

# Premiers pas avec git et Github

## Sommaire

1. Présentation .....	1
1.1. Gestion de versions (VCS) .....	1
1.2. Git .....	2
1.3. Github .....	3
2. Mini-projet .....	4
2.1. Visibilité et droits d'accès .....	4
2.2. Démarrer .....	4
2.3. L'espace de travail git .....	5
2.4. Dépôt local vs dépôt distant .....	7
2.5. A quoi sert le dépôt local ? .....	8
2.6. A quoi sert le dépôt distant hébergé ? .....	8
2.7. Configuration de git .....	8
2.8. Cycle de travail .....	9
2.9. Étiqueter des versions .....	10
2.10. Travailler avec des branches .....	10
2.11. Fusion de branche .....	12
2.12. Cycle de travail avec une branche .....	13
2.13. Conflit de fusion .....	13
2.14. Cycle de travail collaboratif .....	13
3. Voir aussi .....	15

Thierry Vaira - <[tvaira@free.fr](mailto:tvaira@free.fr)> - version v0.1 - 27/02/2022 - [btssn-lasalle84.github.io](https://btssn-lasalle84.github.io)

Objectif : Utiliser git et GitHub en mini-projet.

## 1. Présentation

### 1.1. 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**.



Un gestionnaire de version 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 dispositions 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 sources.

## 1.2. Git

**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/>

Ressources :

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

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 commité afin d'effectuer d'autres tâches.

Obtenir de l'aide :

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

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

Aide-mémoire :

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

Simulateurs :

**ExplainGit** est un **simulateur** permettant d'expérimenter visuellement le résultat des commandes qui agissent directement sur le dépôt Git. Il ne simule ni le répertoire de travail ni l'espace d'index, mais uniquement le dépôt.

- (fr) : <https://gauthier.frama.io/explain-git-with-d3/>
- (en) : <https://onlywei.github.io/explain-git-with-d3/>

Il existe aussi [Learn Git Branching \(fr\)](#).

## 1.3. Github

**GitHub** est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git.

D'autres hébergeurs de dépôts git : <https://git.wiki.kernel.org/index.php/GitHosting>

Ressources :

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

## 2. Mini-projet

La section de **BTS SN Lasalle Avignon** dispose d'une organisation ([btssn-lasalle-84](#)) sur Github qui permet d'héberger les dépôts pour les projets et mini-projets.

Lien : <https://github.com/btssn-lasalle-84>

Chaque mini-projet dispose de son propre dépôt sur Github, par exemple : <https://github.com/btssn-lasalle-84/mp12-meteo-2022>

### 2.1. Visibilité et droits d'accès

Seuls les membres d'un mini-projet ont un droit d'accès en **écriture** sur le dépôt.



Les enseignants de la section ont un droit d'administration sur l'ensemble des dépôts.

L'ensemble des dépôts sont en visibilité **privée** pendant toute la durée du mini-projet.



Si le travail final réalisé est de qualité, le dépôt basculera en visibilité publique.

### 2.2. Démarrer



On "travaille" rarement directement sur l'interface web du dépôt hébergé sur Github.

Pour démarrer, il faut tout d'abord récupérer localement le dépôt distant.

Il faut le copier (cloner) avec la commande `git clone` pour obtenir un dépôt local :`

```
$ git clone <depot>
```



Il est possible de cloner le dépôt GitHub de plusieurs manières, notamment :

- en SSH : en utilisant des clés, <https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/connecting-to-github-with-ssh>
- en HTTPS : en utilisant 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>

La commande `git clone` effectue les actions suivantes :

- crée 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.

```
$ cd <depot>

$ ls -al
drwxrwxr-x 8 tv tv 4096 févr. 27 11:25 .git
-rw-rw-r-- 1 tv tv  17 févr. 27 11:25 README.md
```



Pour l'instant, seul le fichier `README.md` est versionné.

```
$ 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
...
```

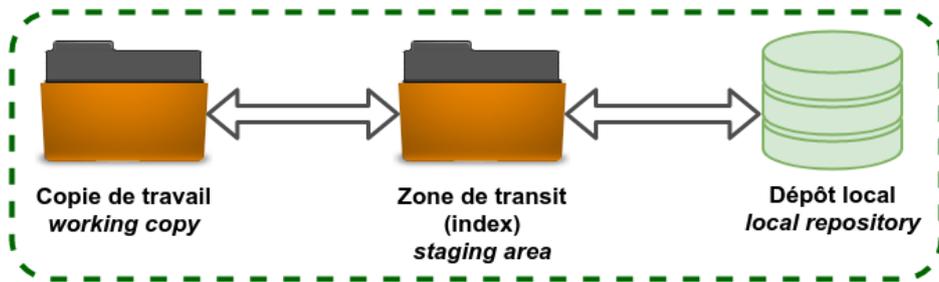


Ne pas "toucher" au répertoire `.git` car il est géré directement par l'outil `git`.

## 2.3. L'espace de travail git

On distingue trois zones :

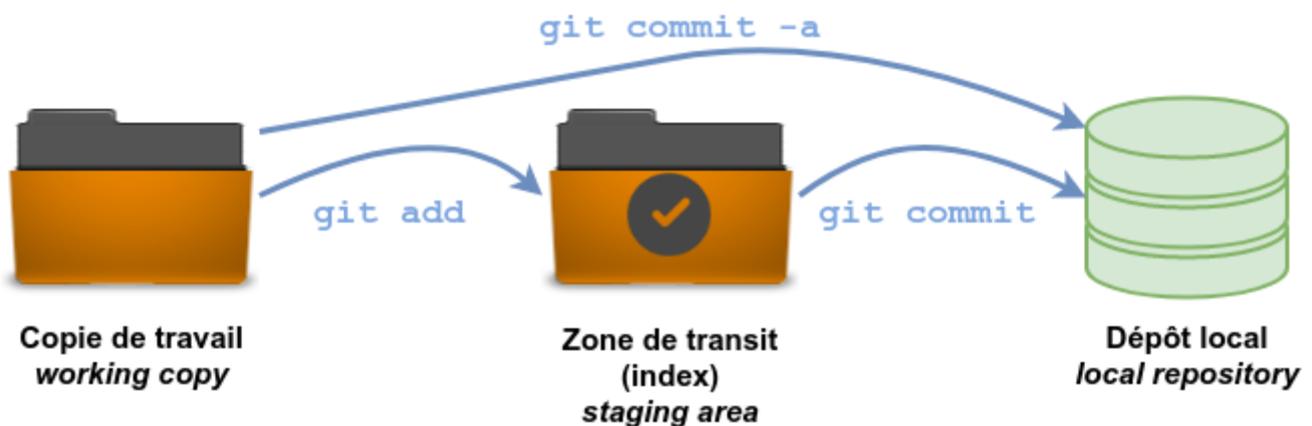
- le **répertoire de travail** (*working directory*) : il contient une extraction unique d'une version du projet pour pouvoir travailler
- l'**index** ou zone de transit (*staging area*) : un simple fichier (ici `.git/index`) qui stocke les informations concernant ce qui fera partie du prochain instantané (*commit*)
- le **dépôt local** (*local repository*) : le répertoire `.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`.

L'utilisation standard de Git se passe comme ceci :

- on édite des fichiers dans le répertoire de travail (*working directory*) ;
- on indexe les fichiers modifiés avec la commande `git add`, ce qui ajoute des instantanés de ces fichiers dans la zone d'index (*staging area*) ;
- on valide les modifications avec la commande `git commit`, 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>

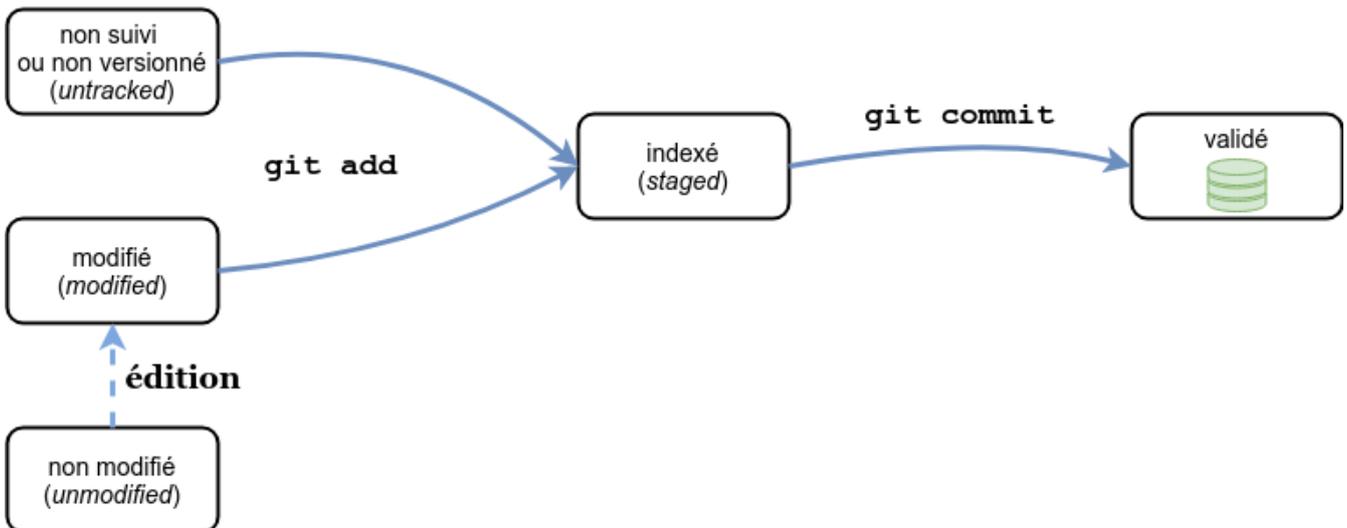


Une fois des changements validés avec la commande `git commit`, le dépôt local n'est plus synchronisé avec le dépôt distant ! Voir [Dépôt local vs dépôt distant](#)

Les différents états d'un fichier :

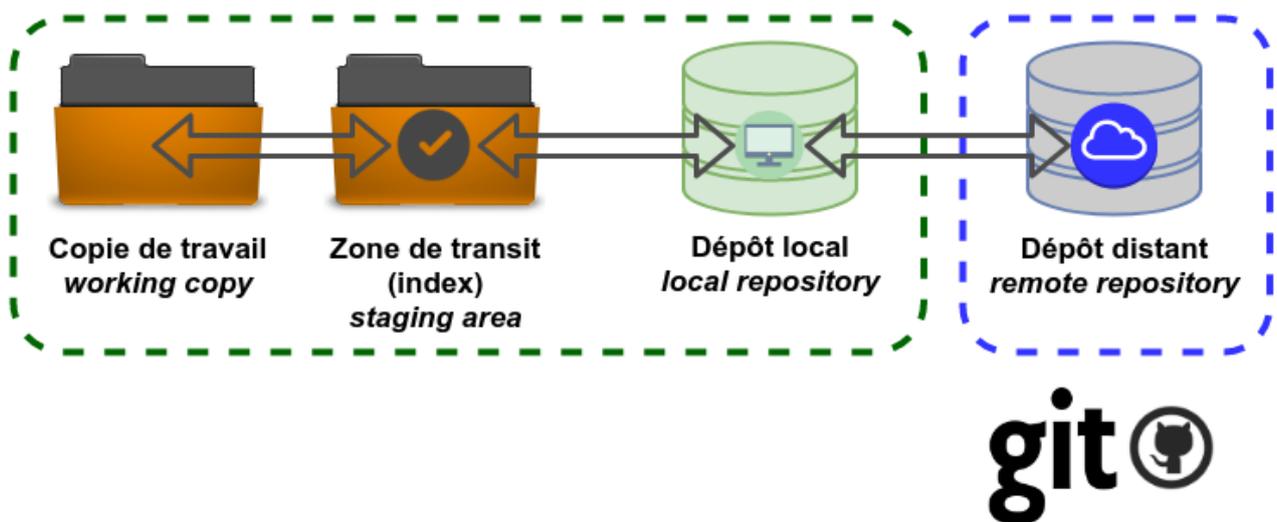
- non suivi ou non versionné (*untracked*) : aucun instantané existe pour ce fichier
- non modifié (*unmodified*) : non modifié depuis le dernier instantané
- modifié (*modified*) : modifié depuis le dernier instantané mais n'a pas été indexé
- indexé (*staged*) : modifié et ajouté dans la zone d'index

- validé : une version particulière d'un fichier



Pour obtenir l'état des fichiers, on utilise (très souvent) la commande `git status`.

## 2.4. Dépôt local vs dépôt distant



Au fur et à mesure de l'évolution du projet, il est normal que les dépôts local et distant ne soient plus synchronisés. Des commandes spécifiques seront utilisées pour gérer cela :

- `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` récupère ("tire") le dépôt distant en effectuant essentiellement un `git fetch` immédiatement suivi par un `git merge` dans la plupart des cas. Le répertoire de travail peut donc être modifié.

## 2.5. A quoi sert le dépôt local ?

Cela permet d'avancer le développement d'un projet de manière décentralisée ("hors connexion") car on dispose de l'ensemble du projet dans son dépôt local.

## 2.6. A quoi sert le dépôt distant hébergé ?

Il centralise l'ensemble des fichiers d'un projet sur Internet.

Cela permet essentiellement :

- de sauvegarder le projet et le rendre accessible depuis Internet
- de partager le projet et donc de travailler collaborativement

## 2.7. Configuration de git

Avant de commencer à "travailler" sur le dépôt, il est préférable de configurer l'outil `git` :

*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 : <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'
```

## 2.8. Cycle de travail



Il faut utiliser régulièrement la commande `git status` qui fournit un guide précieux pour l'utilisation des fonctionnalités de git.

### Sur le dépôt local

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



Il est essentiel de garder en tête que le message de *commit* doit répondre aux questions “Quoi ?” et “Pourquoi ?”, mais surtout pas à la question “Comment ?”. Comme l’objet d’un courriel, (la première ligne d’)un message de *commit* doit donc décrire brièvement (en 50 caractères max) **ce qui a été fait** (un résumé des changements introduits). Utiliser uniquement le présent des verbes et l’impératif. Commencer par une majuscule et pas de point final.

### Vers le dépôt distant

- Envoyer les changements (`git push`)

## 2.9. 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.

Pour les versions, on utilisera des étiquettes annotées.

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>`

## 2.10. Travailler avec des branches

**Créer une branche** signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne.

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



La branche **master** ou **main** n'est pas une branche spéciale. Elle est identique à toutes les autres branches.

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.

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>`.



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.

## En résumé

On crée des nouvelles branches 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.

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 locale existe avant la branche distante, il faudra utiliser l'option `-u` ou `--set-upstream-to` de `git push` pour créer le suivi.

Si on se trouve sur une branche de suivi :

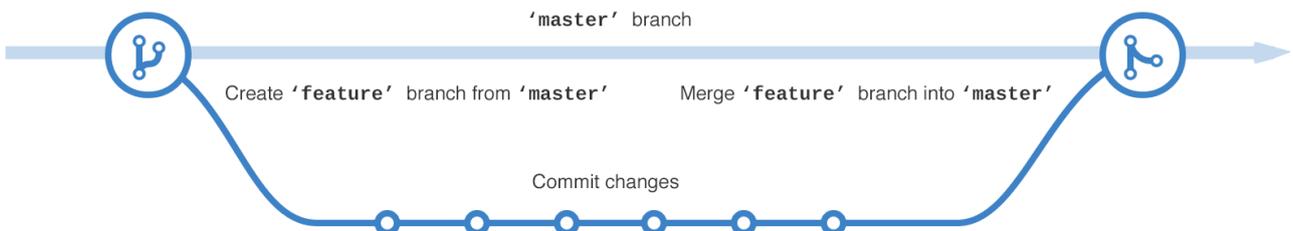
- `git push` sélectionne automatiquement le serveur vers lequel pousser les modifications.
- `git pull` récupère toutes les références distantes et fusionne automatiquement la branche distante correspondante dans la branche actuelle.

## 2.11. Fusion de branche

Une fois le travail réalisé (terminé et testé) dans la branche, il est prêt à être fusionné dans la branche principale.

La fusion permet d'implémenter les modifications apportées à la branche dans une autre branche (souvent la branche principale).

On réalise ceci au moyen de la commande `git merge` qui gère la fusion des changements.



Généralement une fois le travail fusionné, on n'a plus besoin de cette branche locale. On peut la supprimer avec l'option `-d` de la commande `git branch`.



Pour supprimer une branche distante :

```
$ git push origin --delete <branche>
```

## En résumé

En gardant une branche principale 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.

## 2.12. Cycle de travail avec une branche

- Créer une branche thématique et basculer dessus (`git branch <branche>` puis `git checkout <branche>` ou `git checkout -b <branche>`)
  - Éditer des fichiers
  - Ajouter les changements (`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 complémentaires à utiliser :

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

## 2.13. 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.

Lorsque Git rencontre un conflit au cours d'une fusion, il l'indique dans les fichiers concernés 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".

On peut ensuite terminer la fusion en faisant un `git commit` ou en suivant les indications de `git status`.

## 2.14. Cycle de travail collaboratif

Un développement collaboratif avec Github et git s'appuie sur la notion de *Pull Request* et l'utilisation des branches.



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

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

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.

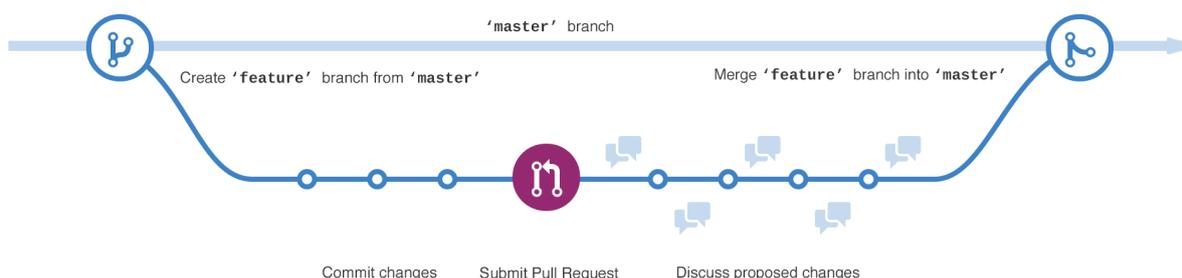
Lien : [Collaborating with pull requests](#)

## 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 informés 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* par exemple).

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. 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.

Généralement une fois le travail fusionné, on n'a plus besoin des branches locale et distante.

- Pour supprimer une branche locale :



```
$ git branch --delete <branche>
```

- Pour supprimer une branche distante :

```
$ git push origin --delete <branche>
```

## 3. Voir aussi

Liens :

- <https://btssn-lasalle84.github.io/guides-developpement-logiciel/git.html> (Version PDF)
- <https://btssn-lasalle84.github.io/guides-developpement-logiciel/jira.html> (Version PDF)

---

Thierry Vaira - <[tvaira@free.fr](mailto:tvaira@free.fr)> - version v0.1 - 27/02/2022 - [btssn-lasalle84.github.io](https://btssn-lasalle84.github.io)