Git

Table des matières

Guillaume RODRIGUES – 2024

Git

Présentation

Git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre et gratuit, créé en 2005 par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. Le principal contributeur actuel de Git, et ce depuis plus de 16 ans, est Junio C Hamano.

Depuis les années 2010, il s'agit du logiciel de gestion de versions le plus populaire dans le développement logiciel et web, qui est utilisé par des dizaines de millions de personnes, sur tous les environnements (Windows, Mac, Linux)3. Git est aussi le système à la base du célèbre site web GitHub, le plus important hébergeur de code informatique.

Wikipedia

Git est un outil de versionning. Il permet de conserver toutes les modifications faites sur des fichiers de tous types.

Il est souvent utilisé conjointement avec Github ou Gitlab, qui permettent de conserver les dépôts sur des serveurs distants et ainsi permettre le travail collaboratif.

Installation

Windows

Télécharger l'installeur ici : https://git-scm.com/download/win

Lorsque l'installateur vous demandera de choisir un éditeur de texte, je vous conseille Nano, qui sera plus simple et largement suffisant pour l'utilisation qu'on en fera que les autres solutions.

Lorsque l'installateur vous demandera de choisir le mode de gestion des sauts de ligne, je vous conseille de choisir : « Checkout as−is, commit Unix−style line endings » pour une meilleure compatibilité sur les systèmes Unix.

Linux

Sous Linux, il suffit généralement d'installer la version de Git disponible dans les dépôts de votre distribution : « apt install » git pour les distris debian-like

Git

Utilisation de Git bash

Commandes linux de base :

- Commande « **cd** » : changer de répertoire

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~
$ cd chemin/vers/mon/repertoire
```

- Commande « mkdir » : créer un répertoire

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~
$ mkdir chemin/vers/mon/repertoire
```

- Commande « touch » : créer un fichier vide

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~
$ touch mon_fichier
```

- Commande « ls » : lister le contenu d'un dossier

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ ls
Dockerfile  LICENSE  README.md  README_XDebug_config.md  compose.yaml

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ ls -la
total 38
drwxr-xr-x 1 guillaumerodrigues 1049089    0 Oct 17 10:27 ./
drwxr-xr-x 1 guillaumerodrigues 1049089    0 Oct 23 11:04 ../
drwxr-xr-x 1 guillaumerodrigues 1049089    0 Oct 17 10:27 .git/
-rw-r--r-- 1 guillaumerodrigues 1049089    1 Sep 29 09:41 .gitignore
-rw-r--r-- 1 guillaumerodrigues 1049089  937 Oct 17 10:27 Dockerfile
-rw-r--r-- 1 guillaumerodrigues 1049089 1063 Sep 29 09:42 LICENSE
-rw-r--r-- 1 guillaumerodrigues 1049089  620 Sep 29 09:42 README.md
-rw-r--r-- 1 guillaumerodrigues 1049089 4238 Sep 29 11:57 README_XDebug_config.m
d
-rw-r--r-- 1 guillaumerodrigues 1049089  691 Oct 17 10:27 compose.yaml
```

Git

- Commande « rm » : suppression d'un fichier ou répertoire (avec l'option -R)

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ touch test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ ls
Dockerfile  LICENSE  README.md  README_XDebug_config.md  compose.yaml  test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ rm test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ ls
Dockerfile  LICENSE  README.md  README_XDebug_config.md  compose.yaml
```

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ mkdir test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ ls
Dockerfile  LICENSE  README.md  README_XDebug_config.md  compose.yaml  test/

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ rm -R test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/docker-lamp (main)
$ ls
Dockerfile  LICENSE  README.md  README_XDebug_config.md  compose.yaml
```

Configuration

Nom d'utilisateur du développeur

```
$ git config --global user.name "nom_utilisateur"
```

Adresse mail du développeur

```
$ git config --global user.email mail_utilisateur
```

Vérifier sa configuration :

```
$ git config --list
```

Git

Activation de la coloration (facultatif)

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

Configurer son éditeur et un outil de merge (facultatif)

```
$ git config --global core.editor nano
$ git config --global core.editor code
$ git config --global merge.tool vimdiff
```

Initialisation d'un dépôt

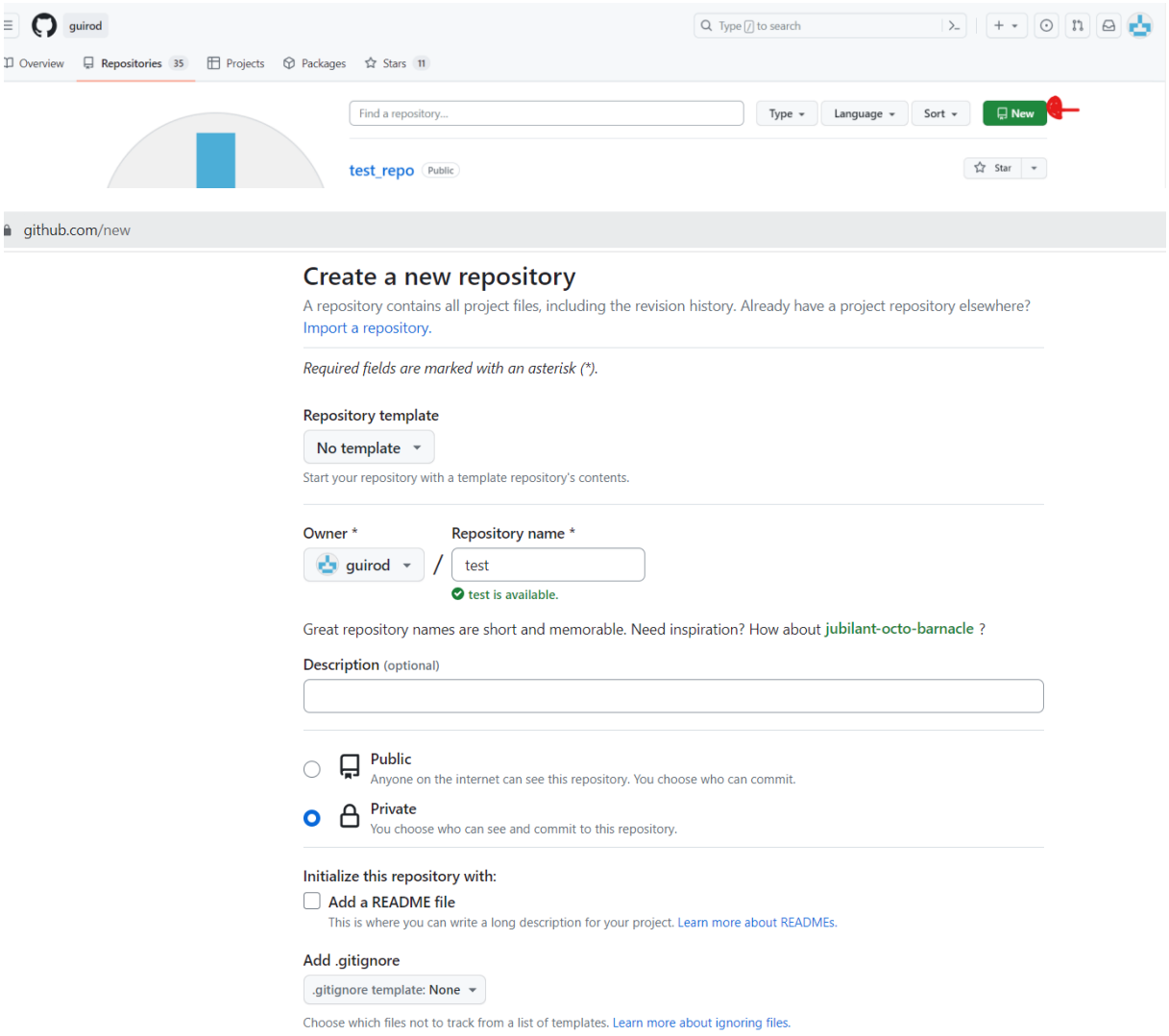
git init

Se placer dans le répertoire de notre projet et taper la commande « git init » permettra d'initialiser un dépôt.

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace
$ mkdir test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace
$ cd test

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test
$ git init
Initialized empty Git repository in C:/Users/guillaumerodrigues/Workspace/test/.git/
```

Git

Bien souvent, il faudra ensuite configurer notre dépôt local pour qu'il suive notre dépôt distant. Cela permettra de sauvegarder et partager nos modifications.



Git

Puis suivez les instructions indiquées sur le site Github :

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test (main)
$ git remote add origin https://github.com/guirod/test.git
```

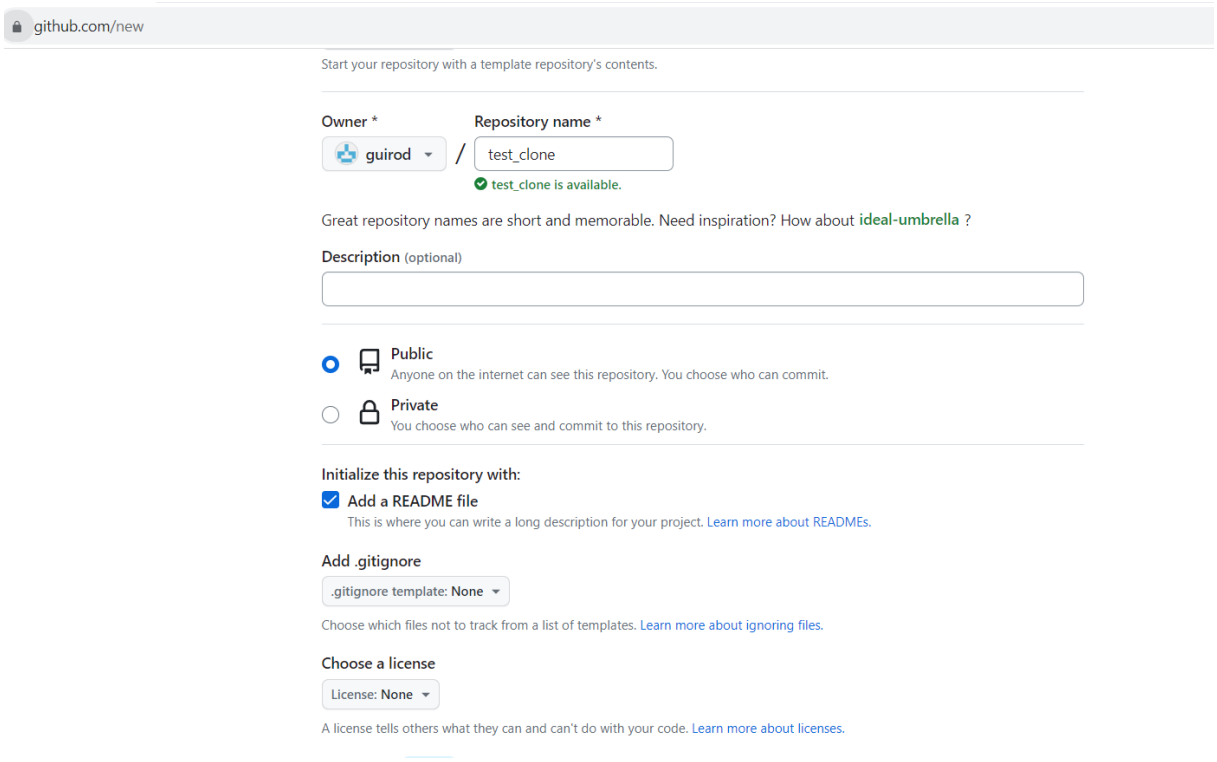
```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test (main)
$ touch readme.md

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test (main)
$ git add .

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test (main)
$ git commit -m "first commit"
[main (root-commit) bac53dc] first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 readme.md
```

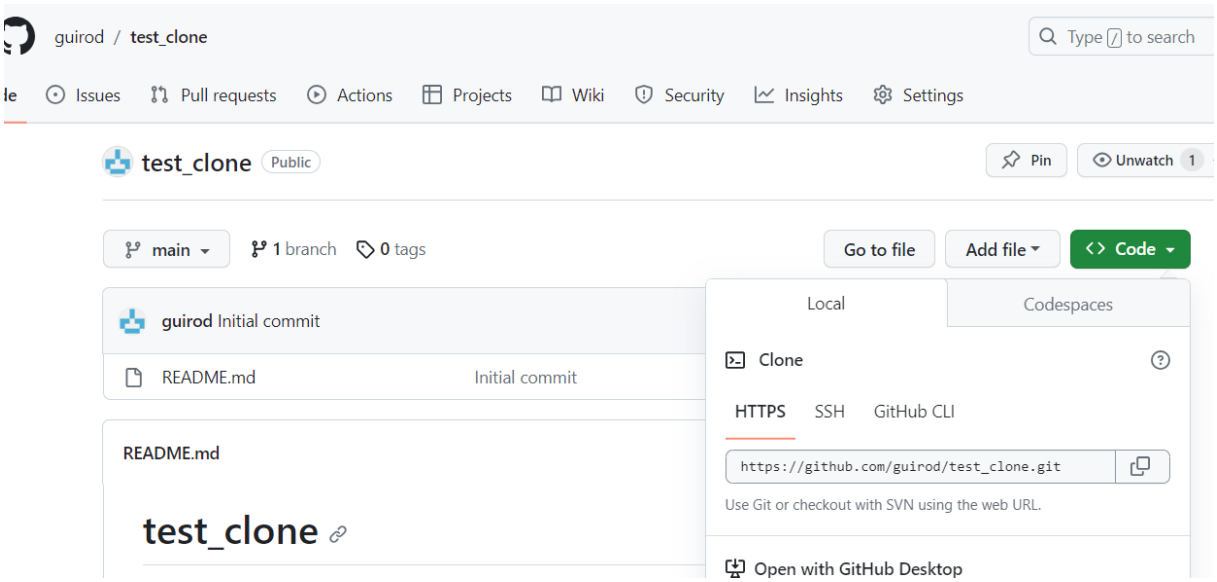
```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test (main)
$ git push --set-upstream origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 208 bytes | 29.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/guirod/test.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

Git

git clone

Pour un nouveau projet, le moyen le plus simple de mettre en relation un dépôt distant avec un dépôt local est de le créer d'abord sur Github, en y ajoutant un fichier à la création (README.md par exemple), puis de cloner le dépôt dans notre espace de travail.





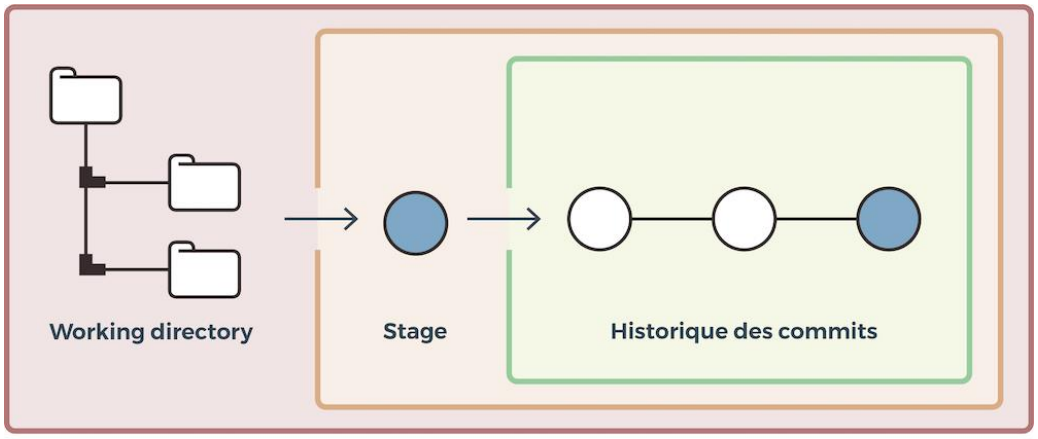
Puis dans le dossier de votre espace de travail :

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace
$ git clone https://github.com/guirod/test_clone.git
Cloning into 'test_clone'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Git

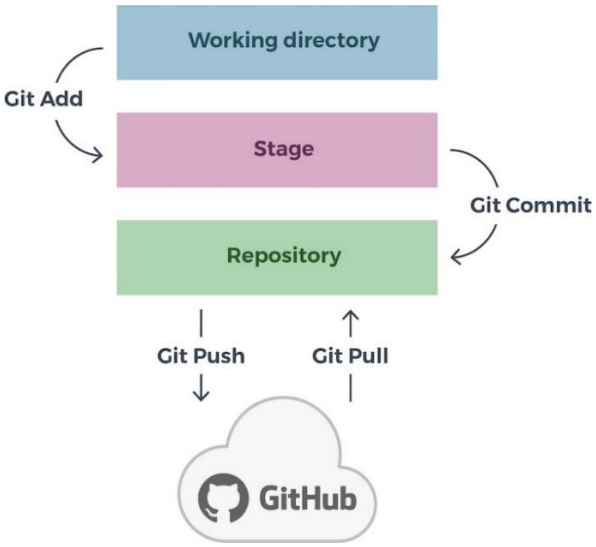
Le système de sauvegarde (commit) :

Présentation du système de sauvegarde de git :



Git gère les versions de nos travaux locaux à travers 3 zones locales majeures :

- **Le répertoire de travail** (working directory/WD) ;
- **L'index, ou *stage*** (File d'attente) ;
- **Le dépôt local** (Git directory/repository **commit**).



Git

Mettre à jour le dépôt local avec le dépôt distant

git fetch

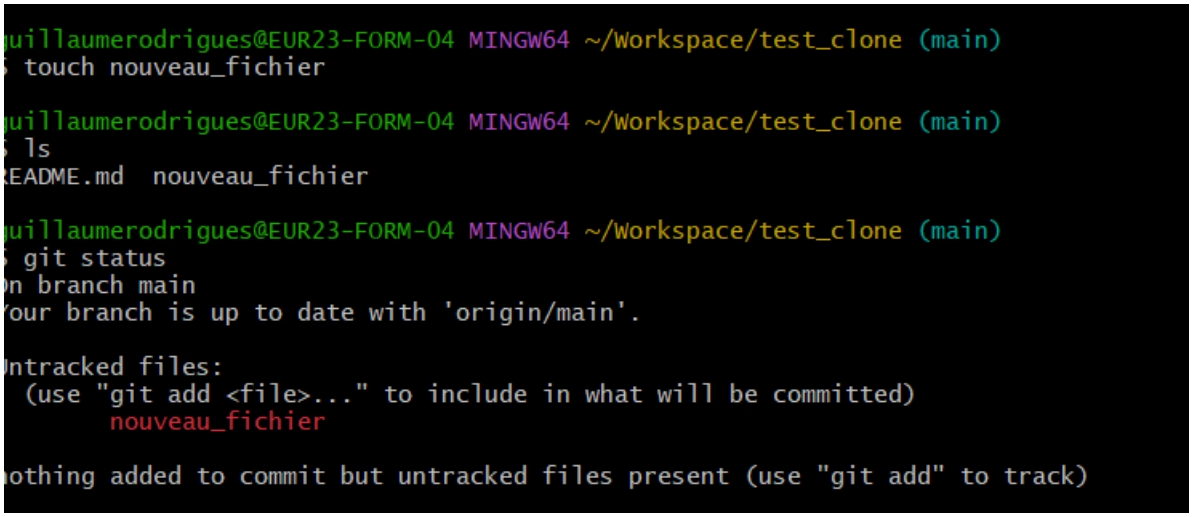
La commande « git fetch » permet d'interroger le dépôt distant pour y lister les modifications qui ont été faites récemment. Lorsque l'on travaille en collaboratif, il est important de faire cette commande assez régulièrement.

git pull

La commande « git pull » va d'abord réaliser un « git fetch », puis essayera de télécharger les modifications depuis le serveur distant. Là encore, il est important de le faire assez régulièrement, mais attention, cette fois ci les fichiers sont téléchargés dans le dépôt local, ce qui peut parfois causer des conflits.

git status

La commande « git status » permet de vérifier l'état du dépôt local par rapport au dépôt distant.



Ici, on voit que nous sommes à jour avec le dépôt distant, mais que nous avons des « untracked files » : des fichiers non-suivis. Il va être nécessaire d'ajouter les fichiers que l'on souhaite versionner.

Git

git add

Lorsque l'on a travaillé sur notre projet, on souhaitera ajouter des fichiers au système de suivi de versions. Pour cela il faut utiliser la commande « git add ».

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git add nouveau_fichier

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   nouveau_fichier
```

Si nous souhaitons ajouter tous les fichiers non encore suivis, nous pouvons utiliser « git add . » :

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ touch nouveau_fichier1

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ touch nouveau_fichier2

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ touch nouveau_fichier3

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   nouveau_fichier

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        nouveau_fichier1
        nouveau_fichier2
        nouveau_fichier3


guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git add .

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   nouveau_fichier
        new file:   nouveau_fichier1
        new file:   nouveau_fichier2
        new file:   nouveau_fichier3
```

Git

git commit

Une fois les fichiers ajoutés au suivi, nous pouvons les « commit » afin de valider les changements.

L'option « -m » permet d'ajouter un message de commit. Si cette option n'est pas précisée, un éditeur de texte sera ouvert pour que nous puissions le renseigner.

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git commit -m "my 1st commit"
[main 869ad50] my 1st commit
 4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 nouveau_fichier
 create mode 100644 nouveau_fichier1
 create mode 100644 nouveau_fichier2
 create mode 100644 nouveau_fichier3

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$
```

« git status » nous informe maintenant que notre branche est en avance d'un commit par rapport à la branche distant. Il va falloir pousser nos commits sur le dépôt distant.

git push

Pour pousser nos commits, il faut utiliser la commande « git push ».

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 289 bytes | 32.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/guirod/test_clone.git
   38e3f7a..869ad50  main -> main

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$
```

Git

Fichier gitignore

Le fichier « .gitignore » est un fichier que vous devrez ajouter à la plupart de vos dépôts git, à la racine de votre projet. Il permet de lister les fichiers et répertoires qui seront ignorés par git, et qui ne seront donc pas versionnés.

Ce fichier permettra ainsi d'ignorer :

- Les fichiers de configuration contenant des informations sensibles (mots de passes, clés d'API, adresses mail, …)
- Les librairies externes, qui ne feraient qu'alourdir le projet et qui sont souvent déjà versionnées de leur côté
- Les répertoires et fichiers de configuration de l'éditeur de texte que nous utilisons (PHPStorm, vscode, …)

A la création d'un dépôt sur Github, il nous proposera de créer un template de .gitignore en fonction du framework utilisé (Symfony, VueJS, …). Ca peut être une bonne base de départ.

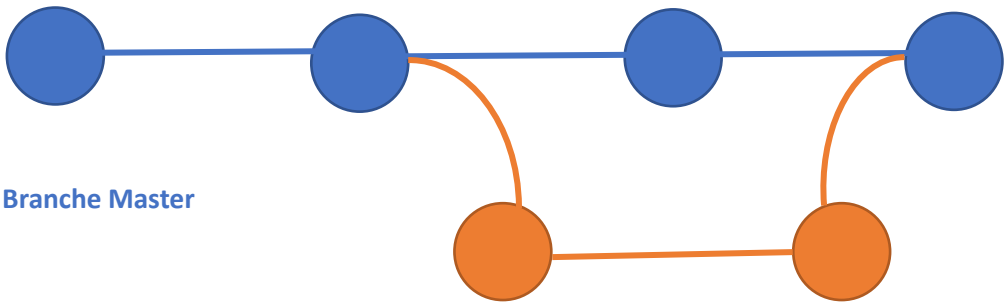
Les branches

Une des forces de Git est sa gestion de différentes branches de travail sur un même projet. Il faut voir le système de branche comme des « dossiers virtuel ».

Ces branches vont permettre de sauvegarder des versions différentes du code, et donc de travailler sur des branches en ayant le moins d'interactions possibles avec les branches principales du projet.

Souvent, on a une branche principale (main ou master) sur laquelle on ne devrait pousser que les modifications stables et testées.

Nous aurons parfois des branches d'intégration, et des branches dédiées aux nouvelles features ou bugfix.



Branche Master

Git

Pour lister les branches, on utilisera la commande : « **git branch** »

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git branch
* main
```

Pour sélectionner une autre branche afin de travailler dessus, on utilisera la commande « **git checkout nom_de_la_branche** ».

Pour créer une branche, il faudra utiliser la commande « **git branch nom_branche** ».

Nous pouvons aussi utiliser la commande « **git checkout** » pour créer une nouvelle branche et se positionner dessus : « **git checkout -b nouvelle_branche** »

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git checkout -b nouvelle_branche
Switched to a new branch 'nouvelle_branche'

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (nouvelle_branche)
$ git status
On branch nouvelle_branche
nothing to commit, working tree clean

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (nouvelle_branche)
$ git branch
  main
* nouvelle_branche

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (nouvelle_branche)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$
```

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git branch test
$*
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git branch
* main
  nouvelle_branche
  test
```

Git

Les Remises (Stash) :

Les remises permettent de mettre de côté des modifications (ajout, modifications …) pour pouvoir les copier sur la branche actuelle ou une autre branche à n'importe quel moment. Cela fonctionne comme une liste d'attente.

1 Création d'une remise :

Pour effectuer la remise nous allons utiliser la commande « git stash save » :

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git add .

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   remise
        new file:   test


guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash save
Saved working directory and index state WIP on main: 869ad50 my 1st commit

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Cela va créer une remise et mettre en attente les modifications (*les fichiers ou modifications disparaitront de votre dossier local*).

2 Affichage des remises

La commande « git stash list » permet de lister les remises et nous indiquer leur id.

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash list
stash@{0}: WIP on main: 869ad50 my 1st commit
```

Git

3 Application de la remise :

La commande « git stash pop » va appliquer la dernière remise enregistrée et la supprimer de la liste des remises. Il est aussi possible de préciser l'id de la remise que l'on souhaite « pop ».

```
guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ touch a_remiser_2

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git add .

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash save
Saved working directory and index state WIP on main: 869ad50 my 1st commit

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash list
stash@{0}: WIP on main: 869ad50 my 1st commit
stash@{1}: WIP on main: 869ad50 my 1st commit

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash pop
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   a_remiser_2

Dropped refs/stash@{0} (a1d01a59668514ef3b88ebb5e41dabaa70822563)

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash list
stash@{0}: WIP on main: 869ad50 my 1st commit

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash pop stash@{0}
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   a_remiser_2
        new file:   remise
        new file:   test

Dropped stash@{0} (fbeceb11410036c96c64fc1184af9c0821da53e7)

guillaumerodrigues@EUR23-FORM-04 MINGW64 ~/Workspace/test_clone (main)
$ git stash list
```

Git

Correction de problèmes Git :

1 Modification du message du dernier commit :

Dans le cas où vous souhaitez modifier le message de votre dernier commit nous allons utiliser la commande ci-dessous :

```
$ git commit --amend -m "Votre nouveau message de commit"
```

2 Ajouter un fichier oublié dans le dernier commit :

Vous avez fait votre commit mais vous réalisiez que vous avez oublié un fichier. Ce n'est pas bien grave ! Nous allons réutiliser la commande git --amend, mais d'une autre manière. La fonction git --amend, si vous avez bien compris, permet de modifier le dernier commit.

Nous allons donc réutiliser cette fonction, mais sans le -m qui permettait de modifier son message.

Nous allons dans un premier temps ajouter notre fichier, et dans un deuxième temps réaliser le git --amend.

```
$ git add nom_fichier.extension ou git add * (si plusieurs)
```

```
$ git commit --amend --no-edit
```

Votre fichier a été ajouté à votre commit et grâce à la commande --no-edit que nous avons ajoutée, nous n'avons pas modifié le message du commit.

Pour résumer, git commit --amend vous permet de sélectionner le dernier commit afin d'y ajouter de nouveaux changements en attente. Vous pouvez ajouter ou supprimer des changements afin de les appliquer avec

Commit --amend.

Si aucun changement n'est en attente, --*amend* vous permet de `git commit --amend -`le dernier message de log du commit avec *-m*. exemple ci-dessous :

```
$ git commit --amend -m "Votre nouveau message de commit"
```

3 Restauration de la branche master (*optionnel voir page 22*):

Si vous avez effectué un commit (update des fichiers) non désiré on peut restaurer nos fichiers à un état antérieur.

Pour se faire on va lancer la commande suivante pour récupérer l'id du commit :

Git

-se positionner sur la branche master

```
$ git checkout master
```

-Afficher le log des commits pour récupérer l'identifiant avec la commande ci-dessous :

```
$ git log
```

-Exemple d'un commit affiché dans git log :

commit ***ca83a6dff817ec66f443420071545390a954664949*** *(identifiant)*

Author: ***nom_utilisateur*** <***mail_utilisateur***>

Date: Mon Mar 19 21:52:11 2019 -0700

-Récupérer l'identifiant ci-dessus :

-Assurez-vous de bien être sur la branche master.

-Taper la commande ci-dessous :

```
$ git reset --hard HEAD^
```

-Basculez sur la branche ou vous voulez copier les modifications :

```
$ git checkout nom_de_la_branche
```

-taper la commande ci-dessous : (on n'a besoin que des 8 premiers caractères de l'identifiant)

```
$ git reset --hard ca83a6df
```

-Cela va appliquer un commit (sauvegarde) sur la branche sélectionnée

4 Corriger un mauvais commit à distance :

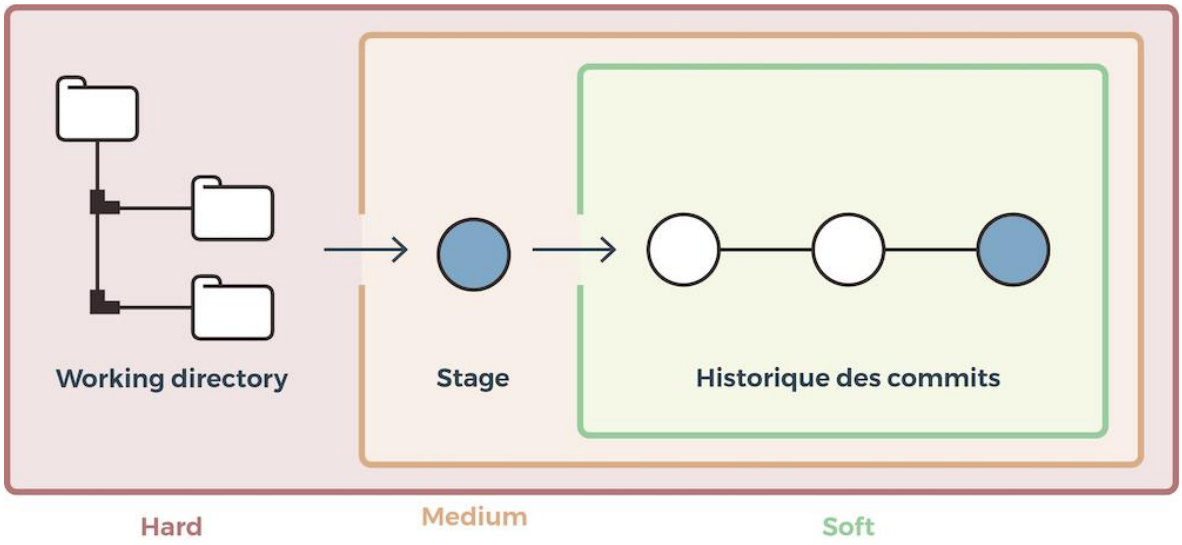
Pour corriger un mauvais commit (le dernier) envoyé avec un push sur votre Github saisir la commande ci-dessous :

```
$ git revert HEAD^
```

Git

Les 3 types de réinitialisation de GIT :

La commande git reset est un outil complexe et polyvalent pour **annuler les changements**. Elle peut être appelée de trois façons différentes, qui correspondent aux arguments de ligne de commande **--soft, --mixed et --hard**.



1 Commande Reset –hard :

Cette commande permet de revenir à n'importe quel commit mais il faut faire très attention car l'on **perd tous les commits suivants !!!**

Pour se faire il nous faut l'identifiant du commit que l'on obtient avec la commande :

```
$ git log
```

Puis on saisit la commande suivante :

```
$ git reset identifiant_commit --hard
```

Cette commande est à utiliser en dernier recours car l'on **perd tous les commits suivants** !!!!

Git

2 Commande Reset –mixed :

Le `git reset --mixed` va permettre de revenir juste après votre dernier commit ou le commit spécifier, sans supprimer vos modifications en cours. Il va par contre créer un HEAD détaché. Il permet aussi, dans le cas de fichiers indexés mais pas encore commités, de désindexer les fichiers.

Si rien n'est spécifié après git reset, par défaut il exécutera un `git reset --mixed HEAD~`

3 Commande Reset –soft :

Le `git reset --Soft` permet juste de se placer sur un commit spécifique afin de voir le code à un instant donné ou créer une branche partant d'un ancien commit. Il ne supprime aucun fichier, aucun commit, et ne crée pas de HEAD détaché.

Git

Scénarios d'erreurs :

1° cas J'ai créé une branche que je n'aurais pas dû créer :

Avant de créer une branche, vous devez créer votre branche principale (master). Pour créer la branche master, vous devez simplement ajouter un fichier et le commiter.

1 Créez un fichier "PremierFichier.txt" dans votre répertoire Test, et ajoutez-le avec la commande :

```
$ git add PremierFichier.txt
```

```
$ git commit ou git commit –m « message »
```

On vous demande alors d'indiquer le message du commit puis de valider. Pour valider le message, une fois que vous l'avez écrit, appuyez sur Echap (votre curseur va basculer sur la dernière ligne) et tapez :x où :x!

Cette commande va sauvegarder et quitter l'éditeur des messages de commit.

2 Nous allons maintenant créer une branche brancheTest avec la commande ci-dessous :

```
$ git branch brancheTest
```

Nous pouvons le vérifier avec la commande ci-dessous :

```
$ git branch
```

Youppiiii !

En fait, non, nous voulions ajouter nos fichiers avant de la créer et nous sommes maintenant bloqués avec cette branche que nous ne voulions pas tout de suite. Heureusement, il est très simple sous Git de supprimer une branche que nous venons de créer.

3 Pour cela, il suffit d'exécuter la commande :

```
$ git branch -d brancheTest
```

Attention, si toutefois vous avez déjà fait des modifications dans la branche que vous souhaitez supprimer, il faudra soit faire un commit de vos modifications, soit mettre vos modifications de côté.

4 Forcer la suppression en faisant :

```
$ git branch -D brancheTest
```

Attention : Forcer la suppression de cette manière entraînera la suppression de tous les fichiers et modifications que nous n'aurez pas commités sur cette branche.

2 ° cas J'ai modifié la branche principale :

Nous allons dans un premier temps voir ensemble le cas où vous avez modifié votre branche master mais que vous n'avez pas encore fait le commit, et nous verrons dans un second temps le cas où vous avez commité.

Vous avez modifié votre branche master avant de créer votre branche et vous n'avez pas fait le commit. Ce cas est un peu plus simple. Nous allons faire ce qu'on appelle une remise. La remise va permettre de mettre vos modifications de côté, le temps de créer votre nouvelle branche et ensuite appliquer cette remise sur la nouvelle branche.

Afin de voir comment cela fonctionne, allez sur votre branche master, modifiez des fichiers. Vous pouvez à tout moment voir à quel état sont vos fichiers en faisant :

```
$ git status
```

1 créer une remise :

```
$ git stash
```

Vous pouvez maintenant vous assurer que votre branche master est de nouveau propre, en faisant un nouveau

```
$ git status
```

Git

Vous devriez avoir :

```
$ git status
# On branch master
nothing to commit, working directory clean
```

2 créer notre branche brancheCommit :

```
$ git branch brancheCommit
```

3 basculer sur la nouvelle branche :

```
$ git checkout brancheCommit
```

Et finalement, nous allons pouvoir appliquer la remise, afin de récupérer nos modifications sur notre nouvelle branche.

4 réaliser la remise :

```
$ git stash apply
```

Cette commande va appliquer la dernière remise qui a été faite. Si pour une raison ou une autre, vous avez créé plusieurs remises, et que la dernière n'est pas celle que vous souhaitiez appliquer, pas de panique, il est possible d'appliquer une autre remise. Nous allons d'abord regarder la liste de nos remises.

5 afficher la liste des remises :

```
$ git stash list
```

Cette commande va nous retourner un "tableau" des remises avec des identifiants du style :

```
$ git stash list
stash@{0}: WIP on master: f337838 création de la branche master
```

Maintenant, admettons que vous ayez réalisé vos modifications et qu'en plus vous ayez fait le commit. Le cas est plus complexe, puisque vous avez enregistré vos modifications sur la branche master, alors que vous ne deviez pas.

Git

6 modifiez des fichiers, et réalisez-le commit :

Nous allons devoir aller analyser vos derniers commits avec la fonction **git log**, afin de pouvoir récupérer l'identifiant du commit que l'on appelle couramment le *hash*. Par défaut, **git log** va vous lister par ordre chronologique inversé tous vos commits réalisés.

```
$ git log
```

```
commit ca83a6dff817ec66f443420071545390a954664949
Author: Mathieu <mathieu.mith@laposte.com>
Date: Mon Mar 19 21:52:11 2019 -0700
```

Maintenant que vous disposez de votre identifiant, gardez-le bien de côté. Vérifiez bien que vous êtes sur votre branche master

```
$ git checkout master
```

7 réalisez la commande suivante :

```
$ git reset --hard HEAD^
```

Cette ligne de commande va permettre de supprimer de la branche master votre dernier commit. Le Head^ indique que c'est bien le dernier commit que nous voulons supprimer.

8 nous allons maintenant créer notre nouvelle branche :

```
$ git branch brancheCommit
```

9 basculer sur cette branche :

```
$ git checkout brancheCommit
```

Maintenant que nous sommes sur la bonne branche, nous allons de nouveau faire un **git reset**, mais celui-ci va permettre d'appliquer ce commit sur notre nouvelle branche ! Il n'est pas nécessaire d'écrire l'identifiant en entier. Seuls les 8 premiers caractères sont nécessaires.

```
$ git reset --hard identifiant
```

Notre cas est résolu.

3 ° cas je souhaite changer le message de mon commit :

Lorsque l'on travaille sur un projet avec Git, il est très important, lorsque l'on propage les modifications, de bien marquer dans le message descriptif les modifications que l'on a effectuées. Si jamais ne vous faites une erreur dans l'un de vos messages de commit, il est tout à fait possible de changer le message après coup.

Attention cette commande va fonctionner sur votre dernier commit.

Imaginons que vous veniez de faire un commit et que vous ayez fait une erreur dans votre message. L'exécution de cette commande, lorsqu'aucun élément n'est encore modifié, vous permet de modifier le message du commit précédent sans modifier son instantané. L'option **-m** permet de transmettre le nouveau message.

```
$ git commit --amend -m "Votre nouveau message de commit"
```

On peut vérifier avec la commande **git log**.

Git

4 ° cas J'ai oublié un fichier dans mon dernier commit :

Imaginons maintenant que vous ayez fait votre commit mais que vous réalisiez que vous avez oublié un fichier. Nous allons réutiliser la commande git **--amend**, mais d'une autre manière. La fonction **git --amend**, si vous avez bien compris, permet de modifier le dernier commit.

Nous allons donc réutiliser cette fonction, mais sans le **-m** qui permettait de modifier son message.

1 ajouter votre fichier, et dans un deuxième temps réaliser le **git --amend** :

```
$ git add nom du fichier.extension
```

```
$ git --amend --no-edit
```

Votre fichier a été ajouté à votre commit et grâce à la commande --no-edit que nous avons ajoutée, nous n'avons pas modifié le message du commit.

Pour résumer, **git commit --amend** vous permet de sélectionner le dernier commit afin d'y ajouter de nouveaux changements en attente. Vous pouvez ajouter ou supprimer des changements afin de les appliquer avec commit --amend. Si aucun changement n'est en attente, **--amend** vous permet de modifier le dernier message de log du commit avec **-m**.

Corrigez vos erreurs en local et à distance :

Vous avez par mégarde push des fichiers erronés. Le problème, c'est que maintenant ce n'est plus que sur votre dépôt local, mais à disposition de tout le monde.

Il est possible d'annuler son commit public avec la commande **Git revert**. L'opération Revert annule un commit en créant un nouveau **commit**. C'est une méthode sûre pour **annuler des changements**, car elle ne risque pas de **réécrire l'historique du commit**.

2 utiliser la commande :

```
$ git revert HEAD^
```

Nous avons maintenant revert notre dernier commit public et cela a créé un nouveau commit d'annulation. Cette commande n'a donc **aucun impact sur l'historique** ! Par conséquent, il vaut mieux utiliser `git revert` pour annuler des changements apportés à une branche publique, et `git reset` pour faire de même, mais sur une branche privée.

Git

Fusion (merge):

Comment fonctionne la fusion sous Git ?

Il est très courant sous Git de vouloir fusionner le travail fait sur différentes branches. Pour cela, nous avons la fonction **Merge**. Un `git merge` ne devrait être utilisé que pour la récupération fonctionnelle, intégrale et finale d'une branche dans une autre, afin de préserver un graphe d'historique sémantiquement cohérent et utile, lequel représente une véritable valeur ajoutée. Comme son nom l'indique, `merge` réalise une **fusion**. `git merge` va combiner plusieurs séquences de commits en un historique unifié. Le plus souvent, `git merge` est utilisé pour combiner deux branches. `git merge` va créer un nouveau commit de merge.

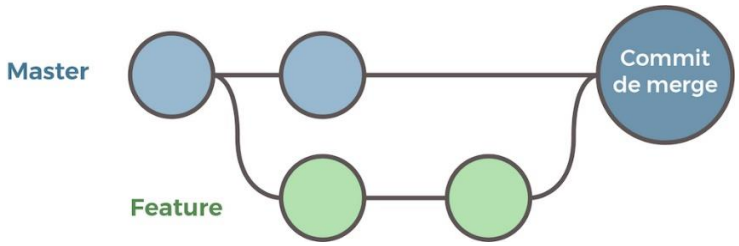
Imaginons que vous ayez votre branche master et une branche "Nouvelle fonctionnalité". Nous souhaitons maintenant faire un merge de cette branche de fonctionnalité dans la branche master. Appeler cette commande permettra de merger la fonctionnalité de branche spécifiée dans la branche courante, disons master.

Attention Il faut toujours préparer le terrain avant de réaliser un merge !

Vous devez toujours vous assurer d'être sur **la bonne branche**. Pour cela, vous pouvez réaliser un `git status`. Si vous n'êtes pas sur la bonne, réalisez un `git checkout`, pour changer de branche. Maintenant que le terrain est prêt, vous pouvez réaliser votre merge.

```
$ git merge nom_branche
```

Votre branche Nouvelle fonctionnalité va être fusionnée sur la branche master en créant un nouveau commit.



Si les deux branches que vous essayez de fusionner modifient toutes les deux la même partie du même fichier, Git ne peut pas déterminer la version à utiliser. Lorsqu'une telle situation se produit, Git s'arrête avant le commit de merge, afin que vous puissiez résoudre manuellement les conflits.

Merge une branche utiliser les commandes suivantes :

Git

1 Se déplacer sur la branche à merge :

```
$ git checkout branche_qui_va_recevoir_le_merge
```

2 Merge la branche :

```
$ git merge Nom_branche_à_merge
```

Ressources documentaires

Site officiel : https://git-scm.com/docs

Tutoriels Atlassian : https://www.atlassian.com/git/tutorials

Documentation Gitlab (attention à ne regarder que la partie concernant Git, le reste concerne Gitlab) : https://docs.gitlab.com/ee/topics/git/

Et comme d'habitude, cherchez vous la killer cheat sheet.

Quelques pistes :

https://training.github.com/downloads/fr/github-git-cheat-sheet.pdf

https://education.github.com/git-cheat-sheet-education.pdf

https://about.gitlab.com/images/press/git-cheat-sheet.pdf

https://atlassian.com/git/tutorials/atlassian-git-cheatsheet