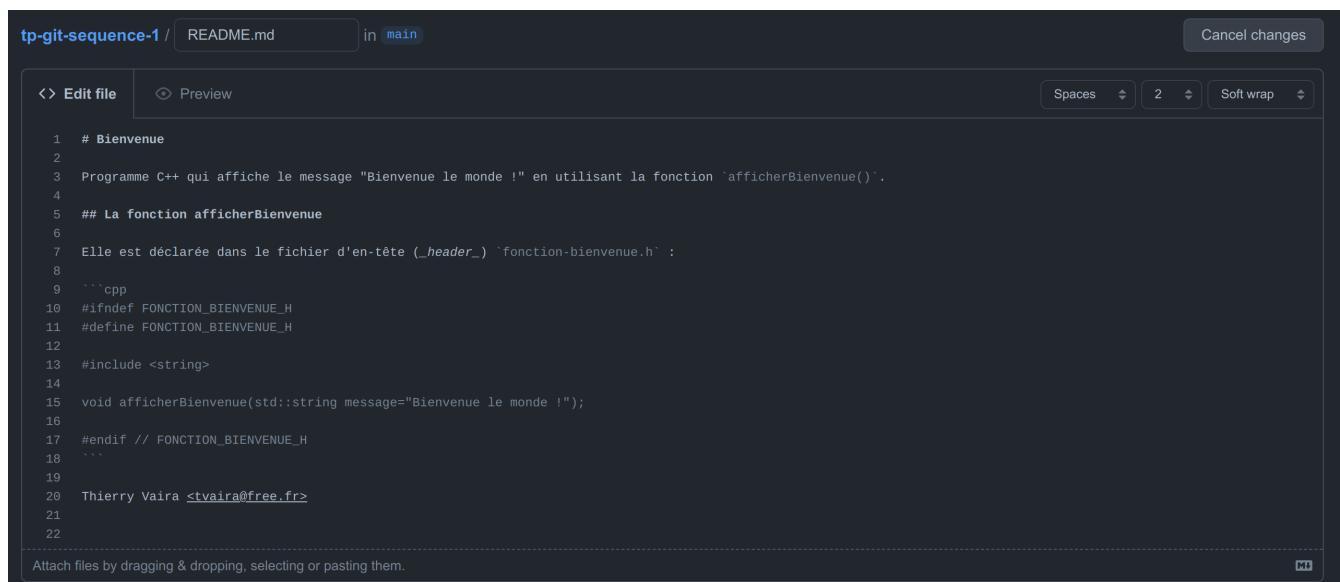
```
$ git remote prune origin
```

## 9.6. Travailler dans GitHub

GitHub permet aussi d'éditer les fichiers, notamment ceux en [Markdown](#) :



Dans la nouvelle fenêtre, on peut éditer le fichier :



Le prévisualiser :

[Edit file](#) [Preview](#) [Show diff](#)

## Bienvenue

Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction `afficherBienvenue()`.

### La fonction `afficherBienvenue`

Elle est déclarée dans le fichier d'en-tête (*header*) `fonction-bienvenue.h` :

```
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

#include <string>

void afficherBienvenue(std::string message="Bienvenue le monde !");

#endif // FONCTION_BIENVENUE_H
```

Thierry Vaira [tvaira@free.fr](mailto:tvaira@free.fr)

Et terminer en réalisant le *commit* :

 **Commit changes**

Update README.md

Add an optional extended description...

Commit directly to the `main` branch.

Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

[Commit changes](#) [Cancel](#)

Le *commit* ayant été réalisé directement sur le dépôt distant, il n'est pas (encore) disponible sur le dépôt local :

```
$ git status
Sur la branche main
Votre branche est en retard sur 'origin/main' de 1 commit, et peut être mise à jour en avance rapide.
  (utilisez "git pull" pour mettre à jour votre branche locale)

rien à valider, la copie de travail est propre
```

On peut mettre à jour le dépôt local et le répertoire de travail directement avec la commande `git pull` (équivalente à `git fetch` suivi d'un `git merge`) :

```
$ git pull
Mise à jour 3cf9129..73c7f78
Fast-forward
 README.md | 45 ++++++-----+
 1 file changed, 44 insertions(+), 1 deletion(-)
```

Vérification :

```
$ cat README.md
# Bienvenue
```

Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction 'afficherBienvenue()'.

## La fonction afficherBienvenue

Elle est déclarée dans le fichier d'en-tête (\_header\_) 'fonction-bienvenue.h' :

```
'''cpp
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

#include <string>

void afficherBienvenue(std::string message="Bienvenue le monde !");

#endif // FONCTION_BIENVENUE_H
'''
```

Et définie dans le fichier 'fonction-bienvenue.cpp' :

```
'''cpp
#include "fonction-bienvenue.h"
#include <iostream>

void afficherBienvenue(std::string message/*="Bienvenue le monde !"*/)
{
    std::cout << message << std::endl;
}
'''
```

La fonction reçoit en argument le \*\*message\*\* de type 'string' et affiche sur la sortie standard (par défaut l'écran). Si la fonction est appelée sans argument, elle affiche par défaut "Bienvenue le monde !" :

```
'''cpp
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
    afficherBienvenue();

    return 0;
}
'''
```

Lire : [Affichage avec cout](http://tvaira.free.fr/dev/cours/affichage-cout.html) et [Saisie avec cin en C++](http://tvaira.free.fr/dev/cours/saisie-cin.html) en C++.

Thierry Vaira <tvaira@free.fr>

On obtient un joli **README.md** :

## Bienvenue

Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction `afficherBienvenue()`.

### La fonction `afficherBienvenue`

Elle est déclarée dans le fichier d'en-tête (*header*) `fonction-bienvenue.h` :

```
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

#include <string>

void afficherBienvenue(std::string message="Bienvenue le monde !");

#endif // FONCTION_BIENVENUE_H
```

Et définie dans le fichier `fonction-bienvenue.cpp` :

```
#include "fonction-bienvenue.h"
#include <iostream>

void afficherBienvenue(std::string message/*="Bienvenue le monde !"*/)
{
    std::cout << message << std::endl;
}
```

La fonction reçoit en argument le **message** de type `string` et affiche sur la sortie standard (par défaut l'écran). Si la fonction est appelée sans argument, elle affiche par défaut "Bienvenue le monde !" :

```
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
    afficherBienvenue();

    return 0;
}
```

Lire : [Affichage avec cout](#) et [Saisie avec cin en C++](#) en C++.

On peut créer et publier un dernier *tag* :

```
$ git tag -a 1.2 -m 'La version 1.2'

$ git push --tags
Décompte des objets: 1, fait.
Écriture des objets: 100% (1/1), 154 bytes | 154.00 KiB/s, fait.
Total 1 (delta 0), reused 0 (delta 0)
To github.com:tvaira/tp-git-sequence-1.git
 * [new tag]          1.2 -> 1.2
```

## 9.7. Pull Request et Révision de code

Les *Pull Requests* sont une fonctionnalité facilitant la collaboration des développeurs sur un projet.



GitHub a popularisé le principe de *Pull Request* et les autres système Git hébergés l'utilisent aussi : Bitbucket Cloud, GitLab (*Merge Request*), ...

Lien : [Collaborating with pull requests](#)

Les *Pull Requests* sont un mécanisme permettant à un développeur d'informer les membres de l'équipe qu'il a terminé un « travail » (une fonctionnalité, une version livrable, un correctif, ...) et de proposer sa contribution au dépôt central.



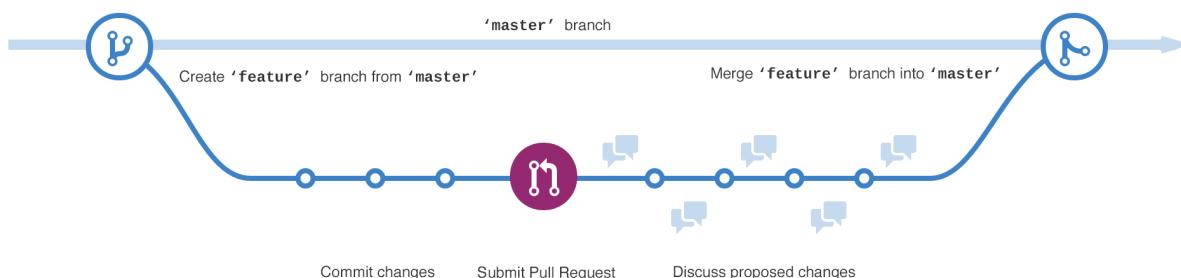
*Pull Request* peut être traduit par « Proposition de révision » (PR) : c'est-à-dire une demande de modification ou de contribution.

Principe :

Une fois que sa branche de suivi est prête, le développeur crée ou ouvre (*Open*) une *Pull Request*.

Tous les développeurs du projet seront informées du fait qu'ils doivent **réviser le code** puis le **fusionner** (*merge*) dans la branche principale (`main` ou `master`) ou dans une branche de développement (`develop`).

Pendant cette révision de code, les développeurs peuvent discuter de la fonctionnalité (commenter le code, poser des questions, ...) et proposer des adaptations de la fonctionnalité en publiant des *commits* de suivi.





Les *Pull Requests* offrent cette fonctionnalité dans une interface Web à côté des dépôts GitHub ou Bitbucket. Cette interface affiche une comparaison des changements, permet l'échange entre développeurs et fournit une méthode simple pour réaliser la fusion (*merge*) du code quand il est prêt.

Les *Pull Requests* peuvent être utilisées avec le *workflow Gitflow* (un modèle de branches strict conçu autour de la livraison du projet).

Liens :

- [Faire une pull request](#)
- [Collaborating with pull requests](#)
- [Workflow git et Gitflow](#)

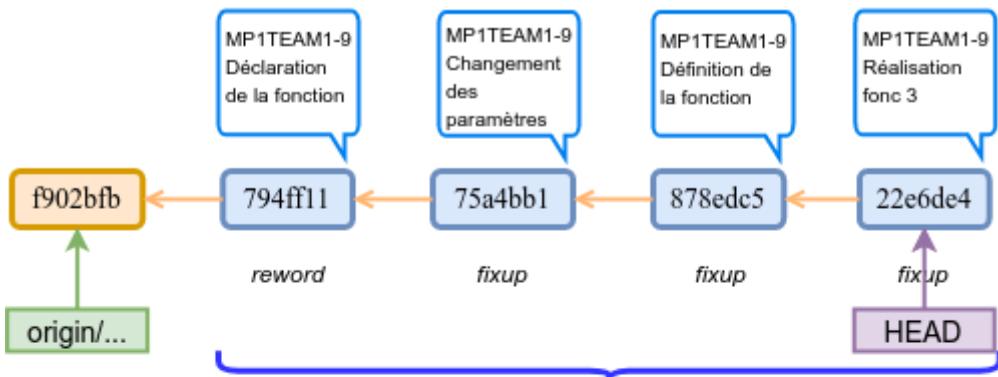
## 9.8. Travail collaboratif

### 1 . Nettoyer son historique local avant de publier

Avant de faire un `git push` sur une branche de suivi, il faut eut être nettoyer son historique local (une série de *commits* dans la branche) afin de pouvoir proposer quelque chose de propre et d'utilisable. Avant de publier la branche, il est conseillé d'effectuer une rebasage interactif avec `git rebase -i`. On a alors une totale liberté pour nettoyer, réécrire, annuler, regrouper les *commits* locaux avant de les partager (`git push`) sur le dépôt distant.

Sur la branche actuelle depuis la dernière synchronisation : `git rebase -i @{upstream}` (ou `git rebase -i origin/feature` ou `git rebase -i HEAD~n`).

Par exemple :



Avant de publier

```
git rebase -i @{upstream}
```



**i** Lorsque l'on développe seul, une branche de suivi peut servir de sauvegarde sur un dépôt distant. Dans le cadre d'un travail collaboratif, cela devient une branche de partage.

## 2 . Travailler à plusieurs sur une branche de fonctionnalité

Il est possible que le `git push` soit refusé en raison d'une branche de suivi obsolète (un travail a été poussé entre-temps) : entre la dernière synchronisation entrante (`git pull`) et le moment où on souhaite effectuer un `git push`, un autre développeur a publié des changements (des *commits*). La branche distante (par exemple `origin/feature`) est donc maintenant plus avancée que sa copie locale.

Un `git pull` provoquerait une fusion avec une divergence mais on souhaite conserver un historique linéaire au sein d'une branche : car ce n'est réalité qu'un problème de séquencement dans le travail sur la branche.

On va demander à `git pull` de faire un *rebase* au lieu d'une fusion (*merge*) en utilisant `git pull --rebase`.

**i** La commande `git rebase` permet de changer la « base » d'une branche, c'est-à-dire son *commit* d'origine. Elle rejoue une série de *commits* sur une nouvelle base.

Bonus : Supprimer toutes ses modifications et commits locaux et récupérer un dépôt distant « propre »

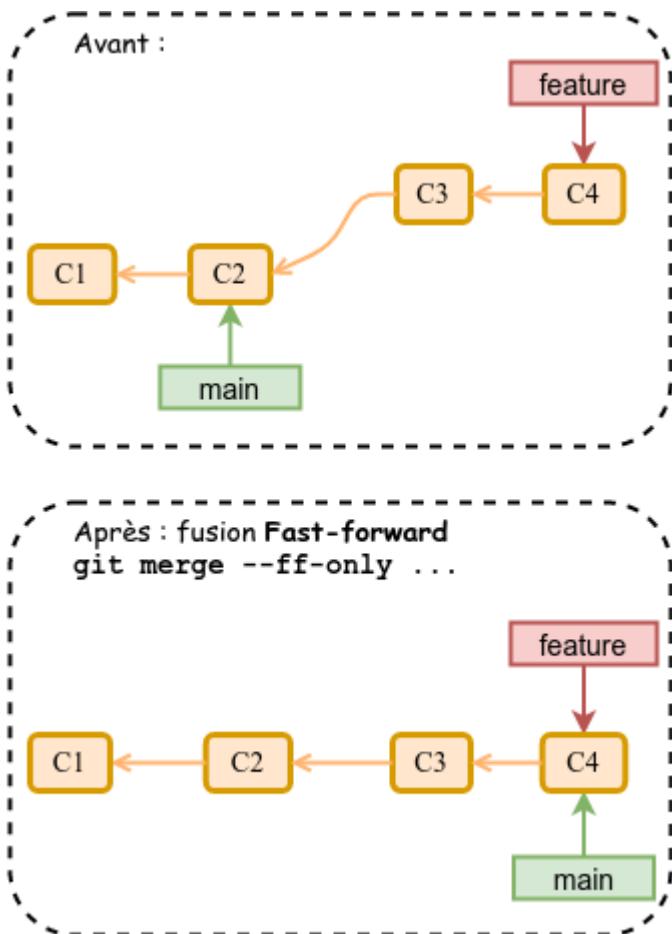
```
$ git fetch origin  
$ git reset --hard origin/main
```

# 10. La fusion

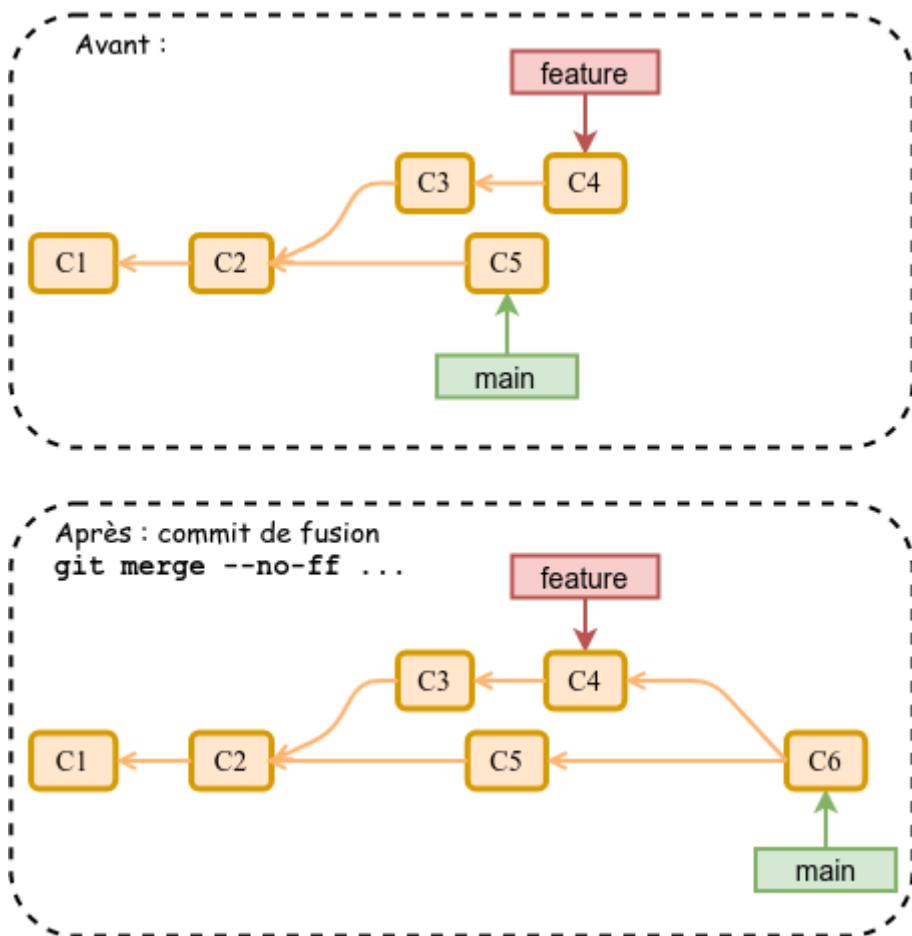
## 10.1. Stratégies de fusion

Les différentes stratégies de fusion :

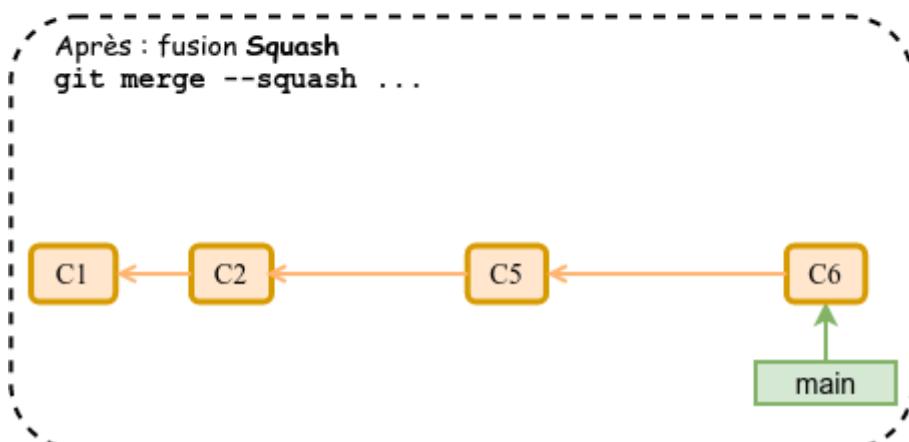
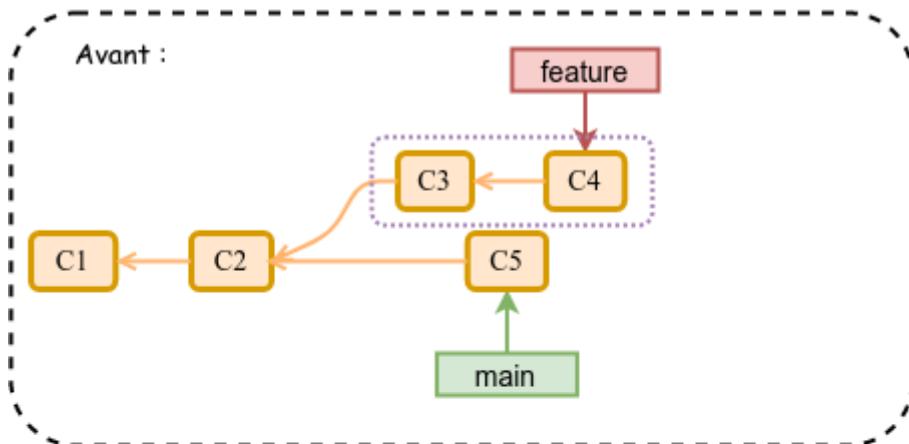
- Avance rapide (*Fast Forward*) : c'est la fusion utilisée **par défaut** par `git merge` si c'est possible. Git déplace les *commits* de la branche **feature** vers la branche destination **main** si il n'y a pas eu de nouveaux *commits* sur cette branche. En réalité, Git déplace simplement le pointeur vers l'avant. On peut réaliser cette fusion avec l'option `--ff-only`.



- *Commit de fusion* : lorsque l'historique de développement a divergé, `git merge` réalise une fusion à trois sources (*three-way merge*) en utilisant les deux *commits* au sommet des deux branches (C4 et C5) ainsi que leur plus proche ancêtre commun (C2) pour créer un nouveau *commit* (C6). On peut réaliser cette fusion avec l'option `--no-ff`.



- **Squash** : on obtient un nouveau *commit* qui regroupe tous les *commits* de la branche. Pour réaliser cette fusion, il faut ajouter l'option **--squash**.



Dans GitHub :

Merge pull request ▾ or view command line

✓ **Create a merge commit**  
All commits from this branch will be added to the base branch via a merge commit.

**Squash and merge**  
The 1 commit from this branch will be added to the base branch.

**Rebase and merge**  
The 1 commit from this branch will be rebased and added to the base branch.

Dans Bitbucket :

Merge strategy

Merge commit

Merge commit  
git merge --no-ff

Squash  
git merge --squash

Fast forward  
git merge --ff-only

## 10.2. Le conflit de fusion

Il est possible qu'une fusion (*merge*) ne puisse pas être réalisée automatiquement par Git. Cela arrive lorsqu'une même partie d'un fichier a été modifiée dans deux branches distinctes.



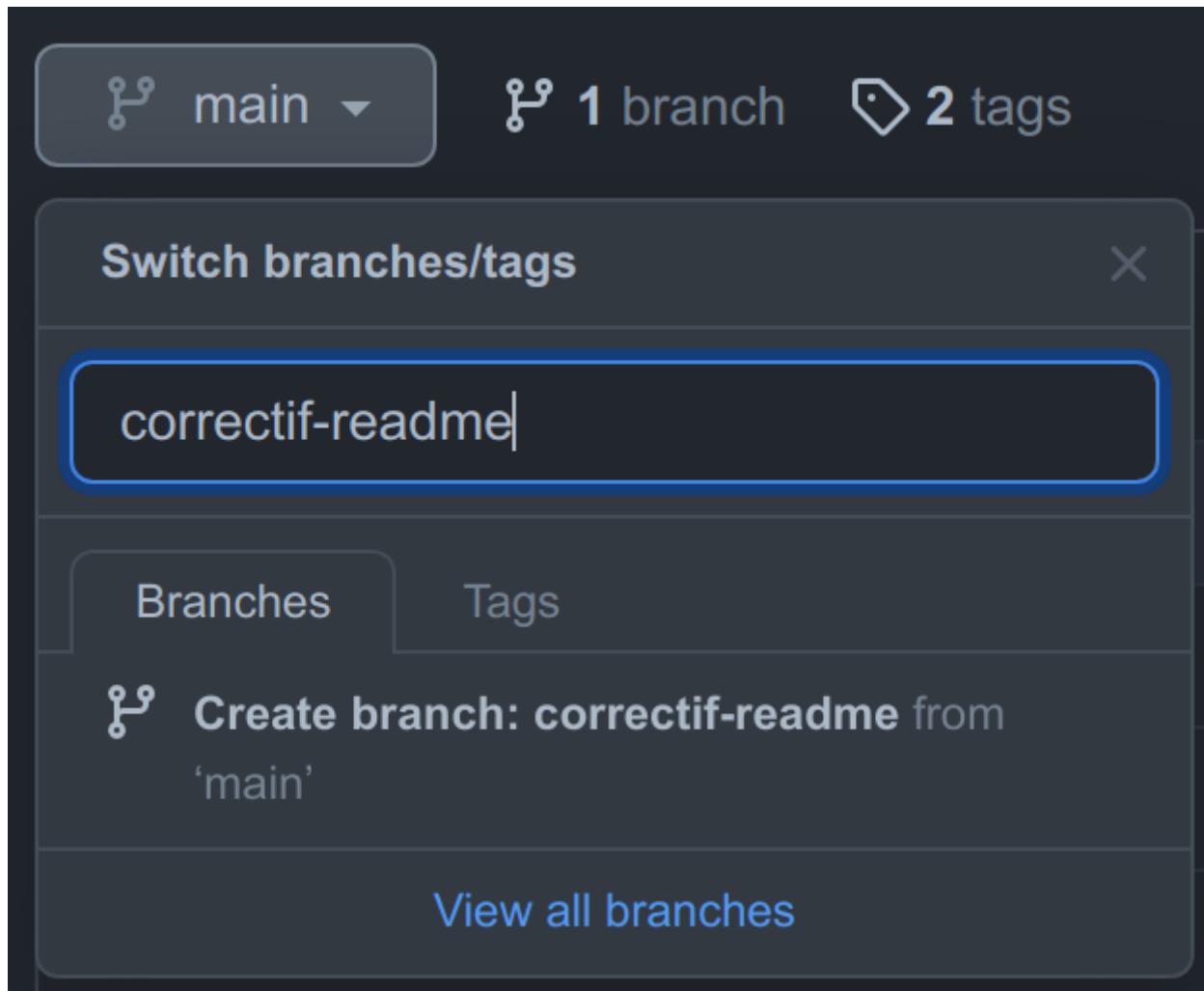
Cette situation peut se produire dans le cadre d'un travail collaboratif mais, rarement en Développeur seul.

On s'aperçoit que le fichier `README.md` n'a pas été modifié (nom de fonction incorrect) lors de la modification de la fonction (un oubli !) :

```
$ cat README.md  
# Bienvenue
```

Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction  
`'afficherBienvenue()'`.

Il faut donc faire un correctif. Pour cela, on va créer une branche thématique (dans [GitHub](#) pour changer) :



La branche `correctif-readme` est créée sur le dépôt distant mais elle n'est pas encore disponible sur le dépôt local :

```
$ git ls-remote  
From git@github.com:tvaira/tp-git-sequence-1.git  
ce00d3449aa40aa44d51effcc66c796eb9b2ed20    HEAD  
ce00d3449aa40aa44d51effcc66c796eb9b2ed20    refs/heads/correctif-readme  
ce00d3449aa40aa44d51effcc66c796eb9b2ed20    refs/heads/main  
...  
  
$ git branch -vv  
* main ce00d34 [origin/main] Modification afficherBienvenue en afficherMessage
```

Il faut donc récupérer les informations du dépôt distant et les rapatrier dans le dépôt local :

```
$ git fetch  
Depuis github.com:tvaira/tp-git-sequence-1  
* [nouvelle branche] correctif-readme -> origin/correctif-readme
```



La commande `git fetch` ne modifie pas le répertoire de travail, seulement le dépôt.

On bascule sur la branche `correctif-readme` qui sera automatiquement défini comme une branche de suivi :

```
$ git checkout correctif-readme
La branche 'correctif-readme' est paramétrée pour suivre la branche distante
'correctif-readme' depuis 'origin'.
Basculement sur la nouvelle branche 'correctif-readme'

$ git branch -vv
* correctif-readme ce00d34 [origin/correctif-readme] Modification afficherBienvenue en
afficherMessage
  main           ce00d34 [origin/main] Modification afficherBienvenue en
afficherMessage

$ git log --oneline
ce00d34 (HEAD -> correctif-readme, tag: 1.1, origin/main, origin/correctif-readme,
main) Modification afficherBienvenue en afficherMessage
c479e51 Renommage README.md
470794d Modification du fichier README
...
...
```

On modifie le fichier `README.md` et on valide le changement :

```
$ vim README.md
# Bienvenue

Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction
'afficherMessage()'.

$ git add README.md
$ git commit -m "Modification README.md"
[correctif-readme 3d4fa0d] Modification README.md
 1 file changed, 1 insertion(+), 2 deletions(-)
```

## Vérification :

```
$ git branch -vv
* correctif-readme 3d4fa0d [origin/correctif-readme: en avance de 1] Modification
 README.md
  main           ce00d34 [origin/main] Modification afficherBienvenue en
 afficherMessage

$ git log --oneline
3d4fa0d (HEAD -> correctif-readme) Modification README.md
ce00d34 (tag: 1.1, origin/main, origin/correctif-readme, main) Modification
afficherBienvenue en afficherMessage
c479e51 Renommage README.md
470794d Modification du fichier README
...
```

Maintenant, on va basculer sur la branche principale pour créer une nouvelle branche thématique **modification-fonction** pour modifier la fonction (et remettre un peu d'ordre dans le code !) :

## Basculement sur la branche principale :

```
$ git checkout main
Basculement sur la branche 'main'
Votre branche est à jour avec 'origin/main'.

$ git branch -vv
 correctif-readme 3d4fa0d [origin/correctif-readme: en avance de 1] Modification
 README.md
 * main           ce00d34 [origin/main] Modification afficherBienvenue en
 afficherMessage
```

## Création d'une branche thématique :

```
$ git branch modification-fonction

$ git branch -vv
 correctif-readme      3d4fa0d [origin/correctif-readme: en avance de 1] Modification
 README.md
 * main                 ce00d34 [origin/main] Modification afficherBienvenue en
 afficherMessage
 modification-fonction ce00d34 Modification afficherBienvenue en afficherMessage
```

Basculement sur la branche thématique :

```
$ git checkout modification-fonction
Basculement sur la branche 'modification-fonction'

$ git branch -vv
  correctif-readme      3d4fa0d [origin/correctif-readme: en avance de 1] Modification
 README.md
    main                  ce00d34 [origin/main] Modification afficherBienvenue en
 afficherMessage
 * modification-fonction ce00d34 Modification afficherBienvenue en afficherMessage
```

On modifie le projet :

```
$ vim fonction-bienvenue.h
```

```
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

#include <string>

void afficherBienvenue(std::string message="Bienvenue le monde !");

#endif // FONCTION_BIENVENUE_H
```

```
$ vim fonction-bienvenue.cpp
```

```
#include "fonction-bienvenue.h"
#include <iostream>

void afficherBienvenue(std::string message/*="Bienvenue le monde !"*/)
{
    std::cout << message << std::endl;
}
```

```
$ vim bienvenue.cpp
```

```
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
    afficherBienvenue();

    return 0;
}
```

```
$ vim README.md
```

```
# Bienvenue
```

Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction 'afficherBienvenue()'.

Thierry Vaira <tvaira@free.fr>

On teste :

```
$ make rebuild
Fabrication du programme : bienvenue
rm -f *.o
g++ -c -Wall -std=c++11 bienvenue.cpp
g++ -c -Wall -std=c++11 fonction-bienvenue.cpp
g++ -o bienvenue bienvenue.o fonction-bienvenue.o

$ ./bienvenue
Bienvenue le monde !
```

On ajoute les fichiers dans l'index :

```
$ git add fonction-bienvenue.h
$ git add fonction-bienvenue.cpp
$ git add bienvenue.cpp
$ git add REAME.md
```

Et on valide les changements :

```
$ git commit -m "Modification de la fonction afficherBienvenue() qui affiche le
message "Bienvenue le monde !" par défaut"
```

## Vérification :

```
$ git log --oneline
f909007 (HEAD -> modification-fonction) Modification de la fonction
afficherBienvenue() qui affiche le message "Bienvenue le monde !" par défaut
ce00d34 (tag: 1.1, origin/main, origin/correctif-readme, main) Modification
afficherBienvenue en afficherMessage
c479e51 Renommage README.md
470794d Modification du fichier README
```

On rebascule la branche principale et on peut voir qu'il y a maintenant une divergence dans l'historique :

```
$ git log --graph --decorate --oneline --all
* f909007 (modification-fonction) Modification de la fonction afficherBienvenue() qui
  affiche le message "Bienvenue le monde !" par défaut
| * 3d4fa0d (correctif-readme) Modification README.md
|/
* ce00d34 (HEAD -> main, tag: 1.1, origin/main, origin/correctif-readme) Modification
  afficherBienvenue en afficherMessage
* c479e51 Renommage README.md
* 470794d Modification du fichier README
...
...
```

Graph	Description	Date	Author
●	[P] modification-fonction Modification de la fonction afficherBienvenue() qui affiche le message "Bienvenu..." 16 Aug 2021 15:51 tvaira f9090073		
●	[P] correctif-readme Modification README.md 16 Aug 2021 15:34 tvaira 3d4fa0d0		
○ [P] main   origin [P] origin/correctif-readme [C] 1.1	Modification afficherBienvenue en afficherMess... 15 Aug 2021 11:20 tvaira ce00d344		
	Renommage README.md 11 Aug 2021 17:14 Thierry VAIRA c479e51a		
	Modification du fichier README 11 Aug 2021 17:04 tvaira 470794d3		

On fusionne la branche **correctif-readme** dans la branche principale :

```
$ git merge correctif-readme
Mise à jour ce00d34..3d4fa0d
Fast-forward
 README.md | 3 +--
 1 file changed, 1 insertion(+), 2 deletions(-)
```

```
$ git log --graph --decorate --oneline --all
* f909007 (modification-fonction) Modification de la fonction afficherBienvenue() qui
affiche le message "Bienvenue le monde !" par défaut
| * 3d4fa0d (HEAD -> main, correctif-readme) Modification README.md
|/
* ce00d34 (tag: 1.1, origin/main, origin/correctif-readme) Modification
afficherBienvenue en afficherMessage
* c479e51 Renommage README.md
* 470794d Modification du fichier README
...
...
```

On essaye maintenant de fusionner la branche **modification-fonction** dans la branche principale :

```
$ git merge modification-fonction
Fusion automatique de README.md
CONFLIT (contenu) : Conflit de fusion dans README.md
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

*Très utile :*

```
$ git status
Sur la branche main
Votre branche est en avance sur 'origin/main' de 1 commit.
(utilisez "git push" pour publier vos commits locaux)
```

Vous avez des chemins non fusionnés.  
(réglez les conflits puis lancez "git commit")  
(utilisez "git merge --abort" pour annuler la fusion)

Modifications qui seront validées :

```
modifié :      bienvenue.cpp
modifié :      fonction-bienvenue.cpp
modifié :      fonction-bienvenue.h
```

Chemins non fusionnés :  
(utilisez "git add <fichier>..." pour marquer comme résolu)

```
modifié des deux côtés : README.md
```

On va commencer par voir le conflit :

```
$ cat README.md
# Bienvenue

<<<<< HEAD
Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction
`afficherMessage()`.

=====
Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction
`afficherBienvenue()`.

>>>> modification-fonction

Thierry Vaira <tvaira@free.fr>
```



Lorsque Git rencontre un conflit au cours d'une fusion, il l'indique dans les fichiers avec des délimiteurs (`<<<<<`, `=====` et `>>>>>`) qui marquent les deux côtés du conflit.

Pour résoudre le conflit, il faut choisir une partie ou l'autre ou bien fusionner les deux contenus "à la main" :

```
$ vim README.md
```

```
# Bienvenue
```

Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction  
`afficherBienvenue()`.

```
Thierry Vaira <tvaira@free.fr>
```

On peut ensuite terminer la fusion en suivant les indications de `git status` :

```
$ git add README.md

$ git status
Sur la branche main
Votre branche est en avance sur 'origin/main' de 1 commit.
  (utilisez "git push" pour publier vos commits locaux)

Tous les conflits sont réglés mais la fusion n'est pas terminée.
  (utilisez "git commit" pour terminer la fusion)
```

Modifications qui seront validées :

```
modifié : README.md
modifié : bienvenue.cpp
modifié : fonction-bienvenue.cpp
modifié : fonction-bienvenue.h
```

```
$ git commit
[main 3cf9129] Merge branch 'modification-fonction' into main
```

```
$ git log --graph --decorate --oneline --all
* 3cf9129 (HEAD -> main) Merge branch 'modification-fonction' into main
|\ \
| * f909007 (modification-fonction) Modification de la fonction afficherBienvenue()
| qui affiche le message "Bienvenue le monde !" par défaut
* | 3d4fa0d (correctif-readme) Modification README.md
|/
* ce00d34 (tag: 1.1, origin/main, origin/correctif-readme) Modification
afficherBienvenue en afficherMessage
* c479e51 Renommage README.md
* 470794d Modification du fichier README
...
...
```

Graph	Description	Date	Author	
	o main Merge branch 'modification-fonction' into main	16 Aug 2021 16:18	tvaira	3cf9129d
	* modification-fonction Modification de la fonction afficherBienvenue() qui affiche le message "Bienvenu..."	16 Aug 2021 15:51	tvaira	f9090073
	* correctif-readme Modification README.md	16 Aug 2021 15:34	tvaira	3d4fa0d0
	* origin/correctif-readme   origin/main   1.1 Modification afficherBienvenue en afficherMessage	15 Aug 2021 11:20	tvaira	ce00d344
	Renommage README.md	11 Aug 2021 17:14	Thierry VAIRA	c479e51a
	Modification du fichier README	11 Aug 2021 17:04	tvaira	470794d3



Attention, le dépôt distant n'est plus synchronisé :

```
$ git branch -vv
  correctif-readme      3d4fa0d [origin/correctif-readme: en avance de 1] Modification
 README.md
* main                  3cf9129 [origin/main: en avance de 3] Merge branch
 'modification-fonction' into main
   modification-fonction f909007 Modification de la fonction afficherBienvenue() qui
   affiche le message "Bienvenue le monde !" par défaut

$ git status
Sur la branche main
Votre branche est en avance sur 'origin/main' de 3 commits.
  (utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
```

Donc, on "pousse" (*push*) vers le dépôt distant :

```
$ git push
Décompte des objets: 12, fait.
Delta compression using up to 12 threads.
Compression des objets: 100% (12/12), fait.
Écriture des objets: 100% (12/12), 1.25 KiB | 425.00 KiB/s, fait.
Total 12 (delta 7), reused 0 (delta 0)
remote: Resolving deltas: 100% (7/7), completed with 3 local objects.
To github.com:tvaira/tp-git-sequence-1.git
 ce00d34..3cf9129  main -> main
```

**README.md**

## Bienvenue

Programme C++ qui affiche le message "Bienvenue le monde !" en utilisant la fonction `afficherBienvenue()`.

Thierry Vaira [tvaira@free.fr](mailto:tvaira@free.fr)

On peut finir par un nettoyage des branches thématiques qui ne servent plus :

```
$ git branch -D modification-fonction
Branche modification-fonction supprimée (précédemment f909007).

$ git branch -D correctif-readme
Branche correctif-readme supprimée (précédemment 3d4fa0d).

$ git push origin --delete correctif-readme
To github.com:tvaira/tp-git-sequence-1.git
 - [deleted]          correctif-readme

$ git branch -vv
* main 3cf9129 [origin/main] Merge branch 'modification-fonction' into main
```

## 11. Workflow git et Gitflow



Un **workflow** (flux de travaux) est la représentation d'une suite de tâches ou d'opérations effectuées par une personne, un groupe de personnes, un organisme, etc.

Un **workflow git** est une méthode, un processus de travail, une recette ou une recommandation sur la façon d'utiliser **git** pour accomplir un travail de manière cohérente et productive.

Il n'existe pas de processus standardisé sur la façon d'interagir avec **git**. Il est important de

s'assurer que l'équipe de projet est d'accord sur la façon dont le flux de modifications sera appliqué. Un *workflow git* doit donc être défini.

Il existe plusieurs *workflows git* connus qui peuvent être utilisés :

- *workflow centralisé*
- *workflow de branche de fonctionnalité*
- *workflow Gitflow*

Lien : [Comparaison des workflow git](#)

Le *workflow Gitflow* définit un modèle de branchement strict conçu autour de la version du projet. Ce *workflow* n'ajoute pas de nouveaux concepts ou commandes. Gitflow permet de gérer les bugs (*issues*), les nouvelles fonctionnalités (*features*) et les versions (*releases*) en attribuant des rôles très spécifiques à différentes branches et définit comment et quand elles doivent interagir.



Les rôles des branches sont les suivants :

- pour les branches permanentes :
  - La branche `master` stocke l'historique des versions officielles. Tous les *commits* de cette branche sont étiquetés avec un numéro de version (*tags*).
  - La branche `develop` est créée à partir de la branche `master`. Elle sert de branche d'intégration pour les fonctionnalités. Cette branche contiendra l'historique complet du projet.
- pour les branches temporaires :
  - Les branches `features-xxxx` permettent de travailler sur des nouvelles fonctionnalités. Elles sont créées directement à partir de la branche `develop` et une fois le travail fini, fusionnées vers la branche `develop`.
  - Les branches `release-xxxx` permettent de travailler sur une livraison (généralement des tâches dédiées à la documentation). On les crée à partir de `develop` puis on les fusionne dans `master` en leur attribuant un numéro de version (*tag*).

- Les branches **hotfix-xxxx** permettent de publier rapidement (*hot*) une correction (*fix*) depuis la branche **master**. Ces branches seront ensuite fusionnées vers la branche **master** et **develop**.

### *Les extensions git-flow*

Il existe des extensions **git-flow** à **git** pour intégrer le *workflow* Gitflow.

```
$ sudo apt install git-flow
```

```
$ git flow help
usage: git flow <subcommand>
```

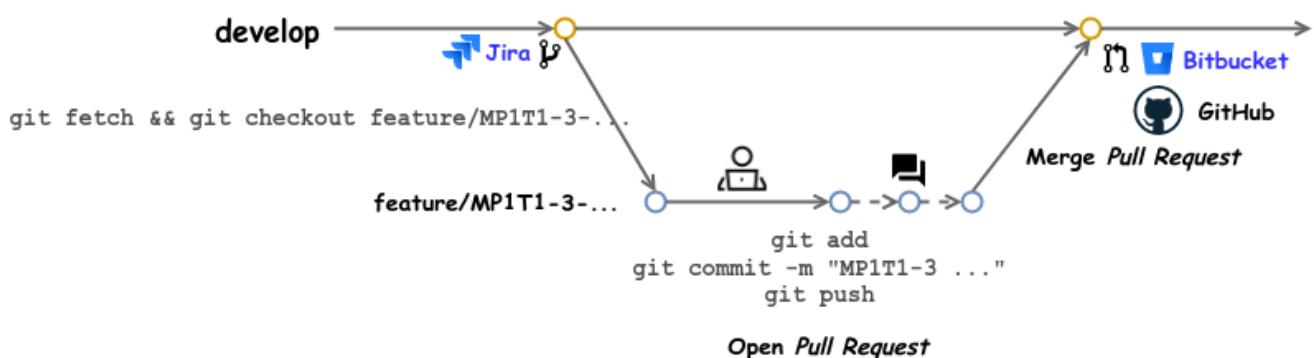
Available subcommands are:

**i** **init** Initialize a new git repo with support for the branching model.  
**feature** Manage your feature branches.  
**bugfix** Manage your bugfix branches.  
**release** Manage your release branches.  
**hotfix** Manage your hotfix branches.  
**support** Manage your support branches.  
**version** Shows version information.  
**config** Manage your git-flow configuration.  
**log** Show log deviating from base branch.

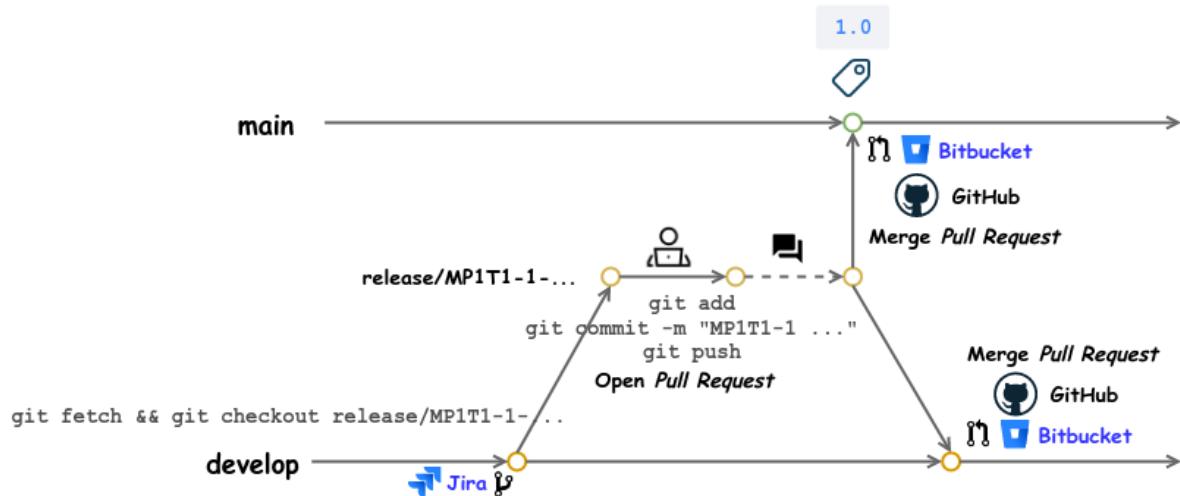
Try 'git flow <subcommand> help' for details.

**!** En projet BTS SN, les branches (*feature*, *release* et *hotfix*) seront créées dans Jira à partir d'un ticket. Les fusions seront réalisées lors d'une revue de code en utilisant les *Pull Requests* dans GitHub ou Bitbucket.

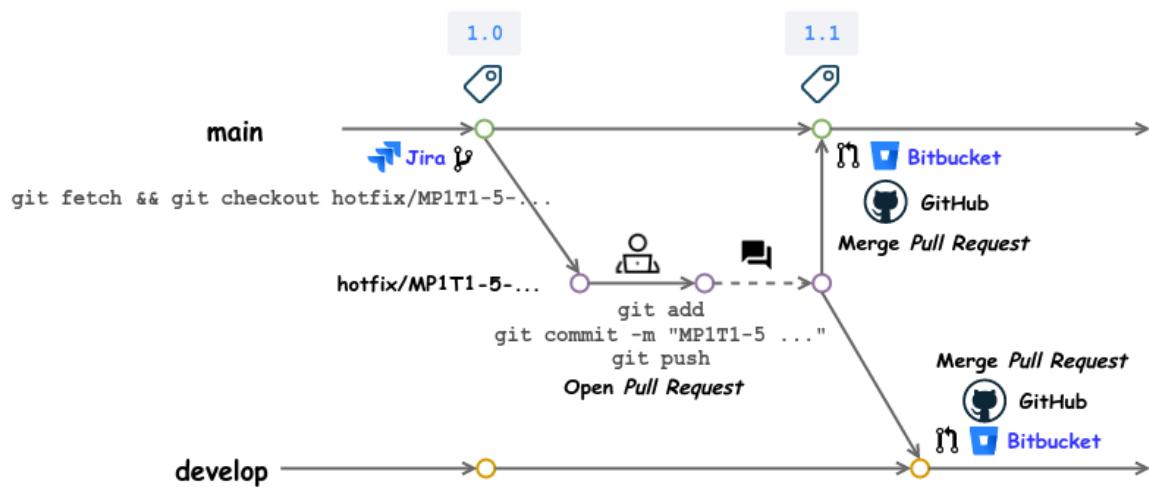
Réalisation d'une fonctionnalité :



Réalisation d'une *release* :



Correction d'un bug :



Une branche représente une ligne de développement indépendante. Lorsqu'elle désigne un travail bien identifié du projet (une fonctionnalité, une *release* ou un correctif), il est préférable (obligatoire) que cela reste visible dans le graphe d'historique, même lorsque la branche est supprimée. Pour éviter que Git utilise par défaut une avance rapide (*Fast Forward*) si c'est possible, il faudra réaliser un *commit* de fusion avec l'option `--no-ff`.



## 12. Environnement de développement intégré (EDI ou IDE)

La plupart des environnements de développement intègre Git ou propose des extensions pour le faire.

### 12.1. Visual Studio Code

[Visual Studio Code](#) (un des IDE les plus utilisés actuellement) intègre la gestion du contrôle de source (SCM) et inclut par défaut la prise en charge de Git.

## Installation de Visual Studio Code :

- Download
- Setup
- Getting Started

The screenshot shows the Visual Studio Code interface with the following details:

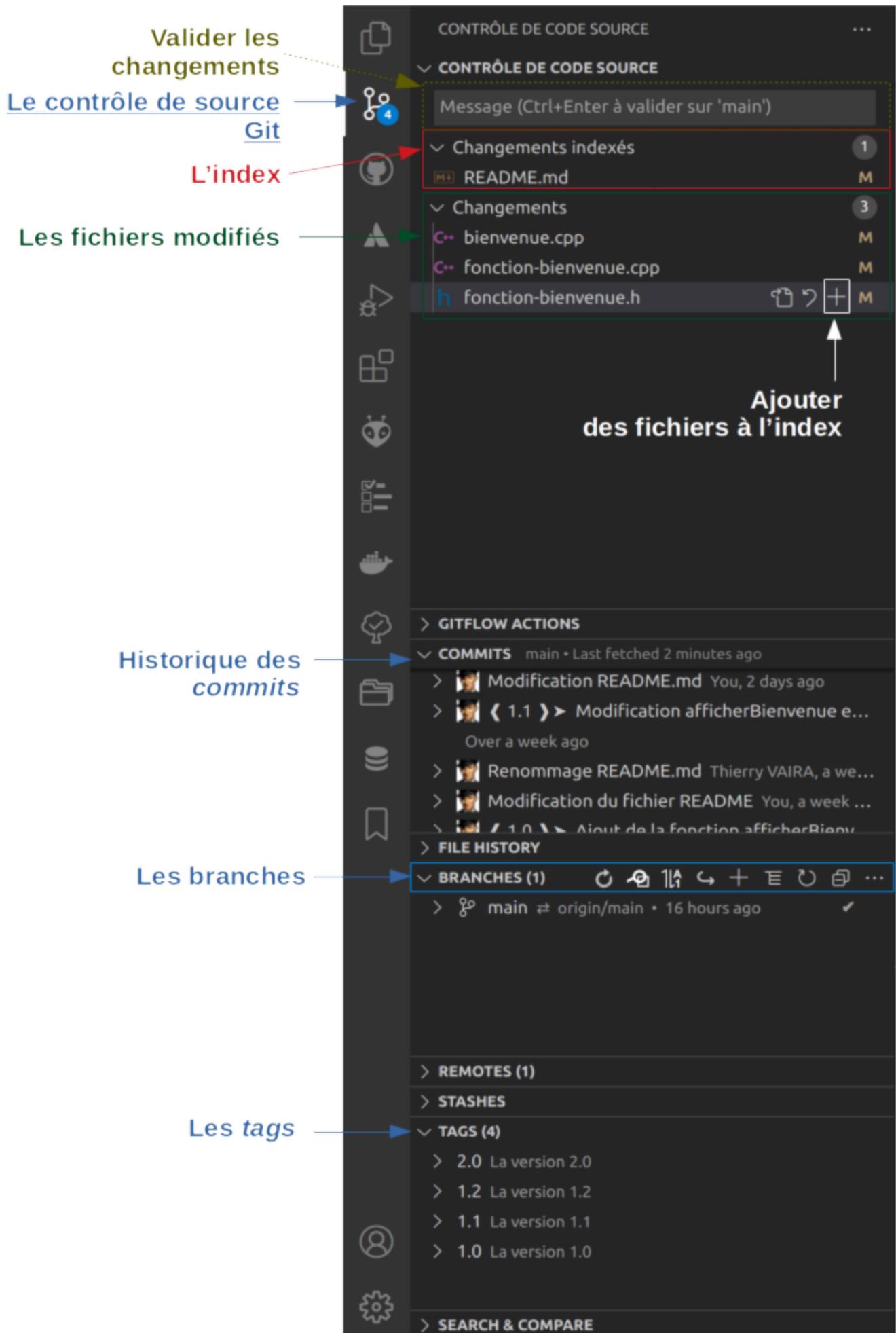
- File Explorer:** Shows the project structure under "TP-GIT-SEQUENCE-1". It includes a ".gitignore" file, "bienvenue", "bienvenue.o", "fonction-bienvenue.cpp", "fonction-bienvenue.h", "hello.cpp", "Makefile", and "README.md".
- Editor:** The main editor window displays the content of "bienvenue.cpp". The code is a simple C++ program that prints "Bienvenue" multiple times based on command-line arguments.
- Terminal:** The bottom terminal window shows the build process:

```
1 # Bienvenue
2
3 Programme C++ qui affiche un ou plusieurs fois un message de bienvenue à partir de la ligne de commande :
4
5 ````sh
6 $ make rebuild
7 Fabrication du programme : bienvenue
8 rm -f *.o
9 g++ -c -Wall -std=c++11 bienvenue.cpp
10 g++ -c -Wall -std=c++11 fonction-bienvenue.cpp
11 g++ -o bienvenue bienvenue.o fonction-bienvenue.o
12
13 $ ./bienvenue
14 Bienvenue le monde !
15
16 $ ./bienvenue Bienvenue
17 Bienvenue
18
19 $ ./bienvenue Bienvenue 2
20 Bienvenue
21 Bienvenue
22
23 $ ./bienvenue "Bonjour le monde" 3
24 Bonjour le monde You, 16 hours ago • TP Séquence n°2
25 Bonjour le monde
26 Bonjour le monde
27 ````
```
- Bottom Status Bar:** Shows the current file path ("README.md - tp-git-sequence-1 - Visual Studio Code"), the date and time ("08:37:31 tv@sedatech tp-git-sequence-1 ±|main x|→"), and various status indicators like "PROBLÈMES", "TERMINAL", "COMMENTS", "OUTPUT", and "CONSOLE DE DÉBOUTAGE".

## Liens :

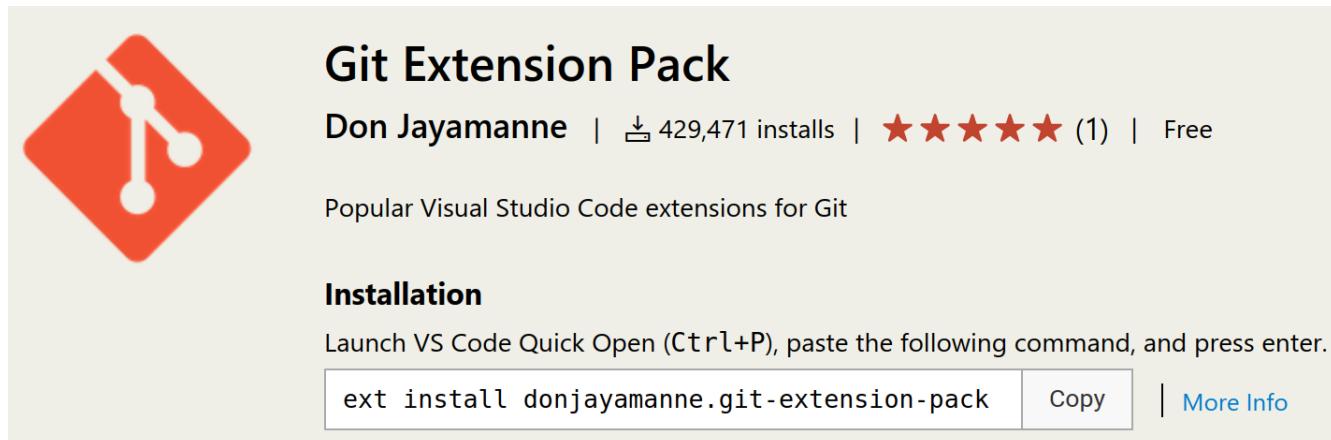
- <https://code.visualstudio.com/docs/editor/versioncontrol>
- <https://code.visualstudio.com/docs/editor/github>

## Le contrôle de source (SCM) :



Il existe de nombreuses extensions pour faciliter l'utilisation de Git dont [Git Extension Pack](#) qui comprend :

- [Git History](#)
- [Project Manager](#)
- [GitLens](#)
- [gitignore](#)
- [Open in GitHub / Bitbucket / VisualStudio.com](#)



The screenshot shows the Visual Studio Marketplace page for the "Git Extension Pack" extension. At the top, there's a large orange diamond icon containing a white git branch icon. To the right of the icon, the title "Git Extension Pack" is displayed in bold black font. Below the title, the author "Don Jayamanne" is listed, along with the number of installs "429,471" and a 5-star rating "(1)". A "Free" badge is also present. A subtitle "Popular Visual Studio Code extensions for Git" follows. Underneath, a section titled "Installation" provides instructions: "Launch VS Code Quick Open (Ctrl+P), paste the following command, and press enter." Below these instructions is a code input field containing the command "ext install donjayamanne.git-extension-pack". To the right of the input field are two buttons: "Copy" and "More Info".

Et quelques autres :

- [GitHub Pull Requests and Issues](#)
- [GitHub Repositories](#)
- [Git Graph](#)
- [Git Project Manager](#)
- [Git Blame](#)
- [Gitflow Actions Sidebar](#)
- [gitflow](#)
- [Jira and Bitbucket \(Atlassian Labs\)](#)

## 13. Les outils graphiques

Il existe de nombreuses interfaces graphiques permettant de gérer des projets Git.

En standard :

- une interface web avec [GitWeb](#)
- une interface de visualisation détaillée et graphique avec [gitk](#) :

The screenshot shows the gitk application window titled "tp-git-sequence-1: Tous les fichiers - gitk". The main pane displays a commit history with the following entries:

- 1.1 main remotes/origin/main Modification afficherBienvenue() tvaira <tvaira@free.fr> 2021-08-15 11:20:06
- Renomme README.md Thierry VAIRA <tvaira@free.fr> 2021-08-11 17:14:46
- Modification du fichier README tvaira <tvaira@free.fr> 2021-08-11 17:04:31
- Ajout de la fonction afficherBienvenue() tvaira <tvaira@free.fr> 2021-08-11 14:13:15
- Ajout README tvaira <tvaira@free.fr> 2021-07-31 11:55:48

Below the commit history, the "Id SHA1" is listed as `ce00d3449aa40aa44d51effcc66c796eb9b2ed20`. The "Recherche" field contains "commit contient :". The "Exact" and "Tous les champs" buttons are visible.

The right side of the interface shows a tree view with nodes like "Patch", "Arbre", "Commentaires", "bienvenue.cpp", "fonction-bienvenue.cpp", and "fonction-bienvenue.h".

The bottom pane shows a diff view for the file `bienvenue.cpp`:

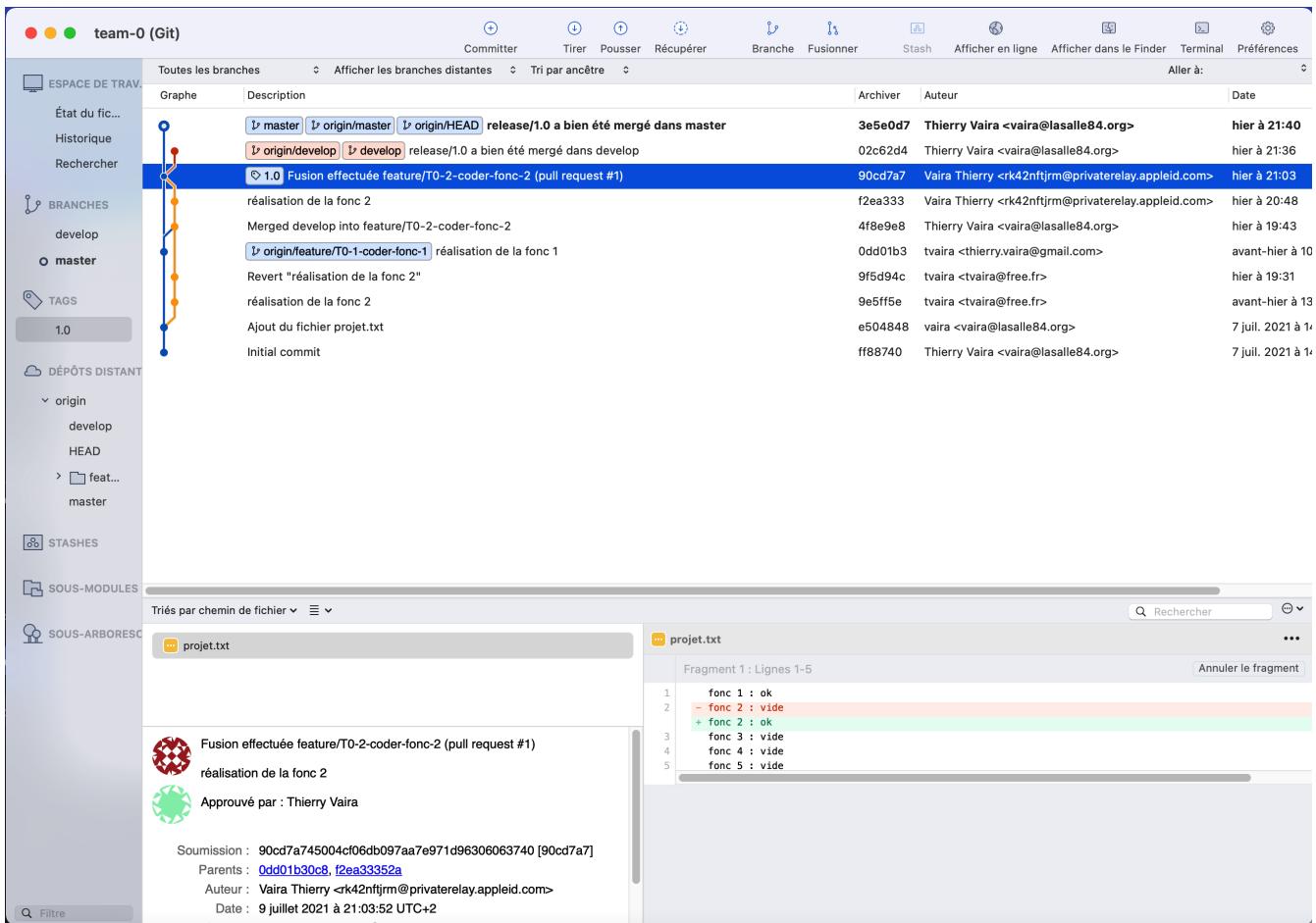
```

index bd5dd64..14c04e1 100644
@@ -4,7 +4,7 @@
int main()
{
- afficherBienvenue();
+ afficherMessage("Bienvenue le monde !");
return 0;
}

```

Il existe également de nombreuses autres applications :

- [qgit](#) propose des outils supplémentaires par rapport à [gitk](#) ;
- [Giggle](#) : une interface en GTK+ ;
- [GitExtensions](#) : un client Git graphique pour Windows © ;
- [TortoiseGit](#) : logiciel libre pour Windows reprenant les éléments d'interface de [TortoiseSVN](#) (un classique) ;
- ...
- [SourceTree](#) : un logiciel propriétaire gratuit pour Windows © et macOS © édité par Atlassian ;



- **GitEye** : un client graphique pour Windows ©, macOS © et Linux

```
$ cd ~/Téléchargements/
$ wget -c https://www.collab.net/sites/default/files/downloads/GitEye-2.2.0-linux.x86_64.zip

$ mkdir /tmp/GitEye
$ unzip -d /tmp/GitEye ~/Téléchargements/GitEye-2.2.0-linux.x86_64.zip

$ sudo chown -R root:root /tmp/GitEye
$ sudo mv /tmp/GitEye /opt/GitEye
$ sudo ln -s /opt/GitEye/GitEye /usr/local/bin/GitEye
$ GitEye
```

CollabNet GitEye

**Git Repositories** | CollabNet Sites

**team-0 [master]** - /home/iris/Documents/jira/team-0/.git

**Branches**

- Local
- Remote Tracking
  - origin/develop 90cd7a7 Fusion effectuée feature/T0-2-coder-func-2 (pull request #1)
  - origin/feature/T0-1-coder-func-1 0dd01b3 réalisation de la fonic 1
  - origin/feature/T0-2-coder-func-2 e504848 Ajout du fichier projet.txt
  - origin/master 90cd7a7 Fusion effectuée feature/T0-2-coder-func-2 (pull request #1)
  - origin/release/1.0 90cd7a7 Fusion effectuée feature/T0-2-coder-func-2 (pull request #1)
- Tags
- References
- Remotes
  - origin
    - https://tvaira@bitbucket.org/btssn-avignon/team-0.git
    - https://tvaira@bitbucket.org/btssn-avignon/team-0.git

**Working Tree** : /home/iris/Documents/jira/team-0 [team-0 master]

**git**

- .gitignore
- .project
- project.txt

1 item selected

**Dashboard** | **Git Files** | **History** | **Task List** | **Builds**

Repository: team-0

ID	Message	Author	Authored Date	Committer	Committed Date
90cd7a7	1.0 develop master release/1.0 origin/HEAD origin/develop origin/master origin/release/1.0 HEAD	Vaira Thierry	12 hours ago	Thierry Vaira	12 hours ago
f2ea333	réalisation de la fonic 2	Vaira Thierry	12 hours ago	Vaira Thierry	12 hours ago
4f5d9e8	Merged develop into feature/T0-2-coder-func-2	Thierry Vaira	13 hours ago	Thierry Vaira	13 hours ago
9f5d94c	Revert "réalisation de la fonic 2"	tvaira	13 hours ago	tvaira	13 hours ago
9ef5ff5e	réalisation de la fonic 2	tvaira	2 days ago	tvaira	2 days ago
0dd01b3	origin/feature/... réalisation de la fonic 1	tvaira	2 days ago	tvaira	2 days ago
e504848	origin/feature/... Ajout du fichier projet.txt	vaira	3 days ago	vaira	3 days ago
fb88740	Initial commit	Thierry Vaira	3 days ago	Thierry Vaira	3 days ago

commit 90cd7a7a504cfe60b07aa7e371d9e36e095740  
Author: Thierry Vaira <tvaira@asuslive84.org> 2021-07-09 21:03:52  
Committer: Thierry Vaira <tvaira@asuslive84.org> 2021-07-09 21:03:52  
Parent: 8dd91b30c3b0e819150193c49edcd79fb232 (réalisation de la fonic 1)  
Parent: f2ea3332ac3786cb05f982116a01bd432cf09 (réalisation de la fonic 2)  
Branches: develop, master, release/1.0, origin/develop, origin/master, origin/release/1.0  
Tags: 1.0

Fusion effectuée feature/T0-2-coder-func-2 (pull request #1)  
réalisation de la fonic 2  
Approuvé par : Thierry Vaira

Thierry Vaira - <tvaira@free.fr> - version v0.2 - 23/08/2021 - [tvaira.free.fr](http://tvaira.free.fr)