

# **Gestion de développement de logiciels et gestion de versions**

Introduction a Git

BRULIN Pierre-Yves

2025-10-03

## Table des matières

<b>1</b>	<b>Problématiques du développement logiciel</b>	<b>3</b>
1.1	Contexte . . . . .	3
1.2	Situations simples à éviter . . . . .	3
1.3	Les questions qui se posent . . . . .	3
1.3.1	De votre point de vue d'étudiant . . . . .	4
1.4	L'intérêt d'un gestionnaire de versions . . . . .	4
1.4.1	Les objectifs de la gestion des versions . . . . .	4
1.5	Trois architectures possibles de versionnement . . . . .	4
1.6	VCS Local . . . . .	5
1.7	VCS Centralisé . . . . .	5
1.8	VCS Décentralisé . . . . .	6
<b>2</b>	<b>Avantages de Git</b>	<b>6</b>
2.1	Focus sur Git . . . . .	6
2.1.1	Pourquoi décentraliser le travail sur le noyau Linux? . . . . .	7
2.2	Intégration de git dans vos IDE . . . . .	7
2.3	Installation de Git . . . . .	8
2.4	Installation de Git (Windows) . . . . .	9
2.5	Pourquoi Git? Utilisation par la pratique . . . . .	9
2.5.1	Questions . . . . .	9
2.6	Rapidité de git . . . . .	10
2.6.1	Delta-based . . . . .	10
2.6.2	Snapshots (Git) . . . . .	10
2.7	Sûreté de git . . . . .	10
2.8	Trois niveaux de signatures pour la conception d'une version . . . . .	11
2.9	Visualiser l'historique du projet . . . . .	12
2.10	Git est donc . . . . .	12
<b>3</b>	<b>Versionnement d'un projet</b>	<b>13</b>
3.1	Initialisation d'un projet sous git . . . . .	13
3.2	Configuration de votre environnement Git . . . . .	13
3.3	Configuration pour pouvoir travailler sous Git . . . . .	13
3.3.1	Votre identité . . . . .	13
3.4	Ajouter des fichiers à une version . . . . .	14
3.4.1	Comprendre les états des fichiers Git . . . . .	14
3.5	Nouveau dépôt . . . . .	15
3.6	Le fichier ".gitignore" . . . . .	15
3.7	Syntaxe du fichier ".gitignore" . . . . .	15
3.8	Création du fichier ".gitignore" . . . . .	17
3.9	Création d'une version (un <i>commit</i> ) . . . . .	17
3.10	Bonnes pratiques sur les messages de commit . . . . .	18
3.11	Les règles de bonnes pratique pour les commits . . . . .	19
3.12	Un mauvais exemple d'implémentation de git . . . . .	19
3.13	Que faire en cas d'oubli d'un fichier lors d'un commit? . . . . .	19
3.14	Premiers commits . . . . .	20
3.15	Opérations sur les fichiers dans l'arborescence . . . . .	20
<b>4</b>	<b>Branches</b>	<b>20</b>
4.1	"HEAD" . . . . .	20
4.2	Enchaînement des commits . . . . .	21
4.3	Balises . . . . .	21
4.3.1	Balise légères . . . . .	21

4.3.2	Balise annotées . . . . .	21
4.4	Balisage de votre projet . . . . .	21
4.5	Une branche, sur un arbre... . . . .	22
4.6	Organisation de tâches par branches (Workflow commun sous Git) . . . . .	22
4.7	Débuter avec les branches . . . . .	23
4.8	Rapatrifier les modifications d'une branche à une autre . . . . .	23
4.8.1	Fusion . . . . .	23
4.8.2	Réintégration . . . . .	23
4.9	Merge . . . . .	24
4.10	Application . . . . .	24
4.11	Visualisation . . . . .	24
4.12	Gestion des conflits de fusion . . . . .	25
4.13	Création forcée d'un problème de fusion . . . . .	26
4.14	Gestion des conflits de fusion . . . . .	26
4.15	Gestion des conflits de fusion . . . . .	27
4.16	En cas de problème et pour abandonner une fusion en cours: . . . . .	27
4.17	Réintégrer une branche . . . . .	27
4.18	Application de git rebase . . . . .	29
4.19	Rebase . . . . .	29
4.20	Gestion des conflits de fusion . . . . .	30
<b>5</b>	<b>Dépôts distants</b>	<b>30</b>
5.1	Introduction sur les dépôts distants . . . . .	30
5.2	Dépôt de code : Github? . . . . .	31
5.3	Deux fonctionnalités clés sur Github . . . . .	31
5.3.1	Fork . . . . .	31
5.3.2	Issues . . . . .	32
5.4	Autres fonctionnalités de Github . . . . .	33
5.5	Application . . . . .	33
5.6	Interagir avec le dépôt . . . . .	34
5.7	Pull Requests (PR) . . . . .	34
5.7.1	Fonctionnement détaillé des Pull Requests . . . . .	35
5.8	Application Pull Request . . . . .	36

## 1 Problématiques du développement logiciel

### 1.1 Contexte

- Un projet logiciel d'entreprise a **une durée de vie de plusieurs années** et subit de nombreuses évolutions au cours de cette période;
- Il existe des **besoins de livrer de nouvelles versions** qui corrigent des bogues ou apportent de nouvelles fonctionnalités de façon récurrente;
- Même en petites entreprises, un même développement logiciel n'est pas assuré uniquement par une seule personne, mais par **une équipe qui peut varier dans le temps**.

#### Exemples concrets en entreprise :

- Un logiciel de gestion client peut évoluer pendant 10-15 ans avec des mises à jour mensuelles
- Les équipes de développement changent : départs, arrivées, changements de rôles
- Nécessité de maintenir plusieurs versions simultanément (version stable + version de développement)
- Obligation de pouvoir reproduire exactement une version donnée pour corriger un bug critique

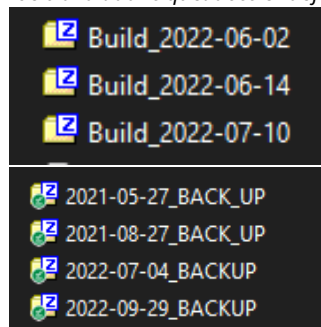
Si un problème intervient au cours de la vie du logiciel, **on doit pouvoir savoir précisément quels fichiers sources font partie de quelle(s) version(s)**.

### 1.2 Situations simples à éviter

Copier son projet dans un autre dossier/archive.

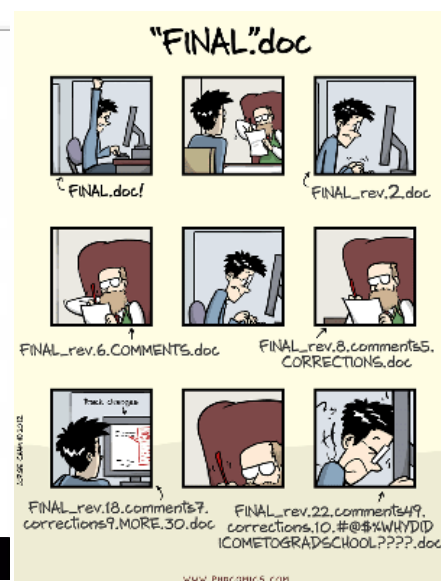
Simple mais dangereux

*"Je travail dans quel dossier déjà?"*



Filename	Last modified
index.html	2017-02-26 07:56:39
index.html	2009-07-20 09:23:19
index.html	2009-07-25 19:22:37
index.html	2009-07-28 10:05:07
index.html	2009-08-07 08:29:23
index.html	2009-08-08 03:24:56
index.html	2009-08-10 08:32:31
index.html	2009-08-17 22:07:20
index.html	2009-08-21 06:55:56
index.html	2009-09-05 01:30:24
index.html	2009-10-11 09:51:32
index.html	2009-10-14 07:11:03
index.html	2009-10-16 10:15:43
index.html	2009-10-23 22:59:09
index.html	2009-10-24 14:31:05
index.html	2009-10-26 00:06:11
index.html	2009-11-09 09:16:28
index.html	2009-11-23 10:02:03
index.html	2009-11-24 10:21:46
index.html	2009-11-30 22:58:23
index.html	2009-12-19 01:50:02
index.html	2010-04-23 23:14:22
index.html	2010-06-28 10:16:59
index.html	2010-08-19 08:19:31
index.html	2010-11-29 08:22:46
index.html	2010-12-28 16:28:04
index.html	2011-02-25 09:10:27
index.html	2011-03-19 10:48:02
index.html	2011-08-24 05:28:04
index.html	2011-11-01 10:20:14
index.html	2013-06-29 02:08:26

**JUST UNDERSCORE IT™**  
EXPERT LEVEL VERSION CONTROL



### 1.3 Les questions qui se posent

Plusieurs questions se posent lors d'un développement logiciel commun à une équipe :

- Comment **publier ses propres modifications** dans un tronc commun?
- Comment **recupérer le travail** d'un autre membre de l'équipe ?
- Comment régler les problèmes en **cas de modifications conflictuelles** (travail sur le même fichier source qu'un ou plusieurs collègues) ?
- Comment **accéder à une version précédente** d'un fichier ou du logiciel entier ?

### 1.3.1 De votre point de vue d'étudiant

Sans même parler de développement logiciel, vous avez déjà été confronté à ces questions.

Lors de la rédaction de vos rapports de TP/TD en groupes, ou lors de la rédaction de vos rapports de stages, par exemple.

Comment procédiez-vous pour travailler sur les mêmes documents?

#### Problèmes typiques rencontrés :

- Envoi de fichiers par email avec des versions “finale”, “finale\_v2”, “finale\_vraiment\_finale”
- Perte de modifications quand plusieurs personnes travaillent simultanément
- Difficulté à savoir qui a fait quoi et quand
- Impossibilité de revenir à une version antérieure qui fonctionnait
- Conflits lors de la fusion de parties rédigées séparément

#### Solutions actuelles limitées :

- Google Drive/OneDrive : versioning basique, pas de gestion des conflits
- Dropbox : conservation des versions mais sans organisation claire
- SharePoint : complexe à utiliser pour de simples documents

## 1.4 L'intérêt d'un gestionnaire de versions

Tout projet logiciel d'entreprise (même mono-développeur) doit faire l'objet d'une **gestion des versions**.

(Revision Control System, **VCS**: **V**ersion **C**ontrol **S**ystem ou versioning).

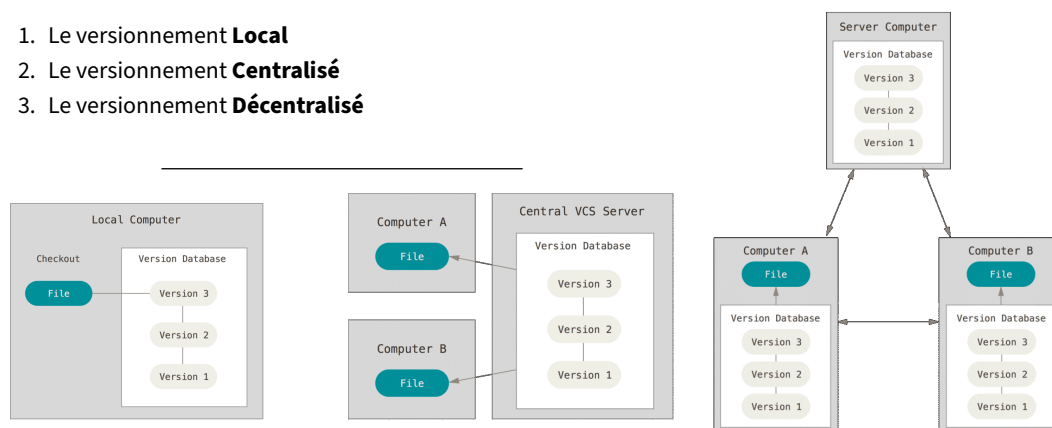
### 1.4.1 Les objectifs de la gestion des versions

La gestion des versions vise les objectifs suivants :

- Un logiciel de gestion des versions est avant tout un **dépôt de code** qui héberge le code source du projet et **définit la manière dont le code sera accessible**. Chaque développeur peut accéder au dépôt afin de récupérer le code source, et y publier ses modifications.
- Le logiciel garde **l'historique** des modifications de chaque fichier du projet et permet de revenir à une version antérieure.
- Un logiciel de gestion des versions permet de **travail collaboratif** et en cas d'apparition d'un **conflit** (modifications simultanées du même fichier par plusieurs développeurs) et doit permettre de le corriger.

## 1.5 Trois architectures possibles de versionnement

1. Le versionnement **Local**
2. Le versionnement **Centralisé**
3. Le versionnement **Décentralisé**



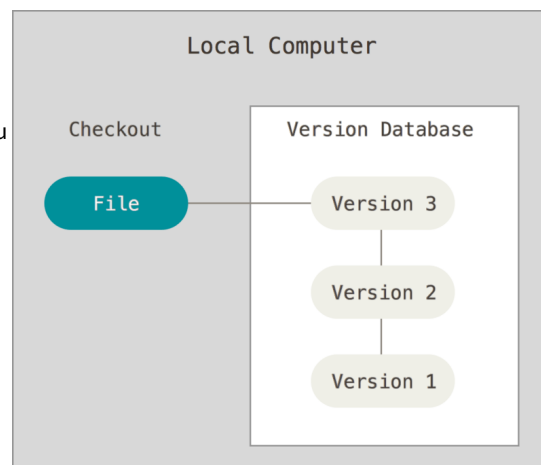
## 1.6 VCS Local

(Solution : GNU RCS)

- Une simple base de données des changements apportés au projet en cours;
- Un seul point de défaillance (tout est sur disque en local);
- Impossible de collaborer depuis d'autres systèmes;

Exemple:

- Tout les désavantages d'une Clé USB



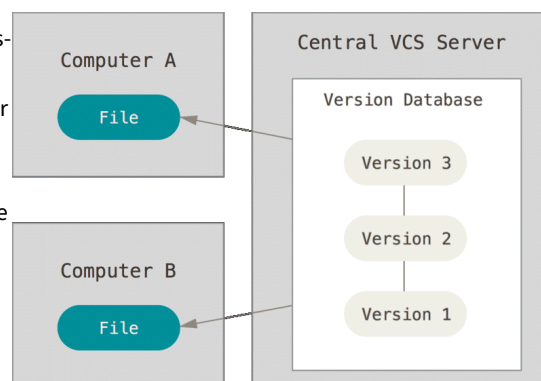
## 1.7 VCS Centralisé

(Solutions : CVS, Subversion (SVN), Perforce, etc.)

- Architecture **client/serveur**
- Collaboration possible.
- Mais la fusion des modifications est complexe (voire impossible)
- Les utilisateurs ont accès aux fichiers dont ils ont besoin sur demande.
- Le projet est hébergé sur un serveur.
- Mais toujours un seul point de défaillance (le serveur cette fois-ci)

Exemple:

- Google Drive / Dropbox
- Parfois seule solution viable: SolidWorks
- "Incendie d'OVHcloud" (Strasbourg 9-10 mars 2021)



**L'exemple OVHcloud illustre parfaitement les risques :** Le 9-10 mars 2021, un incendie a détruit l'un des quatre datacenters OVHcloud et lourdement endommagé un deuxième à Strasbourg, causant :

- La perte définitive de données pour certains clients qui n'avaient pas de sauvegarde
- L'arrêt de milliers de sites web et applications, applications, intranets, messagerie...
- Des pertes financières considérables pour les entreprises affectées

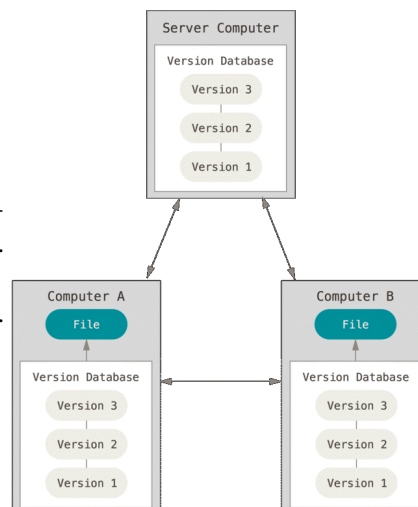
**Pourquoi SolidWorks reste centralisé :**

- Les fichiers CAO sont très volumineux (parfois plusieurs GB)
- Nécessité d'un contrôle strict des accès pour éviter les modifications simultanées
- Gestion des licences coûteuses centralisée
- Collaboration en temps réel sur une même pièce impossible

## 1.8 VCS Décentralisé

(Solutions: **Git**, Mercurial, Bazaar, Darcs, etc.)

- Architecture **pair-à-pair**
- Les utilisateurs n'ont pas accès qu'aux fichiers dont ils ont besoins, mais à la copie et l'historique de **l'ensemble du projet**.
- Tout les collaborateurs peuvent partager leur modifications sur le serveur et restaurer les versions précédentes du projet.
- Un projet est **cloné** et **disséminé** autant de fois que nécessaire pour y collaborer et maintenir son intégration continue (CI).



## 2 Avantages de Git

### 2.1 Focus sur Git

Développé par *Linus Torvald* en 2005 pour encadrer le développement du noyau *Linux* (suite à des problèmes légaux avec le précédent fournisseur du VCS utilisé : BitKeeper).



**Contexte historique :** Avant 2005, le noyau Linux utilisait BitKeeper, un VCS propriétaire gratuit pour les projets open source. En 2005, BitKeeper a retiré la licence gratuite, forçant la communauté Linux à chercher une alternative. Aucune solution existante ne répondait aux besoins spécifiques du développement du noyau :

- Performance exceptionnelle nécessaire (milliers de commits par jour)
- Support de workflows distribués complexes
- Intégrité absolue des données
- Simplicité d'usage malgré la complexité sous-jacente

Linus Torvalds a donc décidé de créer Git en seulement 10 jours, rien que ça...

Pour plus d'informations, vous pouvez consulter l'interview de Linus Torvalds pour les vingt ans de Git: [Git turns 20: A Q&A with Linus Torvalds](#)

---

Nécessités à l'époque:

- Rapidité
  - Conception simple
  - Supporter le développement simultané
  - Projet entièrement distribué
  - Capacité à gérer efficacement de grands projets (comme le noyau Linux)
-


### 2.1.1 Pourquoi décentraliser le travail sur le noyau Linux?

- 17+ millions de lignes de code
- 1500+ contributeurs récurrents qui y travaillent simultanément.

<https://git.kernel.org/>

<https://github.com/torvalds/linux>



 **Kernel.org git repositories**  
Git repositories hosted at kernel.org (dallas)

Name	Description
pub/scm/bluetooth	Unnamed repository; edit this file 'description' to name the repository.
bluetooth-next.git	Bluetooth packet analyzer
bluez-hcidump.git	Bluetooth protocol stack for Linux
bluez.git	OBEEX Server
obexd.git	Bluetooth low-complexity, subband codec (SBC) library
sbc.git	
pub/scm/boot	
dracut/dracut.git	dracut - initramfs generator using udev
eflinux/eflinux.git	The eflinux EFI boot loader
syslinux/syslinux.git	Unnamed repository; edit this file 'description' to name the repository.
pub/scm/development	
pahole/pahole.git	Pahole and other DWARF utils
sparse/chris/sparse.git	Chris L's sparse repository.
sparse/sparse-dev.git	Sparse's development tree
sparse/sparse-logs.git	Raw logs of warnings of Sparse running on the kernel with summary logs.
sparse/sparse.git	C semantic parser
pub/scm/docs	
docs/lee/1394.git	Documentation for Linux FireWire subsystem
docs/korg.git	Source of korg.docs.kernel.org
kernel/kernel-docs.git	Kernel Documentation tree
kernel/kmap.git	Kernel.org keymap source
kernel/pgpkeys.git	Kernel developers PGP keys
kernel/website.git	Kernel.org website source
man-pages/man-pages-posix.git	POSIX manual pages - sections 0p, 1p, 3p
man-pages/man-pages.git	Linux man pages Sections 2, 3, 4, 5, and 7
man-pages/website.git	Website files for /doc/man-pages
tab/tab.git	Unnamed repository; edit this file 'description' to name the repository.
pub/scm/editors	
uemacs/uemacs.git	Micro-emacs

### Chiffres impressionnants du noyau Linux :

- Plus de 17 millions de lignes de code en C
- Plus de 1 500 développeurs actifs chaque année
- Environ 8-10 commits par heure, 24h/24, 7j/7
- Versions majeures tous les 2-3 mois
- Support de milliers d'architectures matérielles différentes
- Utilisé sur 97% des supercalculateurs, la plupart des smartphones Android, serveurs web, etc.

### Défis techniques uniques :

- Gestion de contributions simultanées de développeurs dans le monde entier
- Nécessité de tester sur de multiples plateformes avant intégration
- Processus de revue de code très strict (plusieurs niveaux de validation)
- Branches de développement multiples (stable, développement, expérimental)

## 2.2 Intégration de git dans vos IDE

Git est de loin le VCS le plus largement utilisé aujourd'hui.

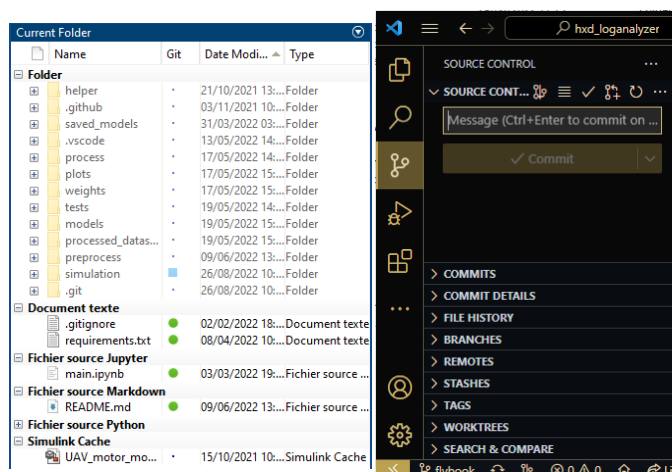
Estimé à plus de 98% de part d'utilisation dans les projets open source et d'entreprise.

Il est suffisamment répandu pour être intégré dans la majorité des IDE (Environnement de Développement Intégré) et logiciels que vous pourriez utiliser.

VSCode intègre git nativement via une interface graphique qui permet facilement de créer des commits, de se déplacer entre versions, de gérer des branches et de résoudre des conflits.

Sous Matlab, une colonne "Git" apparaît dans l'arborescence des fichiers si le dossier courant est un dépôt git.





## 2.3 Installation de Git

Avant de procéder à l'installation, ouvrez un terminal et lancez la commande suivante pour savoir si git est déjà présent sur le système :

```
1 | git --version
```

Si git est déjà installé, tant mieux..

Pour les autres :

OS	Commande
Linux (Debian)	<code>sudo apt-get install -y git</code>
Windows	<a href="https://git-scm.com/download/win">https://git-scm.com/download/win</a>
iMac (Non-recommandé)	<a href="https://git-scm.com/downloads/mac">https://git-scm.com/downloads/mac</a> ou
pour ce cours)	<a href="https://sourceforge.net/projects/git-osx-installer/files/">https://sourceforge.net/projects/git-osx-installer/files/</a>

### Installation détaillée par système :

#### Linux (Ubuntu/Debian) :

```
1 | # Mise à jour de la liste des paquets
2 | sudo apt update
3 | # Installation de git
4 | sudo apt install git-all
5 | # Vérification
6 | git --version
```

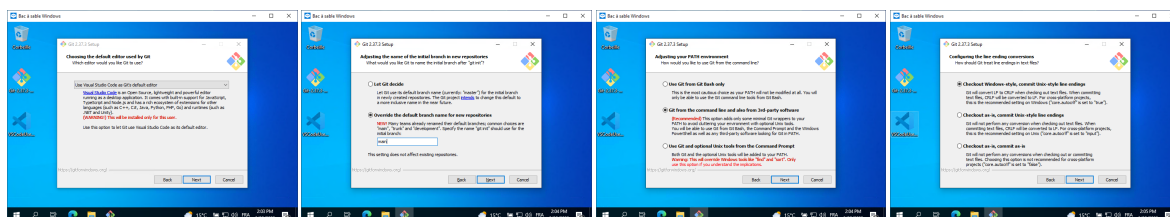
#### Windows :

- Télécharger depuis <https://git-scm.com/download/win>
- Accepter les options par défaut lors de l'installation
- **Important** : Choisir "main" comme nom de branche par défaut
- Installer "Git Bash" pour avoir un terminal Unix-like sous Windows

#### macOS :

```
1 | # Avec Homebrew (recommandé)
2 | brew install git
3 | # Ou depuis le site officiel
```

## 2.4 Installation de Git (Windows)



<https://git-scm.com/download/win>

“Override the default branch name”: `main`

Validez l’installation en ouvrant un terminal et en tapant : `git --version`

## 2.5 Pourquoi Git? Utilisation par la pratique

Copiez le dossier suivant sur votre session depuis le dossier commun:

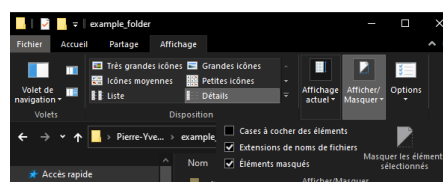
`Q:\5A\0-Transport Connecté Intelligent (TCI)\Gestion de Versions\example_folder`

Puis ouvrez un terminal (Git Bash sous Windows) et utilisez les commandes suivantes pour naviguer dans le dossier:

Opération	Commande
Afficher le statut du projet	<code>git status</code>
Changer de dossier	<code>cd [path]</code>
Lister le contenu du dossier	Unix: <code>ls -al</code> Win: <code>dir [/a: all, /a:h: all-hidden]</code>
Afficher le contenu d’un fichier	Unix: <code>cat [filename]</code> Win: <code>more [filename]</code>

Dans le dossier “example\_folder” précédemment copié, vous y trouverez:

- Un dossier caché: `.git`
- Un fichier visible: `.gitignore`



### 2.5.1 Questions

1. Le dossier `.git` contient-il une copie des fichiers?
2. Le fichier HEAD contient un chemin:
  1. Où pointe-t-il?
  2. L’objet pointé contient un code. Quel longueur le code fait-il?
3. Plusieurs dossiers sont présents dans `.git/objects` :
  1. Que contiennent-ils ?
  2. Quelle est la taille du nom de ces fichiers? Et avec le nom du dossier?
  3. Le contenu de ces fichiers est-il lisible?

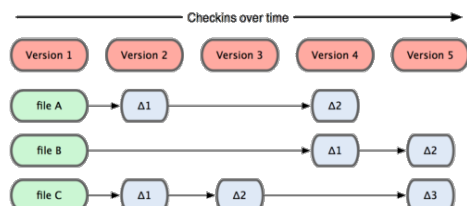
## 2.6 Rapidité de git

Git est connu pour sa rapidité. Comment expliquer cette performance?

Deux méthodes sont couramment utilisées pour effectuer le versionnement de fichiers:

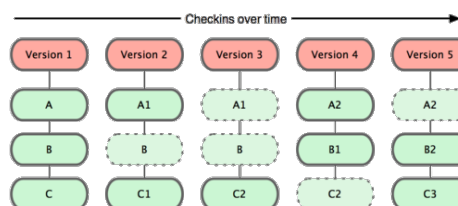
### 2.6.1 Delta-based

Idée initiale simple pour versionner: stocker les fichiers initiaux et les faire évoluer en ne sauvegardant **que** les modifications apportées



### 2.6.2 Snapshots (Git)

Sauvegarde de **l'ensemble des fichiers de la version** (seulement ceux modifiés). Encodés et compressés, ce qui les rends disponibles très rapidement.



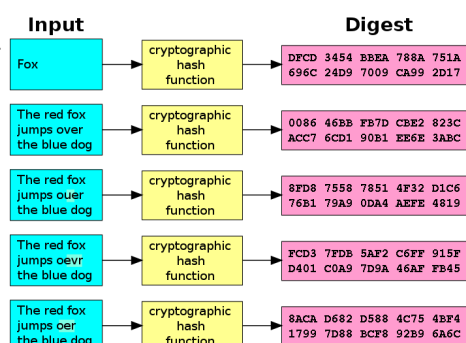
Git est rapide, parce qu'il garde une copie (compressée) de **toutes les versions, de tous les fichiers du projet en local**.

## 2.7 Sûreté de git

Git est sûr, parce qu'il suit le moindre bit des fichiers présents dans le projet.

Le moindre changement apporté à un fichier sera donc perçu et suivi.

De plus, la somme de contrôle (checksum) permet de s'assurer que les fichiers remontés ne peuvent pas être corrompus.



La fonction de hachage *SHA-1* utilisée pour la somme de contrôle prend un fichier et calcule une valeur de 160 bits (soit **40 chiffres hexadécimaux**).

— 1 chance sur  $2^{80}$  d'avoir une clé similaire.

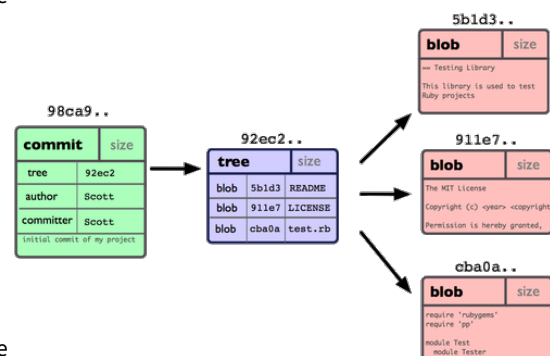
Encoder du texte ou un fichier:

```
1 echo "a" | sha1sum
2 > 3f786850e387550fdab836ed7e6dc881de23001b
3 sha1sum README.md
4 > 149c537b11f54bd65307c2b96e8b9791566e55f5 README.md
```

## 2.8 Trois niveaux de signatures pour la conception d'une version

Il existe trois types d'objets dans Git, chacun avec une signature unique:

Signature	Signification
1. Commit	Le contexte de la version
2. Tree	L'arborescence des fichiers
3. Blob (Binary Large Object)	Les fichiers eux-mêmes



Ces signatures agissent comme des pointeurs permettant à Git de remonter aux états passés:

```

1 | git hash-object README.md
2 | git cat-file -p 7117cf9e99d030083a3229108005e9e08fd8d029

```

```

1 | echo 'test' | git hash-object -w -stdin
2 | git cat-file -p main^{tree} Win:main^{tree}

```

## 2.9 Visualiser l'historique du projet

Opération	Commande
Aperçu des précédentes versions	<code>git log</code>
Liste des commits + modifications	<code>git log -p</code>
Liste des commits + statistiques	<code>git log --stat</code>
Aperçu des modification d'un commit	<code>git show [commit_hash]:[file]</code>
Aperçu des modifications individuelles	<code>git blame [file]</code>
Déplace le projet à la version spécifiée	<code>git checkout [hash]</code> <code>git checkout main</code>

Quelques questions sur le projet d'exemple:

1. Combien de commits totaux dans le projet d'exemple?
2. Quand est-ce que le support Matlab a été ajouté?
3. Quel fichier a été modifié plusieurs fois? (Avec quel intervalle de temps)
4. Visualisez la première version de ce fichier (Deux façons possible)

### Commandes Git utiles pour le dépannage :

```

1  # Voir l'historique détaillé
2  git log --oneline --graph --all --decorate

3  # Voir qui a modifié quoi dans un fichier
4  git blame filename.txt

5  # Trouver quand un bug a été introduit
6  git bisect start
7  git bisect bad          # commit actuel (buggé)
8  git bisect good v1.0.0 # dernière version connue comme stable

9  # Voir les différences entre deux commits
10 git diff HEAD~2 HEAD

11 # Voir les changements d'un commit spécifique
12 git show <commit-hash>

```

## 2.10 Git est donc

Très utile pour le versionnement des **fichiers textes (donc pour le code source)**, pour visualiser les différences et pour cibler l'origine des modifications.

Pas très pratique pour les “fichiers binaires” (exécutables), “fichiers archives” ou autres type de fichiers compilés. C’est le cas:

- Des documents Office365
- Des PDF
- Archives HDF5
- Modèles NN
- Modèles CAO
- etc.

## 3 Versionnement d'un projet

### 3.1 Initialisation d'un projet sous git

Copiez en local le dossier "example\_project\_2" présent sur le réseau vers votre session:

```
Q:\5A\0-Transport Connecté Intelligent (TCI)\Gestion de Versions
```

Mais avant de poursuivre...

```
1 | git config
```

### 3.2 Configuration de votre environnement Git

Git utilise l'outil `git config` qui permet de configurer les variables qui contrôlent son fonctionnement.

Il y existe **3 niveaux** de configuration de ces variables : **Local, Global & System**.

1. **Local** : Les configurations locales sont disponibles pour le dossier/projet de travail **uniquement** et sont stockées dans `[gitrepo]/.git/config`  
(Exemple : `[example_folder]/.git/config`)
2. **Global** : Les configurations globales sont disponibles pour tous les projets de **l'utilisateur actuel** et sont stockées dans `~/.gitconfig` ou `~/.config/git/config`.  
(Exemple : `C:/Users/[Username]/.gitconfig`)
3. **Système** : Disponibles pour **tous les utilisateurs du système**.  
(Exemple : `C:/Program Files/Git/etc/gitconfig`). Cette option est peu, voire pas recommandée.

On configurera la configuration en **global** pour que vous utilisiez la même configuration sur l'ensemble de votre session.

### 3.3 Configuration pour pouvoir travailler sous Git

#### 3.3.1 Votre identité

Configurez votre **nom** et **adresse mail** qui vous servira à *signer vos commits* (Utilisez votre adresse mail ESTACA)

```
1 | git config --global user.name "John Doe"
2 | git config --global user.email john.doe@estaca.eu
```

Validez la configuration avec

```
1 | git config --list
2 | git config --list --global
```

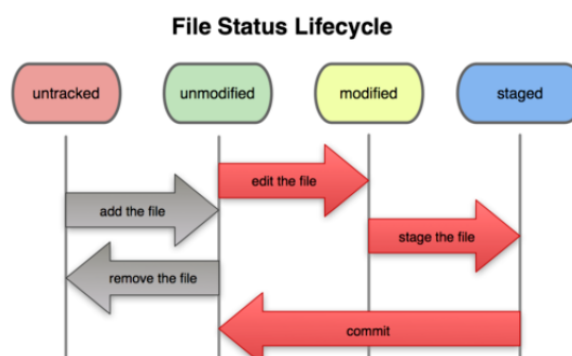
### 3.4 Ajouter des fichiers à une version

Pour créer une version/commit, il est nécessaire d'y ajouter les fichiers que l'on souhaite versionner.

Un fichier peut se trouver dans 4 états différents:

1. **Non-suivis** : Fichiers non-affectés par le VCS
2. **Non-modifiés** : Fichiers inchangés
3. **Modifiés**
4. **Indexés/staged** : Fichiers sélectionnés pour la prochaine version/commit

("staging area" : aire de transit)



`git status` Permet de voir l'état actuel des fichiers

#### 3.4.1 Comprendre les états des fichiers Git

##### 1. Non-suivis (Untracked) :

- Nouveaux fichiers pas encore ajoutés à Git
- Apparaissent en rouge avec `git status`
- Git les ignore complètement jusqu'à ce qu'on les ajoute

##### 2. Non-modifiés (Unmodified) :

- Fichiers déjà dans Git, sans changement depuis le dernier commit
- N'apparaissent pas dans `git status` (tout va bien)

##### 3. Modifiés (Modified) :

- Fichiers suivis par Git avec des changements
- Apparaissent en rouge avec `git status`
- Différences visibles avec `git diff`

##### 4. Indexés/Staged :

- Fichiers préparés pour le prochain commit
- Apparaissent en vert avec `git status`
- Différences visibles avec `git diff --cached`

#### Pourquoi une staging area ?

- Permet de préparer des commits précis
- On peut committer une partie des changements seulement
- Révision avant commit définitif
- Possibilité de grouper des modifications logiquement liées

#### Commandes utiles pour naviguer :

```

1  # Voir l'état complet
2  git status
3
4  # Voir les différences non-indexées
5  git diff
6
7  # Voir les différences indexées (fichiers dans l'aire de transit)
8  git diff --cached
  
```

```
7 # Voir toutes les différences
8 git diff HEAD
```

### 3.5 Nouveau dépôt

Copiez le dossier “`example_folder_2`” sur le réseau et initialisez le en tant que dépôt git.

Opération	Commande
Initialisation du dépôt	<code>git init</code>
Etat des fichiers	<code>git status</code>
Indexer des modifications (staged)	<code>git add [file]</code> <span style="float: right;"><code>git add .</code></span>
Désindexer des modifications (unstaged)	<code>git reset (HEAD) [file]</code> <span style="float: right;"><code>git reset .</code></span>
Aperçu des différences non-indexées	<code>git diff</code>
Aperçu des différences indexées	<code>git diff --cached</code>

1. Initialisez le dépôt `example_folder_2`
2. Indexez les fichiers un par un et validez l'indexage avec `git status`
3. Indexez tous les fichiers et validez l'indexage
4. Désindexez tous les fichiers (retour à l'initialisation)

**Attention**, pour désindexer un fichier (“unstage”), **ne pas utiliser**: `git rm [file]`

### 3.6 Le fichier “.gitignore”

Le fichier “.gitignore” est un simple fichier texte qui permet d'ignorer automatiquement certains fichiers pendant l'indexage. Il est à utiliser pour:

- Cacher des **secrets** (e.g. clés d'API, noms d'utilisateurs, etc.)
- Enlever des **configuration locales** optionnelles (e.g. dossier “.vscode”)
- Enlever des **fichiers trop lourds** (Github: limite des fichiers individuel à 100Mo)
- Éviter les **fichiers temporaires**, de **compilations**, etc. (Tout ce qui va être régénéré quoiqu'il arrive)

Exemples :

- les dossiers de compilation : `build/`
- les fichiers cache ou back-up de Simulink : `_.slxc` | `_.autosave`
- les fichiers textes : `*.txt` -> sauf, la liste des dépendances : `!requirements.txt`

### 3.7 Syntaxe du fichier “.gitignore”

Syntaxe : “glob patterns”	Signification
<code>*.txt</code>	Ignore tout les fichiers qui termine par ‘.txt’
<code>!file.txt</code>	‘!’ Exception aux règles précédentes dans le fichier
<code>file[0-9].png</code>	Ignore les fichiers avec un chiffre à cette position
<code>file?.png</code>	Les fichiers où ‘?’ n’est pas vide (ex: ‘files.png’) ‘file_test.png’ ne remplis pas cette condition



Syntaxe : “glob patterns”	Signification
/build	‘/’ Ignorer un dossier
folder/	Ignore tout les fichiers présents dans le dossier

Il existe une base de données de fichiers “.gitignore” pour à peu près tout les langages de programmations que vous pouvez utiliser en tant que base: <https://github.com/github/gitignore>

En cas d’oubli d’un fichier déjà indexé : `git rm --cached [file]`

### Exemples détaillés de fichiers à ignorer :

#### Secrets et configurations sensibles :

```

1 # Clés API et configurations
2 .env
3 config.json
4 *.key
5 *.pem
6 credentials.txt

```

#### Fichiers de compilation et build :

```

1 # Java
2 *.class
3 *.jar
4 target/
5
6 # Python
7 __pycache__/
8 *.pyc
9 *.pyo
10 build/
11 dist/
12
13 # C/C++
14 *.o
15 *.exe
16 *.so

```

#### Fichiers IDE et éditeurs :

```

1 # Visual Studio Code
2 .vscode/
3 # IntelliJ IDEA
4 .idea/
5 # Vim
6 *.swp
7 *.swo

```

#### Fichiers système :

```

1 # Windows
2 Thumbs.db
3 Desktop.ini
4 # macOS
5 .DS_Store
6 # Linux
7 *~

```

### 3.8 Création du fichier “.gitignore”

Créez un fichier `.gitignore` dans le dossier `example_folder_2` et ajoutez les patterns permettant de respecter les règles suivantes:

1. Ignorez tout les fichiers inutiles, temporaires et/ou secrets: “`DSStore`”, “`clé*`”, `.asv`, etc.
2. Dans le dossier `plots/`, ne gardez que les fichiers `plot_accurate.png` et `plot7.png`
3. Validez que les fichiers à ignorer n'apparaissent plus en tant qu'*untracked files* avec `git status`.
4. Indexez ensuite tous les fichiers restants.

Il ne doit vous rester que: `README.md`, `matlab/script.m`, `plots/plot7.png`, `plots/plot_accurate.png` et `python/script.py` lorsque vous utilisez `git status`.

### 3.9 Création d’une version (un *commit*)

Pour soumettre les modifications indexés et créer une version du projet, il faut créer un **commit**.

Il doit être accompagné d’un **message** qui résume les modifications apportées.

```
1 | git commit -m "[50 characters message]" [-m "[msg2]" ...]
2 | git commit --message "[50 characters message]" [-m "[msg2]" ...]
```

Le message d’un commit fait moins de **50 caractères** et sert à identifier la version.

Il doit **présenter/résumer succinctement les modifications** que vous venez d’apporter au projet et leurs justifications de façon claires et concises.

Communément, les commits respectent des conventions de nommage:

Type de modification	Message de commit
Premier commit	“Initial/First commit”
Fix de bug	“Fix issue ...”
Ajout de fonctionnalité	“Add feature ...”

#### Conventions de commit les plus populaires :

##### Conventional Commits :

```
1 | feat: ajout de la fonction de login
2 | fix: correction du bug de calcul de TVA
3 | docs: mise à jour du README
4 | style: correction indentation
5 | refactor: réorganisation du code auth
6 | test: ajout tests unitaires pour login
7 | chore: mise à jour des dépendances
```

##### Autres exemples concrets :

```
1 | Add user authentication system
2 | Fix memory leak in data processing
3 | Update documentation for API v2
4 | Remove deprecated functions
5 | Improve error handling in parser
```

**Ce qu'il faut éviter :**

- “stuff”
- “update”
- “fixes” (sans précision)
- “Work in progress”
- Messages trop longs ou trop techniques
- Messages en plusieurs langues mélangées
- Les émojis...

**Règle des 7 mots :** Un bon message de commit doit pouvoir compléter la phrase : “If applied, this commit will...”

Exemple : “If applied, this commit will *fix the login bug for Chrome users*”

**3.10 Bonnes pratiques sur les messages de commit**

Message 1	<div>Capitalized, short (50 chars or less) summary</div> <div>More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.</div>
Message 2	<div>Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like git merge and git revert.</div> <div>Further paragraphs come after blank lines.</div> <div>- Bullet points are okay, too</div> <div>- Typically a hyphen or asterisk is used for the bullet, followed by a single space, with blank lines in between, but conventions vary here</div> <div>- Use a hanging indent</div>

La description d'un commit ressemble à un email. Ce n'est pas une coïncidence.

Le message “1” constitue l'objet du commit. Le message “2” en constitue le corps. Les fichiers modifiés sont l'objet du commit.

### 3.11 Les règles de bonnes pratique pour les commits

Parce qu'un commit crée une snapshot de votre projet, il doit être **réfléchi** afin de ne pas polluer l'historique et augmenter inutilement la taille du dépôt.

- Un commit doit être **justifié par l'ajout d'une fonctionnalité ou la résolution d'un problème**.
- Les messages de commit, doivent être concis et clairs. Ce sont les titres de vos corrections/ajouts.
- Certains projets OSS vous imposent un formatage précis pour les messages de vos commit.

Exemple : <https://github.com/ArduPilot/ardupilot/commits>



### 3.12 Un mauvais exemple d'implémentation de git

Un exemple d'implémentation de Git catastrophique...

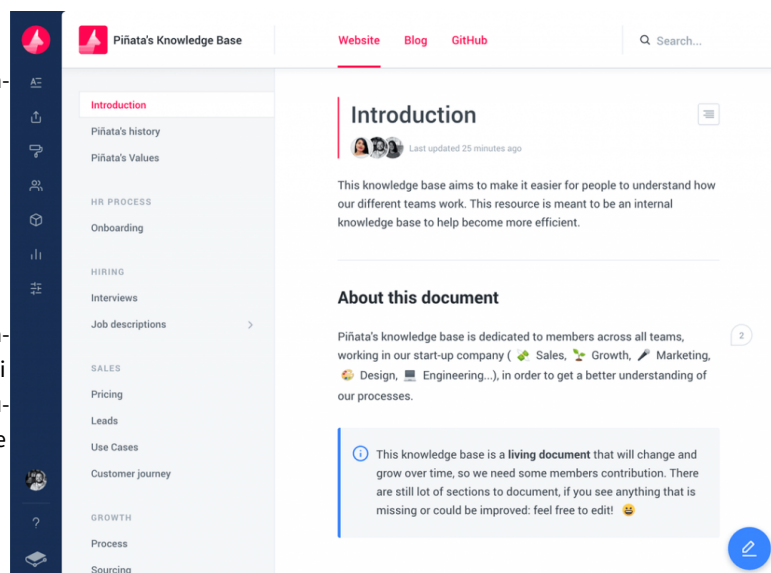


## GitBook

Gitbook est un générateur de sites statiques permettant de créer des livres et de la documentation.

Lors de l'édition du texte depuis la plateforme, le commit est lié au raccourci [Ctrl+S], ce qui crée des commits sans aucune cohésion et avec un unique message générique.

GitBook: No commit message



### 3.13 Que faire en cas d'oubli d'un fichier lors d'un commit?

Pour modifier le dernier commit avec les modifications actuellement indexées :

```
1 | git commit --amend --no-edit
```

Exemple :

```
1 | git commit -m "initial commit"
2 | git add forgotten_file
3 | git commit --amend --no-edit
```

Le commit précédent est réécrit.

**A ne pas abuser** (changement d'arborescence, ajout de fichiers -> Création de nouvelles signatures, etc.)

### 3.14 Premiers commits

1. Validez d'avoir indexé tous les fichiers de l'exercice précédent.
2. Créez le premier commit `"initial commit"`
3. Déplacez manuellement le fichier `plot_accurate.png` dans le dossier `matlab/`.  
(Après l'avoir indexé, notez que git a reconnu que le chemin du fichier avait simplement été renommé)
4. Créez un second commit présentant la modification apportée.
5. Oups! Créez une modification au fichier `.gitignore` pour garder la figure `plot0.png` que vous aviez ignoré précédemment. Modifiez le dernier commit pour y intégrer cette modification.

### 3.15 Opérations sur les fichiers dans l'arborescence

Git permet aussi de réaliser des opérations communes, comme la suppression et le déplacement des fichiers présents, en prenant en compte ces modifications dans l'historique:

Opération	Commande
Supprimer un fichier + Désindexation automatique	<code>git rm [file]</code>
Récupérer un fichier supprimé mais déjà indexé	<code>git checkout HEAD [file]</code>
Déplacer des fichiers indexés	<code>git mv [old_path] [new_path]</code>

`git mv` est équivalent à: `mv [old_path] [new_path] | git rm [old_path] | git add [new_path]`

1. Indexez tous les fichiers
2. Déplacez le fichier `script.py` hors du dossier `python/`, à la racine du projet.
3. Supprimez le fichier `matlab/script.m`. Puis revenez en arrière.
4. Désindexez tous les fichiers (retour à l'initialisation) (Notez que le script est resté à son nouvel emplacement)

## 4 Branches

### 4.1 "HEAD"

```
1 | git reset HEAD [fichier]
2 | git checkout HEAD [fichier]
```

La tête (HEAD) est la version courante du projet sur laquelle vous travaillez. Il s'agit généralement du bout de la branche principale (main/master) si vous travaillez sur la dernière version.

C'est donc la "tête" de la branche actuelle.

Lorsque vous créez un commit, HEAD se déplace automatiquement vers le nouveau commit.

La commande suivante permet de réinitialiser l'index (staging area) au dernier commit:

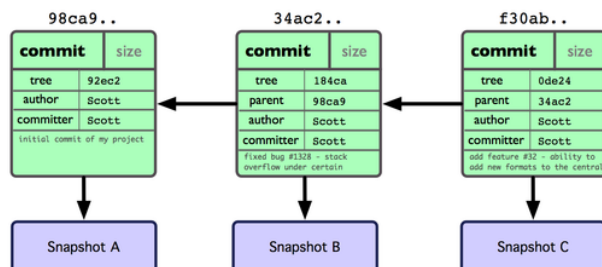
```
1 | git reset HEAD --hard
```

(A utiliser en cas de problème)

## 4.2 Enchaînement des commits

La branche principale d'un projet est une succession linéaire de commits, suivant un principe de base implicite: **Chaque commit possède un unique parent (un commit précédent)**

La succession de plusieurs commits peut être considérée comme une "branche"



## 4.3 Balisage

Pour accentuer certains commits - Git permet le balisage de versions pour marquer des étapes importantes dans l'historique du projet.

Il existe deux types de balises : **légères** et **annotées**. Avec la même commande `git tag ...`

### 4.3.1 Balise légère

Une balise légère correspondrait à une branche figée qui ne changerait pas, il s'agit d'un simple pointeur vers un commit spécifique.

```
1 | git tag v1.4-lw
```

### 4.3.2 Balise annotées

Les balises annotées, par contre sont stockées en tant qu'objets à part entière dans la base de données du dépôt.

```
1 | git tag -a v1.4-lw -m "[message]"
```

## 4.4 Balisage de votre projet

Pour facilement retrouver une version stable de votre projet, vous allez créer un tag annoté `v1.0.0` sur votre dernier commit:

1. Ajoutez le tag `v1.0.0` sur votre dernier commit (le second) - `git tag -a v1.0.0 -m "Version 1.0.0"`
2. Validez que le commit est correctement balisé avec `git log`
3. Revenez à l'état de votre tout premier commit (checkout).
  1. Observez l'historique à partir de ce commit (absence des prochains commits) - `git log`
  2. Observez votre position actuelle dans l'historique complet - `git log --all`
4. Revenez vers votre tag `v1.0.0` - `git checkout v1.0.0`
5. Regardez le contenu du dossier `.git\refs\tags`

## 4.5 Une branche, sur un arbre...

Avec la logique d'un seul parent, on peut faire diverger notre travail sur différentes branches.

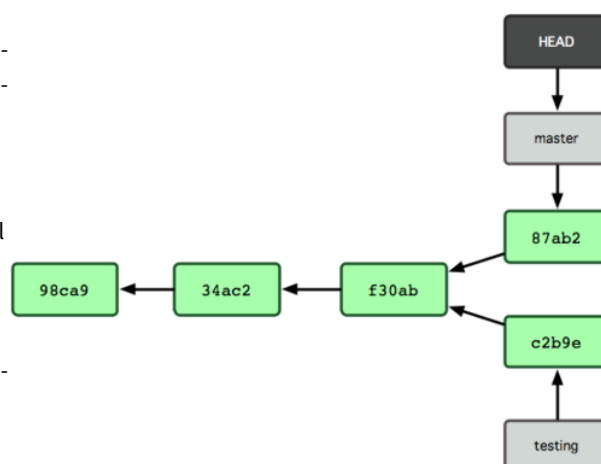
La base d'une branche doit partir d'un état du projet (idéalement fonctionnel) qui va pouvoir évoluer vers différents sous-versions.

Par exemple, on souhaite appliquer la même base d'un travail à différents sous-projets.

### Exemples:

- Une template web déclinée pour différents clients
- Un OS pour microcontrôleur dérivé pour différents modèles de cartes
- ...

Ou plus simplement juste pour un test de nouvelles fonctionnalités

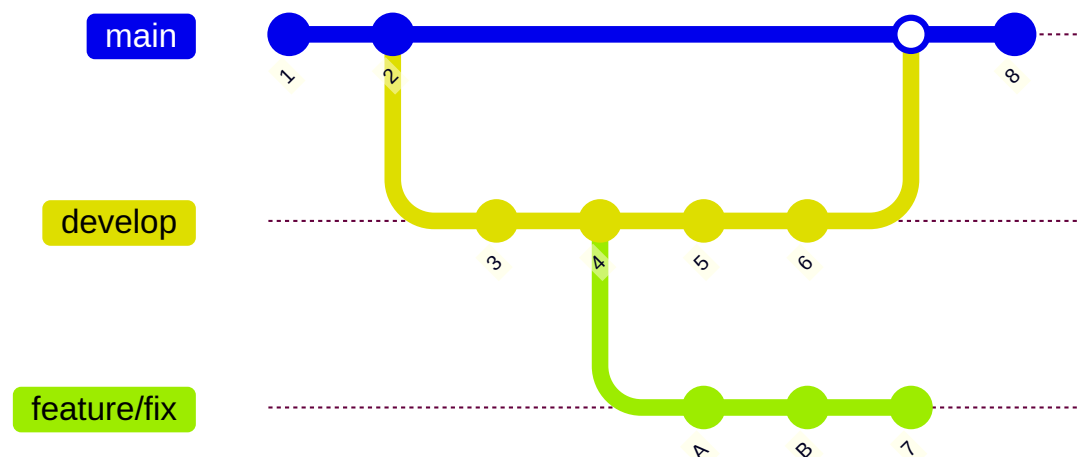


## 4.6 Organisation de tâches par branches (Workflow commun sous Git)

Généralement, on retrouve au moins deux branches principales dans un projet:

- `main` (ou `master`) : version stable et déployable en production
- `develop` : intégration des nouvelles fonctionnalités

Le nom de ces branches varie en fonction des équipes et des projets. Mais l'idée reste la même.



### Workflows Git les plus utilisés en entreprise :

#### Git Flow (pour projets avec versions planifiées) :

- `main` : version de production stable
- `develop` : intégration des nouvelles fonctionnalités
- `feature/*` : développement de fonctionnalités individuelles
- `release/*` : préparation des versions
- `hotfix/*` : corrections urgentes en production

#### GitHub Flow (pour développement continu) :

- `main` : toujours déployable

- Branches feature courtes (quelques jours max)
- Pull Requests pour validation avant fusion
- Déploiement automatique depuis main

#### GitLab Flow (hybride) :

- Combine avantages des deux approches
- Branches d'environnement (staging, production)
- Workflows adaptés aux équipes

#### Bonnes pratiques pour les noms de branches :

```
1 feature/auth-system
2 bugfix/login-error
3 hotfix/security-patch
4 improvement/performance-db
```

{Pour les joueurs de D&D : <https://programmerhumor.io/git-memes/the-git-branch-alignment-chart-gz3y>}

## 4.7 Débuter avec les branches

Définir une branche pour organiser son travail:

- `git branch` pour effectuer des opérations sur les branches.
- `git checkout` pour se déplacer entre les branches.

Opération	Commande
Liste les branches existantes du projet	<code>git branch</code>
Créer une branche	<code>git branch [new_branch]</code>
Se déplacer vers une branche	<code>git checkout [new_branch]</code>
	<code>git checkout (main/master)</code>
Raccourci pour créer et s'y déplacer	<code>git checkout -b [new_branch]</code>
Supprimer une branche	<code>git branch -d [branch]</code>

Essayez simplement de lister les branches existantes de votre projet avec `git branch` pour l'instant.

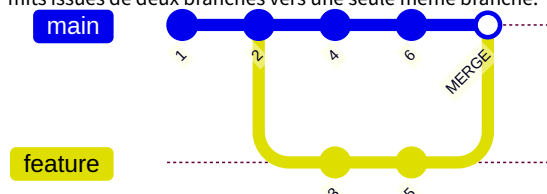
## 4.8 Rapatrier les modifications d'une branche à une autre

Ok, mais on ne va pas se contenter de décomposer systématiquement notre travail...

### 4.8.1 Fusion

`git merge` : Exception à l'unique parent

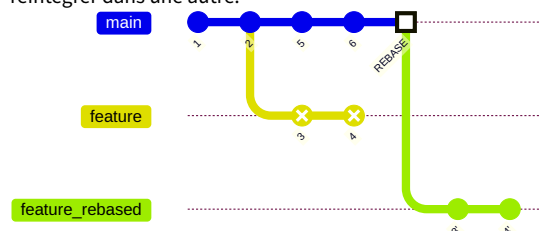
Permet de fusionner les modifications apportées par plusieurs commits issues de deux branches vers une seule même branche.



### 4.8.2 Réintégration

`git rebase` : Réécriture de l'historique

Permet de réécrire l'historique des commits d'une branche pour les réintégrer dans une autre.





## 4.9 Merge

### L'exception à l'unique parent

- Un merge de commits permet de fusionner automatiquement (et intelligemment) les modifications apportées par plusieurs commits issues de deux branches vers une seule même branche.
- Cela permet de travailler simultanément sur plusieurs aspects d'un même projet, puis de rapatrier les différentes déclinaisons du travail en une seule suite logique.
- Après la fusion, un commit de "merge" est créé, qui a deux parents et peut inclure des modifications supplémentaires si des conflits ont été résolus.

Depuis la branche d'arrivée (ici `main`) on utiliserait: `git merge [branch]`



## 4.10 Application

Correction du bug introduit dans le dossier "example\_folder\_2"

Étapes	Commandes
Créez la branche de résolution	<code>git checkout -b fix_pid</code>
???	???
???	???
???	???
Création du commit de correction	<code>git commit -m "Fix PID"</code>
Pour finir	<code>git checkout main</code> <code>git merge fix_pid</code>

## 4.11 Visualisation

```

1 | git log --graph --all --oneline
2 | cat .git/refs/heads/main
3 | cat .git/refs/heads/fix_pid

```

- Toujours nos 40 caractères hachés, qui pointent vers un commit (les têtes de branches)
- Même si la branche `fix_pid` à été fusionnée, elle existe toujours dans l'historique. Elle sera à supprimer manuellement.

```
* commit 316ad6a8927db644ddb435a5e48aadd978b4de20 (HEAD -> main, fix_pid)
Author: Pierre-Yves BRULIN <pierre-yves.brulin@estaca.fr>
Date: Mon Oct 2 13:50:13 2023 +0200

    Fix PID

* commit 643f46198bf0b9e651951ee04fd67e5ca8fb41ba (tag: v1.0.0)
Author: Pierre-Yves BRULIN <pierre-yves.brulin@estaca.fr>
Date: Mon Oct 2 13:44:59 2023 +0200

    Move figure

* commit 1eb0a4d3dbcF374096b7dde9ff9bc6d43e2de226
Author: Pierre-Yves BRULIN <pierre-yves.brulin@estaca.fr>
Date: Mon Oct 2 13:43:06 2023 +0200

    initial commit

* 316ad6a (HEAD -> main, fix_pid) Fix PID
* 643f461 (tag: v1.0.0) Move figure
* 1eb0a4d initial commit
```

## 4.12 Gestion des conflits de fusion

Si vous avez modifié différemment la même partie d'un même fichier dans deux branches que vous souhaitez fusionner, Git ne sera pas capable de réaliser proprement la fusion.

```
1 git merge fix_pid
2 Auto-merging python/script.py
3 CONFLICT (content): Merge conflict in python/script.py
4 Automatic merge failed; fix conflicts and then commit the result.
```

### Comprendre les conflits de fusion :

#### Pourquoi les conflits apparaissent-ils ?

- Deux personnes modifient la même ligne de code
- Suppression d'un fichier d'un côté, modification de l'autre
- Renommage de fichier avec modifications simultanées
- Modifications dans des lignes très proches

#### Anatomie d'un conflit dans le fichier :

```
1 def calculate(self, error):
2     <<<<<< HEAD (Current Change)
3         output = self.kp * error + self.ki * self.integral
4     =====
5         p = self.kp * error
6         output = p + self.ki * self.integral
7     >>>>>> feature-branch (Incoming Change)
8         return output
```

### Stratégies de résolution :

1. **Accept Current** : garder la version de la branche actuelle
2. **Accept Incoming** : garder la version de la branche à fusionner
3. **Accept Both** : combiner les deux modifications
4. **Manual Edit** : réécrire manuellement la section

### Outils pour résoudre les conflits :

- VS Code : interface graphique intégrée
- Git GUI clients : SourceTree, GitKraken, etc.
- Outils de merge spécialisés : Beyond Compare, WinMerge
- Ligne de commande : `git mergetool`

### Prévenir les conflits :

- Communiquer avec l'équipe sur qui travaille sur quoi
- Faire des commits fréquents et petits
- Fusionner régulièrement les modifications de la branche principale
- Utiliser des pull requests pour validation

### 4.13 Création forcée d'un problème de fusion

Créez un conflit de fusion en modifiant la même ligne dans les fichiers `python/script.py` et `matlab/script.m` dans une nouvelle branche `main_cflct` à partir de `v1.0.0`.

Étapes	Commandes
Retour un commit en arrière	<code>git checkout v1.0.0</code>
Création d'une nouvelle branche	<code>git checkout -b main_cflct</code>
Ajoutez la modification Python	<code>(L36) out = min(self.max_clamp, p)</code>
Ajoutez la modification Matlab	<code>(L38) thrust = min(max_clamp, p)</code>
Modification du fichier python	<code>git add .</code>
Indexez et créez le commit	<code>git commit -m "Add max_clamp"</code>
Pour finir intégrez la correction du PID	<code>git merge fix_pid</code>

### 4.14 Gestion des conflits de fusion

```

1 HEAD detached from v1.0.0
2 You have unmerged paths.
3 (fix conflicts and run "git commit")
4 (use "git merge --abort" to abort the merge)
5
6 Unmerged paths:
7 (use "git add <file>..." to mark resolution)
8 both modified: matlab/script.m
9 both modified: python/script.py

```

Lorsque vous allez ouvrir le fichier, il va comporter les deux modifications entourées par les lignes : `<<<<<<`, `=====` et `>>>>>>`

```

35
36 <<<<<< HEAD (Current Change)
37 out = max(self.max_clamp, p)
38 =====
39 out = p + i + d
40 >>>>>> fix_pid (Incoming Change)
41

```

```

36 Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
37 <<<<<< HEAD (Current Change)
38 out = max(self.max_clamp, p)
39 =====
40 out = p + i + d
41 >>>>>> fix_pid (Incoming Change)
42
43 # Save integral for later
44 self.i = 1

```

Resolve in Merge Editor

Current Folder			
	Name	Git	Date Modified
Folder			
	matlab		02/10/2023 14:09
	script.m	!	02/10/2023 14:09
	plot_accurate.png	●	02/10/2023 13:48
	python		02/10/2023 14:09
	script.py	!	02/10/2023 14:09
	__init__.py	●	02/10/2023 13:40
	plots	·	02/10/2023 13:48

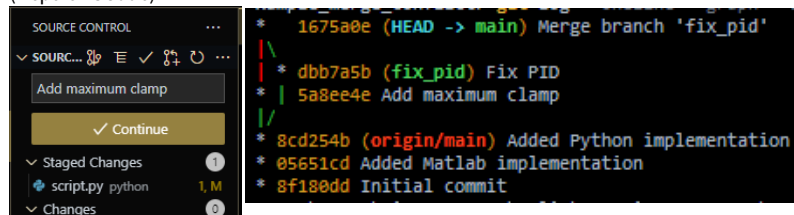
Certains IDE proposent des outils d'aide aux conflits de fusion.

C'est le cas de VSCode. (Ce n'est pas le cas de Matlab...)

Auquel cas, l'IDE peut vous proposer différents choix de résolution de conflits possibles.

## 4.15 Gestion des conflits de fusion

(Depuis VSCode)



Une fois la modification effectuée:

```
1 | git add [script]
2 | git commit
```

Git va essayer d'ouvrir un éditeur de texte (VSCode si configuré lors de l'installation, ou Vim par défaut) pour éditer le message du nouveau commit.

Si VIM s'ouvre, tapez : `:qa!` puis entrée

## 4.16 En cas de problème et pour abandonner une fusion en cours:

```
1 | HEAD detached from v1.0.0
2 | You have unmerged paths.
3 | (fix conflicts and run "git commit")
4 | (use "git merge --abort" to abort the merge)
```

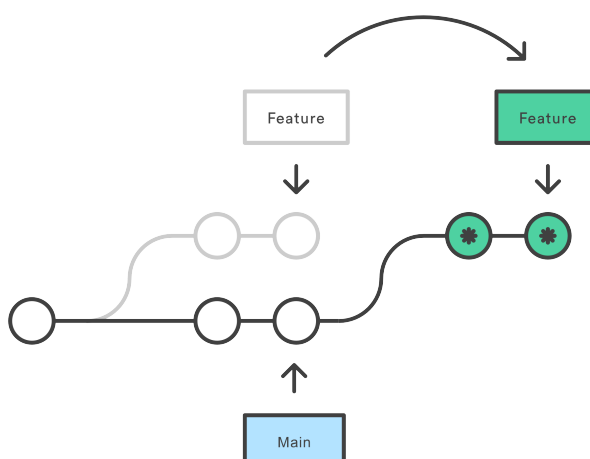
Pour abandonner la fusion en cours:

```
1 | git merge --abort
```

## 4.17 Réintégrer une branche

Réécrire l'histoire: `git rebase`

- Il s'agit d'appliquer une suite de commits d'une branche A (très souvent la branche de production `main`) en tant que parent de la branche B (`development/feature`). Dans le cas où ces deux branches ont divergé.
- Utilisé pour garantir un environnement de développement linéaire et éviter les conflits.
- S'il y a des conflits qui apparaissent lors du rapatriement, ils doivent être résolus au cas par cas.
- Il faut également être attentif à ne pas réintroduire des bugs qui auraient corrigés dans la branche principale.



**Rebase vs Merge - Quelle approche choisir ?**

**Merge (fusion) :**

```

1 | A---B---C feature
2 | /       \
3 | D---E---F---G---H main

```

**Avantages :**

- Préserve l'histoire exacte du développement
- Montre quand les branches ont été créées et fusionnées
- Sûr pour les branches partagées
- Commit de merge explicite avec message descriptif

**Rebase (réintégration) :**

```

1 | D---E---F---A'---B'---C' main

```

**Avantages :**

- Historique linéaire et propre
- Plus facile à comprendre et à naviguer
- Pas de commits de merge "polluants"
- Idéal pour de petites fonctionnalités

**Règles d'usage :****Utilisez MERGE quand :**

- La branche a été partagée avec d'autres développeurs
- Vous voulez préserver l'historique exact
- C'est une fonctionnalité importante qui mérite d'être documentée
- Vous travaillez sur des branches de longue durée

**Utilisez REBASE quand :**

- C'est une branche de travail personnelle
- Vous voulez un historique propre
- Les commits sont petits et logiquement liés
- Avant de partager votre travail (rebase local)

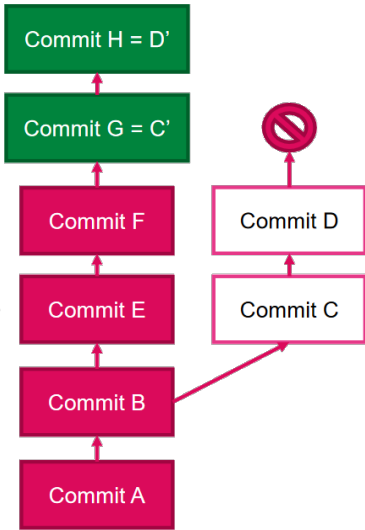
**Workflow populaire :**

1. `git rebase` pour maintenir sa branche à jour localement
2. `git merge --no-ff` pour intégrer les fonctionnalités importantes
3. Squash de petits commits avant le push final

Pratique ...*mais*

---

**MAIS**, ne rebasez/réintégrez jamais des branches déjà publiées et utilisées par d'autres personnes. Quand vous rebasez des commits, vous abandonnez les commits existants et vous en créez de nouveaux qui sont similaires, mais différents. (Changement de parent, donc de hash)  
Si d'autres utilisateurs se basent sur les commits de votre précédente branche pour travailler, puis qu'après coup, vous réécrivez ces commits à l'aide de `git rebase`, vous effacez une partie de l'histoire sur laquelle d'autres personnes se basent.



4.18 Application de git rebase

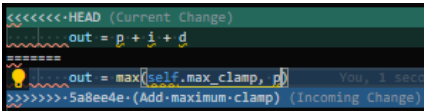
Étapes	Commandes
Repérez le hash du commit <code>max_clamp</code>	<code>git log --all</code>
Déplacement vers le commit de <code>max_clamp</code>	<code>git checkout [hash max_clamp]</code>
Création d'une nouvelle branche pour le rebase	<code>git checkout -b main_rebase</code>
Réécriture de l'historique	<code>git rebase fix_pid</code>

```
>> git log --graph --all --oneline
* acc4c0f (main_cflct) Merge branch 'fix_pid' into main_cflct
| \
|  * 316ad6a (fix_pid) Fix PID
* | 8394a93 (HEAD -> main_rebase) Add max_clamp
| /
* 643f461 (tag: v1.0.0) Move figure
* 1eb0a4d initial commit
```

4.19 Rebase

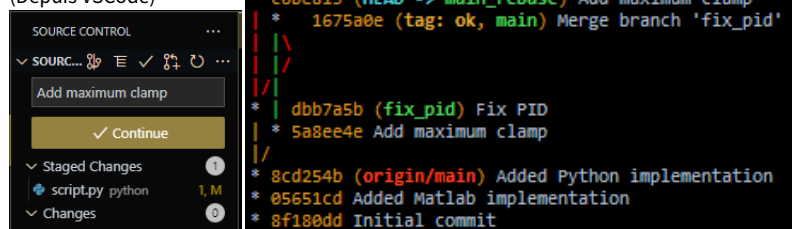
```
>> git rebase fix_pid
Auto-merging matlab/script.m
CONFLICT (content): Merge conflict in matlab/script.m
Auto-merging python/script.py
CONFLICT (content): Merge conflict in python/script.py
error: could not apply 8394a93... Add max_clamp
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 8394a93... Add max_clamp
```

A nouveau un problème de fusion  
(que vous savez corriger)



## 4.20 Gestion des conflits de fusion

(Depuis VSCode)



Une fois la modification effectuée:

```
1 git add [script]
2 git rebase --continue
```

Git va essayer d'ouvrir un éditeur de texte (VSCode si configuré lors de l'installation, ou Vim par défaut) pour éditer le message du nouveau commit.

Si VIM s'ouvre, tapez : `:qa!` puis entrée

## 5 Dépôts distants

### 5.1 Introduction sur les dépôts distants

“Remote” : L'emplacement de stockage de votre dépôt de code.

- Dans le cas de Git qui est décentralisé, un même projet dont vous disposez en local peut-être connecté à plusieurs remote à la fois.
- Votre copie local de travail peut être considérée comme une remote. Il serait possible de spécifier un chemin disque en tant que remote pour y synchroniser vos modifications.

Il nous reste à voir comment interagir avec une remote.

**Concepts fondamentaux des remotes :**

**Remote = dépôt distant :**

- Serveur Git accessible par réseau (HTTP, SSH, Git protocol)
- Peut être GitHub, GitLab, serveur d'entreprise, ou même un autre dossier local
- Chaque remote a un nom (par défaut “origin”)

**Architecture décentralisée :**

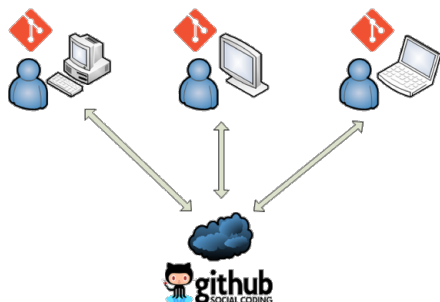
- Chaque développeur a une copie complète du projet
- Pas de “serveur central” obligatoire techniquement
- Mais en pratique, un dépôt fait office de référence (“upstream”)

**Exemples de configurations multi-remotes :**

```
1 # Remote d'origine (votre fork)
2 origin https://github.com/votre-nom/projet.git
3
4 # Remote upstream (projet original)
5 upstream https://github.com/projet-original/projet.git
6
7 # Remote de déploiement
8 production ssh://user@server.com/var/git/projet.git
```

Les principaux dépôts de codes pour Git sont:

- GitHub
- Bitbucket
- GitLab (Open Core)
- GitTea (Open Core)
- etc.



## 5.2 Dépôt de code : Github?

- Github est la première plateforme de développement de code
- Aujourd'hui détenue par Microsoft depuis 2018.
- Github est également une "plateforme sociale", dans le sens qu'elle offre des fonctionnalités tournées autour de la collaboration des développeurs.
- En plus des dépôts de code, la plateforme permet aussi: l'édition de documentation, le suivi des (issues), la gestion des Pull Request (PR), la gestion de plusieurs système d'intégration continue (CI), etc.

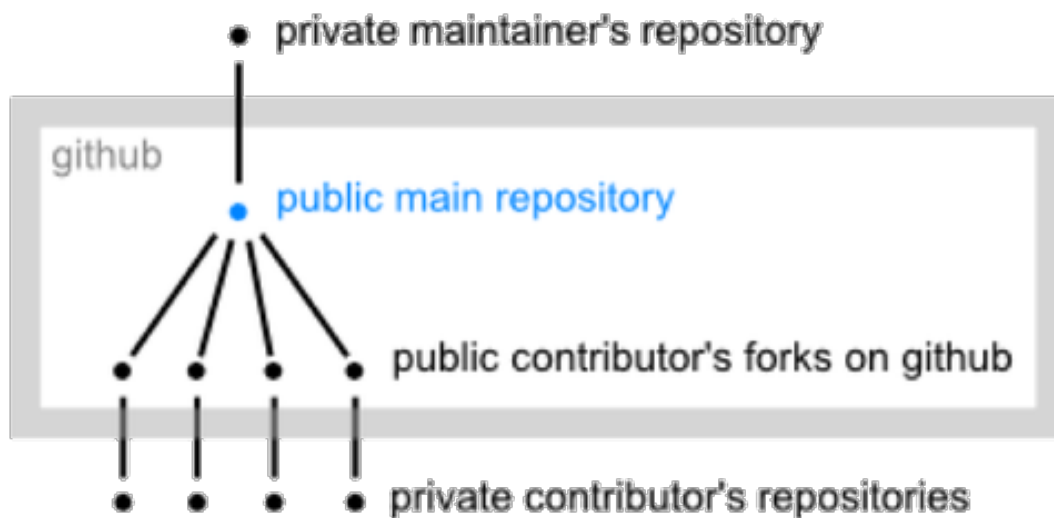


## 5.3 Deux fonctionnalités clés sur Github

### 5.3.1 Fork

Un dépôt principal est dupliqué/"forké" par chaque développeur pour en obtenir une copie spécifique sur son compte, puis cloné sur sa machine locale.





### 5.3.2 Issues

Les issues ("Problèmes" ou "sujets de discussion") permettent de communiquer autour du projet. Elles sont souvent utilisées pour signaler des problèmes ou proposer des idées.

facebook / react

Watch

3,639

Star

51,621

Fork

9,023

<> Code

🔔 Issues 575

🔄 Pull requests 133

📁 Projects 0

📖 Wiki

📊 Pulse

📈 Graphs

Filters

🔍 is:issue is:open

Labels

Milestones

New Issue

🔔 575 Open ✓ 3,196 Closed

Author

Labels

Milestones

Assignee

Sort

🔔 Allow HTML attributes to be dangerously set, without encoding

#7951 opened 20 hours ago by jcfrederico

2

🔔 Fiber Principles: Contributing To Fiber

#7942 opened 2 days ago by sebmahgag

🔔 Making the view difference in the reuse markup error configurable.

#7939 opened 2 days ago by PepijnSenders

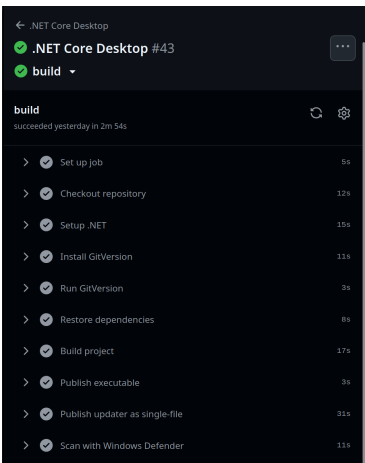
🔔 Use ReactDOM.render target as an implicit JSX root

#7932 opened 3 days ago by yaycmk

5.4 Autres fonctionnalités de Github

Github permet aussi d’autres fonctionnalités:

- Validation automatique des tests unitaires
- La compilation automatique de releases
- Stockage et distribution des releases
- Gestion des équipes
- Permissions utilisateurs
- Gestion des licences
- Etc.



5.5 Application

Interaction avec Github, ou tout autre serveur de dépôt de code.  
Récupérer le projet “example\_folder” cloné au début du cours. Cette fois-ci, il est disponible ici:

```
1 | git clone https://github.com/PYBrulin/example_folder
```

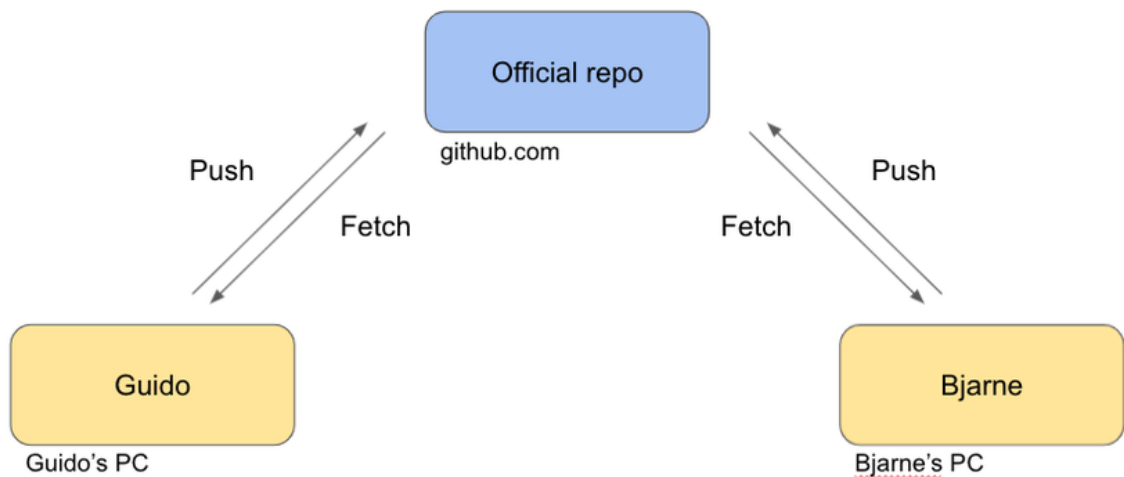
Étapes	Commandes
Cloner un projet	git clone [repository URL]
Liste les dépôts liés au projet	git remote
Liste les infos des dépôts liés	git remote -v
Connecter un dépôt	git remote add [remote] [url]
Déconnecter un dépôt	git remote rm [remote]
Liste des synchronisations actives	git remote show origin
Suivre une branche distante	git checkout -b [branch] [remote]/[branch]

### 5.6 Interagir avec le dépôt

Étapes	Commandes
Télécharge les objets distants	<code>git fetch [remote]</code>
Envoyer les commits d'une branche sur le dépôt	<code>git push [remote] [branch]</code>
Envoyer tout les commits sur le dépôt	<code>git push [remote] --all</code>
Envoyer les commits ET la nouvelle branche sur le dépôt	<code>git push --set-upstream [remote] [branch]</code>
Télécharge les objets distants et les fusionne à la branche actuelle	<code>git pull [remote] [branch]</code>

(Note: `git pull` = `git fetch` + `git merge` )

La figure suivante illustre un workflow Git classique avec un seul dépôt distant : chaque développeur travaille localement, puis pousse ses modifications vers le dépôt officiel sur GitHub. Pour intégrer les changements des autres, il récupère (fetch/pull) les nouveautés du dépôt distant, les fusionne (merge) dans sa branche locale, puis pousse à nouveau ses propres modifications.



### 5.7 Pull Requests (PR)

Les Pulls Requests (PR) sont la formalisation des fusions (merge) sous Github (et sous la plupart des dépôts de codes qui ont suivi cette logique) entre deux copies de travail distinctes, lors du travail entre deux collaborateurs/organisations.

Les PR Pulls Requests (PR) permettent de fusionner deux copies de travail distantes d'une façon amenée pour la collaboration.

Vous pouvez faire le parallèle avec un projet local et un projet distant:

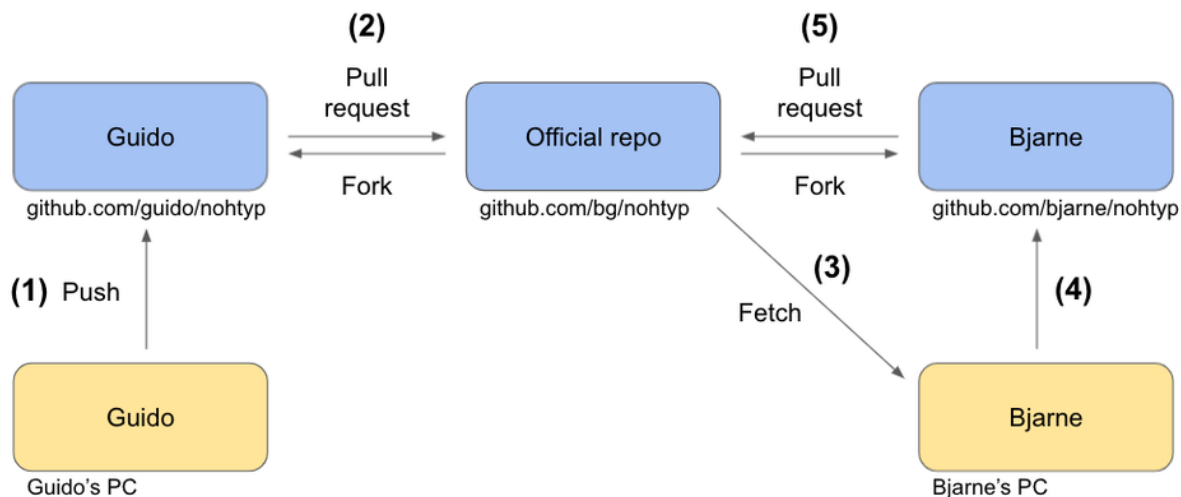
	Type de fusion	Contrôle sur les branches
<b>Local</b>	Merge : fusion de deux branches locales	Vous contrôlez les deux branches
<b>Distant</b>	Pull Request : fusion de deux branches distantes	Vous ne contrôlez qu'une branche. L'administrateur du dépôt distant a le dernier mot sur la fusion.

Avec l'utilisation de plusieurs remotes (ici des forks), le workflow devient plus complexe. mais reste similaire.

Chaque développeur commence par forker le dépôt officiel sur son propre compte, puis pousse ses modifications sur son fork personnel. Pour proposer ses changements au projet principal, il crée une pull request (PR) vers le dépôt officiel.

Les administrateurs du dépôt officiel examinent la PR, demandent éventuellement des modifications, puis fusionnent les changements validés.

Ainsi, personne ne peut modifier directement le dépôt principal : toutes les contributions passent par des PR, ce qui garantit la sécurité et la qualité du projet.



### 5.7.1 Fonctionnement détaillé des Pull Requests

#### Workflow typique d'une PR

1. **Fork** du projet principal vers votre compte
2. **Clone** de votre fork en local
3. **Création** d'une branche pour votre fonctionnalité
4. **Développement** et commits sur cette branche
5. **Push** de la branche vers votre fork
6. **Création** de la Pull Request sur GitHub
7. **Code review** par les mainteneurs
8. **Discussions** et éventuelles modifications
9. **Merge** de la PR dans le projet principal

#### Synchronisation typique

1. `git fetch upstream` : récupère les nouveautés du projet original
2. `git merge upstream/main` : intègre dans votre branche locale
3. `git push origin main` : pousse vers votre fork
4. Pull Request de votre fork vers l'original

#### Avantages des Pull Requests

- **Code review** : révision par les pairs avant intégration
- **Tests automatiques** : CI/CD déclenché automatiquement
- **Documentation** : historique des discussions et décisions
- **Contrôle qualité** : validation avant ajout au code principal
- **Formation** : apprentissage pour les nouveaux contributeurs

#### Bonnes pratiques pour les PR

- Titre clair et descriptif
- Description détaillée des changements
- Commits atomiques et bien nommés
- Tests ajoutés si nécessaire
- Documentation mise à jour
- PR de taille raisonnable (pas trop de changements d'un coup)

**Différence avec d'autres plateformes**

- GitLab : "Merge Requests" (même concept)
- Bitbucket : "Pull Requests" aussi
- Azure DevOps : "Pull Requests"

**5.8 Application Pull Request**

- Regroupez-vous par groupes de 2 (ou 3) étudiants.
- Répartissez-vous entre étudiant A-B(-C) et suivez les consignes spécifiques désignées dans les documents présents sur le réseau.
- Hormis au début des consignes de l'étudiant A ne suivez pas ce que les autres étudiants font, et ne lisez pas les autres consignes. Suivez exclusivement vos consignes.

**Rappel:**

La suite de Syracuse d'un nombre entier  $u_0 > 0$  est définie par récurrence de la façon suivante :

$$\forall u_0 \in \mathbb{N} | u_0 > 0, u_{n+1} = \begin{cases} u_n/2, & \text{si } u_n \text{ pair} \\ 3 \times u_n + 1, & \text{si } u_n \text{ impair} \end{cases} \quad (1)$$

Pour tout  $u_0 \in \mathbb{N} | u_0 > 0$ , il existe un entier  $n$  tel que  $u_n = 1$ .