# Table des matières

# GIT

## Qu’est-ce que c’est ?

Git est un logiciel de 'gestion de version' permettant de travailler sur des projets collaboratifs (en groupe).

Cet outil va permettre de partager le travail effectué et d'en faire des 'sauvegardes'.

Un peu à l'image de Google Drive, qui permet à plusieurs contributeurs d’écrire, de modifier et d’ajouter au contenu d’un fichier texte unique, Git permet aux développeurs (et aux équipes IT en général) de collaborer sur un projet.

Git va s'installer sur votre ordinateur et ensuite gérer les versions de votre projet.

## Gestion de Version / Contrôle de version

Git va garder une trace des modifications que vous apportez à tous vos fichiers. Au fur et à mesure, vous allez ajouter des fichiers, les modifier, en enlever. Git va s’occuper de prendre des ‘captures’ (ce sont des sauvegardes à un instant T) de la version actuelle de votre projet. C’est ce qu’on appelle la gestion de version (ou contrôle de version).

Git va sauvegarder ces ‘captures’ dans un ordre chronologique. De ce fait, vous pourrez utiliser git pour naviguer entre les versions de votre projet.

Cela est également très utile si lorsque vous vous trompez : vous pouvez remonter dans le temps jusqu'à la dernière bonne version avant que vous n’ayez corrompu votre code.

## Installer GIT

Git va s'installer sur votre poste.

Si vous utilisez MacOs : https://git-scm.com/download/mac
Si vous utilisez Windows : https://gitforwindows.org/
Si vous utilisez Linux : https://git-scm.com/book/en/v2

Pour vérifier que Git s'est bien installé sur votre machine, tapez cette petite commande :

```
git --version
```

## Premiers pas

Maintenant que Git est installé sur votre machine, vous allez pouvoir commencer !

Tout d'abord, il faut configurer certaines choses.
Tapez ces lignes de commandes :

```
git config --global user.name 'your_Name'
git config --global user.email 'youremail@mail.com'
git config --global color.ui 'auto'
```

Ici, on configure notre nom et adresse mail, ils nous serviront plus tard. Nous disons également à GIT de mettre des couleurs sympas sur les sorties terminal. Le fait que ce soit global signifie que c'est configuré sur votre machine une bonne fois pour toutes. Vous n'aurez donc pas à refaire cette manipulation sur d'autres projets.

Il est également possible de définir des alias.
Par exemple, au lieu de taper 'git commit', je pourrais taper 'git ci'. Voici un exemple avec deux commandes :

```
git config --global alias.ci commit
git config --global alias.st status
```

Maintenant que vous avez configuré GIT, il est temps de créer son projet !

## Créer un *repository*

Vous allez créer un dossier vide. Ce dossier va contenir votre projet.
Dans mon cas je l'appelle '*TestGit'*.

Placez-vous dans ce dossier et tapez cette commande :

***git init***

Cette commande va créer un nouveau *repo git* (ou dépôt git).
Ce repo c'est comme le dossier de votre projet, mais avec les super pouvoirs de Git. Un référentiel contient tous vos fichiers de projet, ainsi que des éléments supplémentaires, principalement invisibles, générés par git pour suivre et stocker l'historique de chaque fichier.

## Ajouter un fichier README

Ce n'est pas obligatoire mais une bonne pratique.
Vous allez commencer par créer un fichier 'README' pour expliquer en quoi consiste votre projet, ce qui pourrait se trouver dedans etc.
Ajoutez ce fichier à votre projet.

## Contrôler le statut du repo

Vous pouvez visualiser l'état courant du dépôt avec la commande :

***git status***

Pour l'instant, le résultat de la commande doit ressembler à ça :

```
On branch master

No commits yet

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

        README.md

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

## Ajouter des fichiers

'git status' nous dit que nous avons de nouveaux fichiers qui n'ont pas été officiellement ajoutés au processus de suivi git.
Pour dire à git d'ajouter et suivre ces fichiers, il faut utiliser la commande :

*git add README.md*

Vous pouvez ajouter tous les fichiers de cette manière (déconseillé) :

*git .*

Maintenant, la commande 'git status' nous affiche ceci :

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
```

## Valider les changements

Maintenant que vos fichiers ont été ajoutés et sont 'suivi' par Git, vous allez pouvoir 'commit'.
Un commit est comme une 'capture' d'un instant où vous allez pouvez revenir, si nécessaire, pour accéder à votre repo dans son état antérieur.

Pour identifier chacune de ces captures, vous devez fournir un message (grâce à l'option –m).

*git commit -m 'premier commit'*

Vous avez maintenant commencé la gestion de version de votre projet en local !

## Choisir un hébergement distant

Maintenant, il est temps d'enregistrer votre projet à distance (en ligne) !
On appelle ça un *repo distant*.
Il existe de nombreux hébergement qui permet de stocker du code. Nous utiliserons ici GitHub.
Nous allons donc sur GitHub pour créer un nouveau repo (pour l'instant vide).
Nous allons ensuite 'lier' notre repo local à notre repo distant.

```
git remote add origin https://github.com/InsteanAzeros/TestGit.git
```

Vous pouvez maintenant 'envoyer' votre code sur GitHub :

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

L'option –u permet d'associer définitivement tous les prochains 'git push' sur la branche *master* du dépôt *origin*.
Il est possible d'avoir plusieurs repos distants, celui-ci sera référencé par *origin*.

## Comparer local et distant

Pour l'exemple, je vais rajouter un fichier *'test.java'* au projet.
Ensuite je l'ajoute avec la commande '*git add *.java'.*
Et je réalise un *'commit'* pour valider mes changements.

Maintenant grâce à la commande status, je peux voir que je suis en avance d'un commit par rapport à mon repo distant :

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

## Les branches

La branche principale de votre repo (*Master*) doit toujours contenir du code fonctionnel et stable.
Cependant, il est courant de bosser sur du code qui n'est pas stable, comme un ajout de features ou du debbugage etc… Pour autant, vous avez besoin de sauvegarder ces modifications. C'est ici que la notion de branche rentre en jeu.

Il est possible de créer des branches qui vous permet de travailler sur une copie distincte de votre projet sans affecter la branche principale. Quand vous allez créer une branche, un clone complet de votre branche principale est créé sous un nouveau nom. Vous allez pouvoir ensuite modifier le code de cette nouvelle branche sans impact sur votre branche principale !

Vous pourrez, quand votre code est stable, fusionner les deux branches !

Tout d'abord, créons une branche de test :

```
git checkout -b branchTest
```

Une nouvelle branche est alors créée et vous êtes automatiquement 'transféré' sur celle-ci.

Essayez cette petite commande :

```
git branch
```

Cette commande va lister les branches disponibles sur votre repo et colorer celle sur laquelle vous vous trouvez :

```
MacBook-Pro-de-Simon:TestGit Instean$ git branch
* branchTest
  master
```

Vous pouvez vous déplacer d'une branche à l'autre avec la commande :

```
git checkout nomdelabranche
```

Maintenant, vous allez modifier le fichier *test.java* et voir ce qu'il se passe avec la commande *'git status'* :

```
On branch branchTest
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.java

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

Vous allez sauvegarder les changements :

```
git commit -am 'modif de test java'
```

L'option –a va automatiquement ajouter les modifications des fichiers connus.

## Fusionner les branches

Maintenant que la branche a été testé et validé, vous allez pouvoir l'intégrer (la fusionner) à la branche principale.
Pour cela, on va d'abord se positionner sur la branche principale et lui dire qu'elle est la branche avec laquelle fusionner.

```
git checkout master
git merge branchTest
```

Maintenant que notre branche de développement a été fusionné, elle est devenue 'inutile'. On va donc la détruire. L'historique des modifications ne sera pas perdu même en détruisant la branche.

```
git branch -d branchTest
```

Maintenant que notre repo contient une version stable, nous allons pouvoir le transférer au repo distant :

```
git push
```

## Cloner un repo distant

Si vous voulez récupérer le contenu d'un repo distant, par exemple si vous rejoignez un projet en cours, utilisez cette commande :

```
git clone https://github.com/InsteanAzeros/TestGit
```

## Les release

Avec git, il est possible de publier des versions spécifiques de votre code. Une fois la release crée, la branche perd la possibilité de changer l'historique des commits.

Pour publier une release on utilise des tags.
On va taguer des branches et c'est ce mécanisme qui permet de gérer simplement les releases.

```
git tag -a v1.0 -m "Release 1.0"
```

l'option –a crée un 'tag annoté'.
L'option –m spécifie le message du tag, qui es stocké avec le tag.

Afficher les tags :

```
git tag
```

Vous pouvez voir les détails d'un tag :

*git show v1.0*

Vous devriez voir quelque chose de semblable :

```
commit 43ae84ba76aadc13a98c0493434d85a16ba17c25 (HEAD -> master, tag: v1.0, orig
in/master, branchTest)
Author: Simon Bertrand <fctinstean@gmail.com>
Date:   Wed Apr 17 16:53:14 2019 +0200

    modif de test java

diff --git a/test.java b/test.java
index 608c659..7f1fb48 100644
--- a/test.java
+++ b/test.java
@@ -1,5 +1,5 @@
 package State;

 public abstract class test {
-       public abstract void printStatus(Context context);
+       public abstract void print(Context context);
 }
```

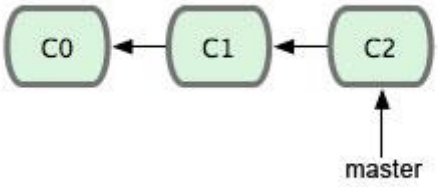
Il est également possible de taguer sans annoter, comme ceci :

*git tag -a v1.2*

l'option –a crée un 'tag annoté'.
L'option –m spécifie le message du tag

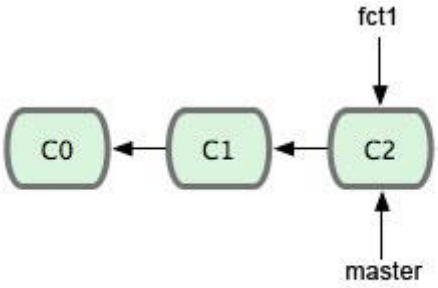
## Comprendre un peu mieux les branches

Dans cette partie, nous allons illustrer un exemple d'utilisation de branche. Imaginons un projet où il n'y a que la branche Master et déjà 3 commits.



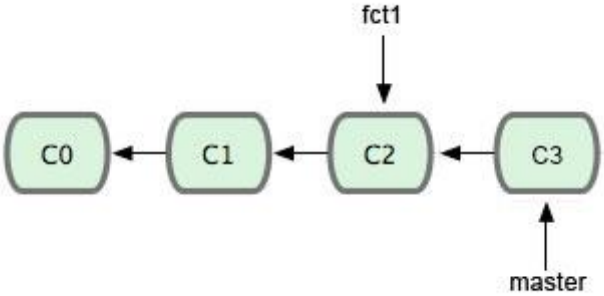

C0, C1 et C2 représente les commits.

Maintenant, nous avons besoin de développer une nouvelle fonctionnalité. Pour cela, on crée une nouvelle branche 'fct1' (pour fonctionnalité numéro 1).

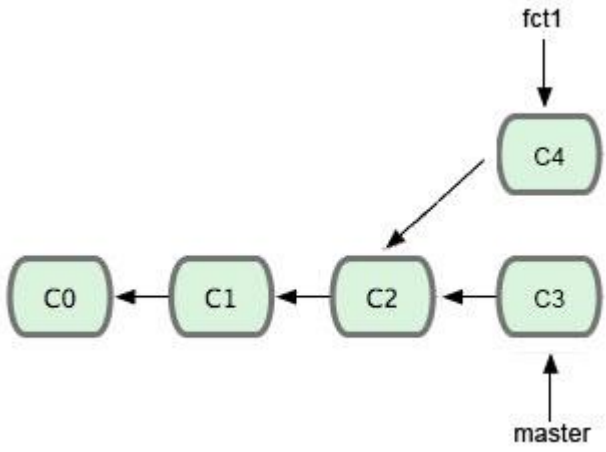



Git crée un pointeur vers le commit C2.

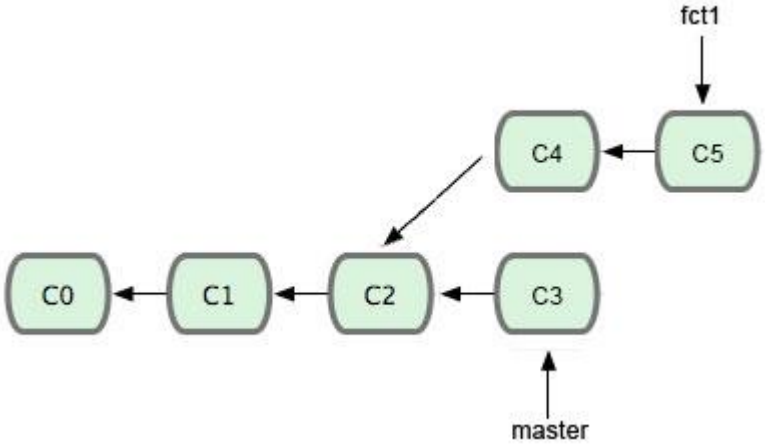
Ensuite, on va se positionner sur le master, faire un changement et commit.



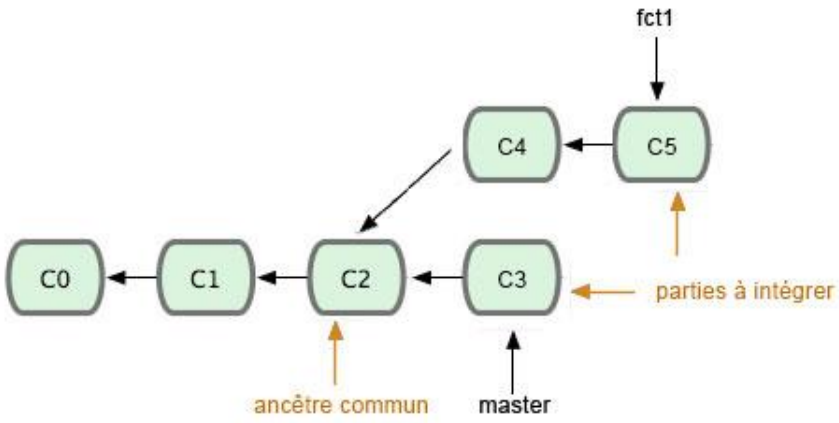
On va faire pareil sur la branche 'fct1', changement et commit.




On va effectuer un deuxième changement et commit sur la branche (fct1).



Notre fonctionnalité est maintenant opérationnelle, on va vouloir merge la branche fct1 dans le master.




Git recherche la racine commune (ici C2) pour intégrer les commits un par un et vérifier les conflits par itérations à partir de cette racine.

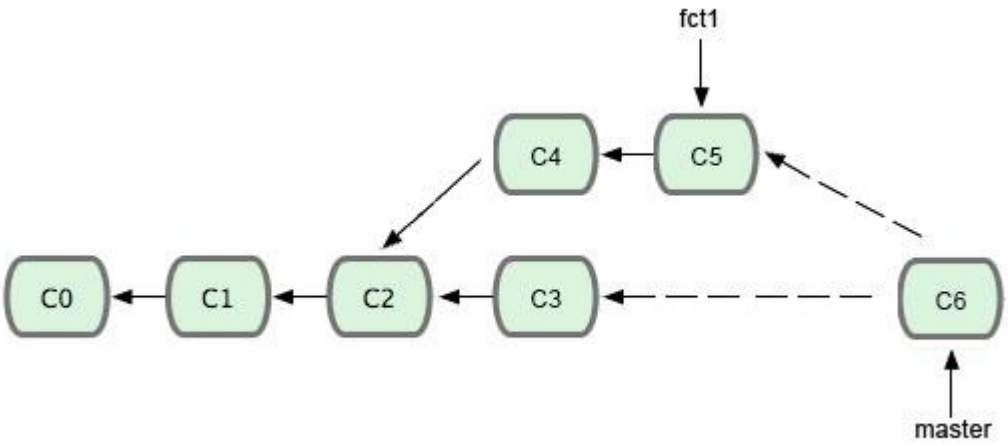



Illustration inspiré de : [https://git-scm.com/book/en/v2](https://git-scm.com/book/en/v2)

## Bonne(s) pratique(s)

git ignore

Attention à ne pas versionner n'importe quoi !
C'est vous qui choisissez le(s) fichier(s) à versionner dans votre projet.
Vous pouvez créer un fichier '.gitignore' pour empêcher l'ajout de fichiers indésirables.

Généralement, on ne versionne pas les exécutables, les images, les binaires etc…

Standard

Il existe plusieurs standards quant à la gestion de vos projets sous git.
En voici un.

*Les branches principales :*

Deux branches principales avec une durée de vie infinie : Master et Develop.

Master est une branche toujours stable et sert au déploiement.
Develop est la branche de travail qui contient la dernière version du code en cours.
Elle contient donc les dernières modifications de développement pour la prochaine version.

Lorsque le code de la branche Develop atteint un point stable et est prêt à être déployé, tous les changements doivent être fusionnés dans le master puis taguer avec un numéro de release.

*Les branches secondaires :*

En plus des branches principales *master* et *develop*, on peut utiliser une variété de branches secondaires pour faciliter le développement parallèle entre les membres de l'équipe, faciliter le suivi des fonctionnalités, préparer les versions de production et aider à résoudre rapidement les problèmes de production réelle.

Contrairement aux branches principales, ces branches ont toujours une durée de vie limitée, car elles seront éventuellement supprimées.

Les différents types de branches fréquemment utilisées :
> Feature branches
> Release branches
> Hotfix branches

> Feature branches
Les features branches sont utilisées pour développer de nouvelles fonctionnalités pour la version à venir ou future.
La branche existe tant que la fonctionnalité est en développement mais elle sera finalement fusionnée à Develop (pour ajouter définitivement la nouvelle fonctionnalité à la version à venir) ou supprimée (en cas de test décevant).

> Release branches
Les release branches prennent en charge la préparation d'une nouvelle version de production.
En outre, ils permettent de corriger des bugs mineurs et de préparer des métadonnées pour une version (numéro de version, dates de construction, etc.).

Le moment clé pour 'sortir' de develop et créer une release branch est le moment où Develop reflète (presque) l'état souhaité de la nouvelle version.
Au moins toutes les fonctionnalités qui sont ciblées pour la version à construire doivent être fusionnées pour se développer à ce stade.

> Hotfix branches
Les branches Hotfix ressemblent beaucoup aux branches de release dans la mesure où elles sont également destinées à préparer une nouvelle version de production, bien que non planifiée. Ils découlent de la nécessité d'agir immédiatement sur un état non souhaité d'une version en production.
Lorsqu'un bug critique dans une version de production doit être résolu immédiatement, une branche de correctif logiciel (hotfix) est créée.

SOURCE PHOTO : https://nvie.com/posts/a-successful-git-branching-model/